



Eberhard Wolff

Microservices

Grundlagen flexibler Softwarearchitekturen

dpunkt.verlag



Eberhard Wolff arbeitet seit mehr als fünfzehn 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 Büchern 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.

Papier
plus⁺
PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus⁺:

www.dpunkt.de/plus

Microservices

Grundlagen flexibler Softwarearchitekturen

Eberhard Wolff



Eberhard Wolff

eberhard.wolff@gmail.com

Lektorat: René Schönenfeldt

Copy-Editing: Sandra Gottmann (Münster-Nienberge)

Herstellung: Birgit Bäuerlein

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:

Buch 978-3-86490-313-7

PDF 978-3-86491-841-4

ePub 978-3-86491-842-1

mobi 978-3-86491-843-8

1. Auflage 2016

Copyright © 2016 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.

5 4 3 2 1 0

Inhaltsverzeichnis

1 Vorwort

- 1.1 Überblick über Microservices
- 1.2 Warum Microservices?

Teil I Motivation und Grundlagen

2 Einleitung

- 2.1 Überblick über das Buch
- 2.2 Für wen ist das Buch?
- 2.3 Übersicht über die Kapitel
- 2.4 Essays
- 2.5 Pfade durch das Buch
- 2.6 Danksagung
- 2.7 Links & Literatur

3 Microservice-Szenarien

- 3.1 Eine E-Commerce-Legacy-Anwendung modernisieren
- 3.2 Ein neues Signalsystem entwickeln
- 3.3 Fazit

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

4 Was sind Microservices?

- 4.1 Größe eines Microservice
- 4.2 Das Gesetz von Conway
- 4.3 Domain-Driven Design und Bounded Context
- 4.4 Microservice: Mit UI?
- 4.5 Fazit
- 4.6 Links & Literatur

5 Gründe für Microservices

- 5.1 Technische Vorteile
- 5.2 Organisatorische Vorteile
- 5.3 Vorteile aus Geschäftssicht
- 5.4 Fazit
- 5.5 Links & Literatur

6 Herausforderungen bei Microservices

6.1 Technische Herausforderungen

6.2 Architektur

6.3 Infrastruktur und Betrieb

6.4 Fazit

6.5 Links & Literatur

7 Microservices und SOA

7.1 Was ist SOA?

7.2 Unterschiede zwischen SOA und Microservices

7.3 Fazit

7.4 Links & Literatur

Teil III Microservices umsetzen

8 Architektur von Microservice-Systemen

8.1 Fachliche Architektur

8.2 Architekturmanagement

8.3 Techniken zum Anpassen der Architektur

8.4 Microservice-Systeme weiterentwickeln

8.5 Microservice und Legacy-Anwendung

8.6 Event-driven Architecture

8.7 Technische Architektur

8.8 Konfiguration und Koordination

8.9 Service Discovery

8.10 Load Balancing

8.11 Skalierbarkeit

8.12 Sicherheit

8.13 Dokumentation und Metadaten

8.14 Fazit

8.15 Links und Literatur

9 Integration und Kommunikation

9.1 Web und UI

9.2 REST

9.3 SOAP und RPC

9.4 Messaging

9.5 Datenreplikation

9.6 Schnittstellen: intern und extern

9.7 Fazit

9.8 Links & Literatur

10 Architektur eines Microservice

10.1 Fachliche Architektur

10.2 CQRS

10.3 Event Sourcing

10.4 Hexagonale Architekturen

10.5 Resilience und Stabilität

10.6 Technische Architektur

10.7 Fazit

10.8 Links & Literatur

11 Testen von Microservices und Microservice-Systemen

11.1 Warum testen?

11.2 Wie testen?

11.3 Risiken beim Deployment minimieren

11.4 Tests des Gesamtsystems

11.5 Legacy-Anwendungen mit Microservices testen

11.6 Tests einzelner Microservices

11.7 Consumer-Driven Contract Test

11.8 Technische Standards testen

11.9 Fazit

11.10 Links & Literatur

12 Betrieb und Continuous Delivery von Microservices

12.1 Herausforderungen beim Betrieb von Microservices

12.2 Logging

12.3 Monitoring

12.4 Deployment

12.5 Steuerung

12.6 Infrastrukturen

12.7 Fazit

12.8 Link & Literatur

13 Organisatorische Auswirkungen der Architektur

13.1 Organisatorische Vorteile von Microservices

13.2 Alternativer Umgang mit dem Gesetz von Conway

13.3 Spielräume schaffen: Mikro- und Makro-Architektur

13.4 Technische Führung

13.5 DevOps

13.6 Schnittstelle zu den Fachbereichen

13.7 Wiederverwendbarer Code

13.8 Microservices ohne Organisationsänderung?

13.9 Fazit

13.10 Links & Literatur

Teil IV Technologien

14 Ein Beispiel für eine Microservices-Architektur

14.1 Fachliche Architektur

14.2 Basistechnologien

14.3 Build

14.4 Deployment mit Docker

14.5 Vagrant

14.6 Docker Machine

14.7 Docker Compose

14.8 Service Discovery

14.9 Kommunikation

14.10 Resilience

14.11 Load Balancing

14.12 Integration anderer Technologien

14.13 Tests

14.14 Fazit

14.15 Links & Literatur

15 Technologien für Nanoservices

15.1 Warum Nanoservices?

15.2 Definition Nanoservice

15.3 Amazon Lambda

15.4 OSGi

15.5 Java EE

15.6 Vert.x

15.7 Erlang

15.8 Seneca

15.9 Fazit

15.10 Links und Literatur

16 Wie mit Microservices loslegen?

16.1 Warum Microservices?

16.2 Wege zu Microservices

16.3 Microservice: Hype oder Realität?

16.4 Fazit

Index

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

Im Mittelpunkt des Buchs stehen Microservices – ein Ansatz zur Modularisierung von Software.

Microservice: vorläufige Definition

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.

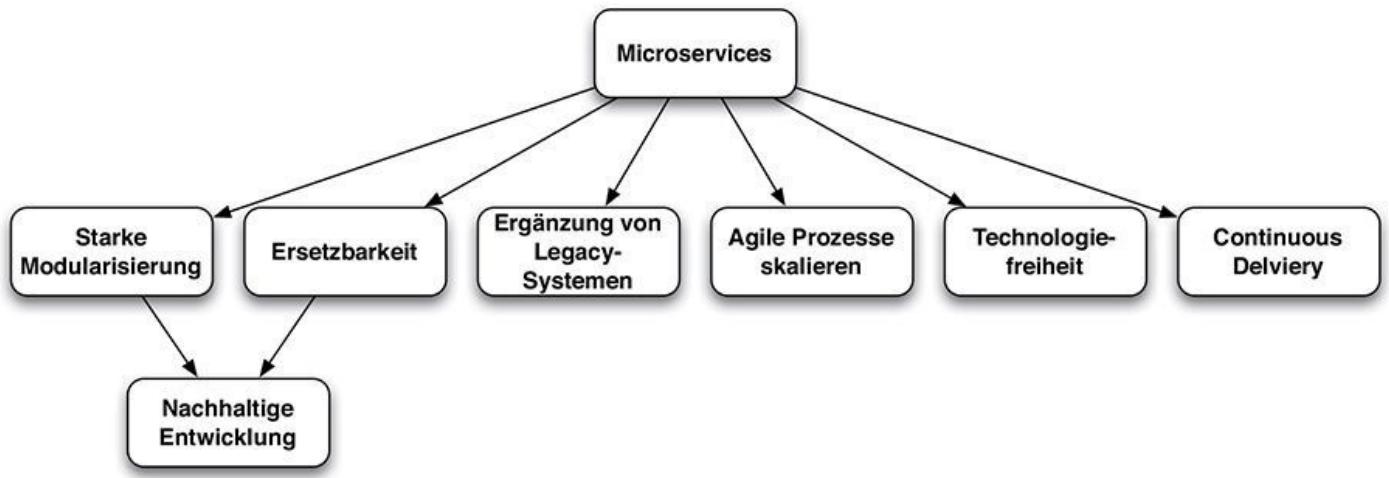
Dieser Ansatz betrachtet die Größe des Microservice nicht. Trotz des Namens »Microservice« ist die Größe für eine grobe Definition nicht so entscheidend.

Microservices grenzen sich von Deployment-Monolithen 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:

- 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.
- 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.
- 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.

- Der Einstieg in eine Microservices-Architektur ist Legacy-Anwendung erweitern
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.
- Microservices erlauben ein besseres Time-to-Market. 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 Deployment-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. Continuous Delivery

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

- Die Architektur des Systems besteht aus den *Beziehungen sind versteckt.* Beziehungen der Services. Aber ohne Weiteres ist nicht klar, welcher Microservice welchen anderen aufruft. Dadurch wird Architekturarbeit zu einer Herausforderung.
- Die starke Modularisierung hat auch Nachteile: *Refactoring ist schwierig.* 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.
- 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. *Fachliche Architektur ist wichtig.*
- 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. *Betrieb ist 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. *Verteilte Systeme sind komplex.*

- [1] <http://jandiandme.blogspot.com/2006/10/jaoo-2006-werner-vogelscto-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 Buches. 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 *Manager* 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.
- *Entwickler* erhalten eine umfassende Einleitung in die technischen Aspekte und können damit die notwendigen Fähigkeiten aufbauen, 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* lernen Microservices aus einer *Architekten* 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

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](#) 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.

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.

[Teil III](#)

- 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 – Ein Beispiel für eine Microservices-Architektur	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 Gloge,

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 Links & Literatur

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

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 angemessen 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.

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.

Es gibt Teams, die an verschiedenen neuen Features

Langsame Continuous Delivery Pipeline

Parallele Arbeit ist kompliziert.

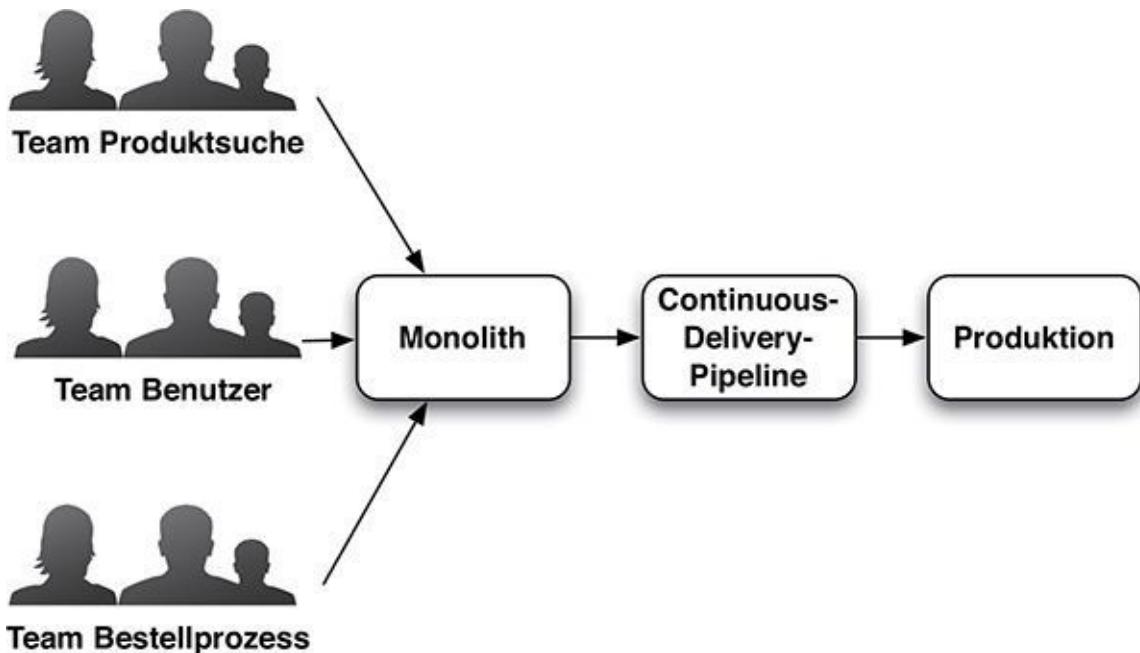
arbeiten. Aber die parallele Arbeit ist kompliziert: Die Struktur der Software ist dafür zu schlecht. Die einzelnen Module sind zu schlecht separiert und haben zu viele Abhängigkeiten untereinander. Da alles nur gemeinsam deployt werden kann, muss der gesamte Deployment-Monolith vorher auch getestet 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.

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 Versionskontrolle 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 anderen 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 Log-dateien 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.

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.

Abb. 3–2 Entkoppelte Arbeit durch Microservices

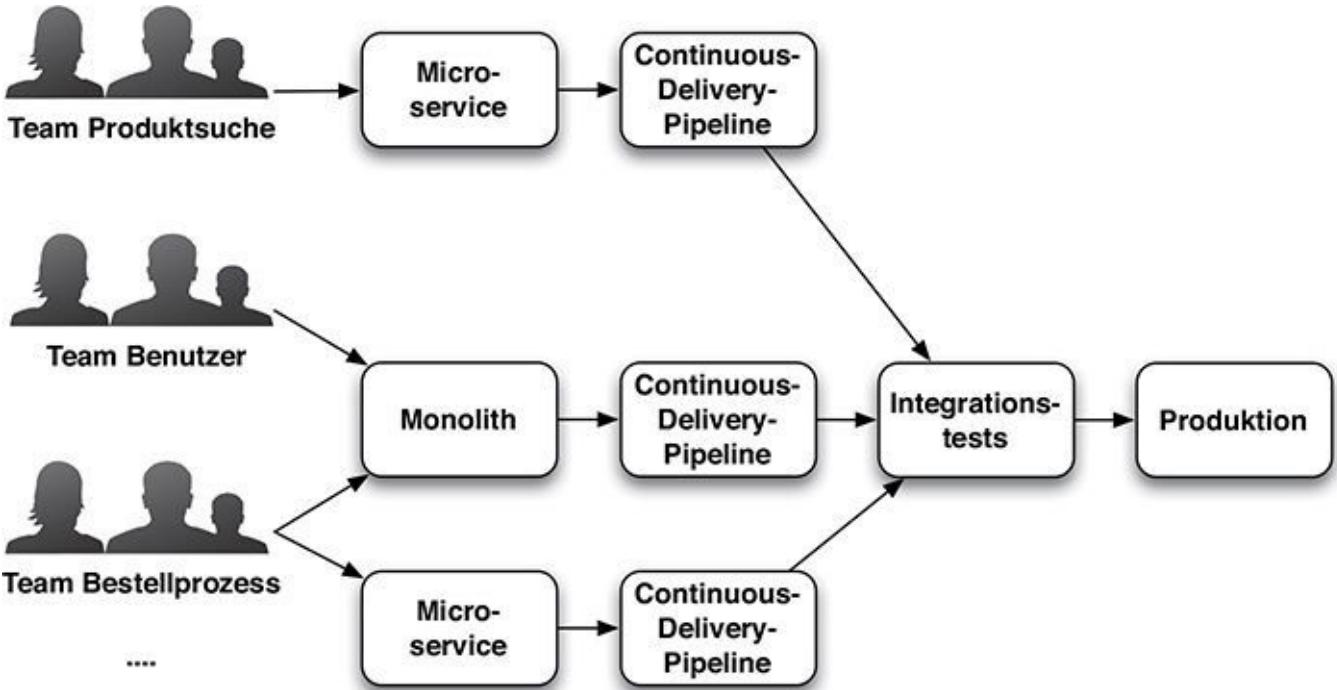


Abbildung 3-2 zeigt den aktuellen Stand: Die Produktsuche arbeitet auf ihrem eigenen Microservice und vollständig unabhängig vom Deployment-Monolithen. Eine Koordinierung mit den anderen Teams ist kaum noch notwendig. Nur im letzten Schritt des Deployments müssen der Deployment-Monolith und die Microservices 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.

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.

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 Einsatzkontakte sehr attraktiv.

Dieses Beispiel verdeutlicht auch den Einfluss von Microservices auf die Organisation. Die Teams arbeiten

Schnelle und unabhängige Entwicklung neuer Features

Einfluss auf die Organisation

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

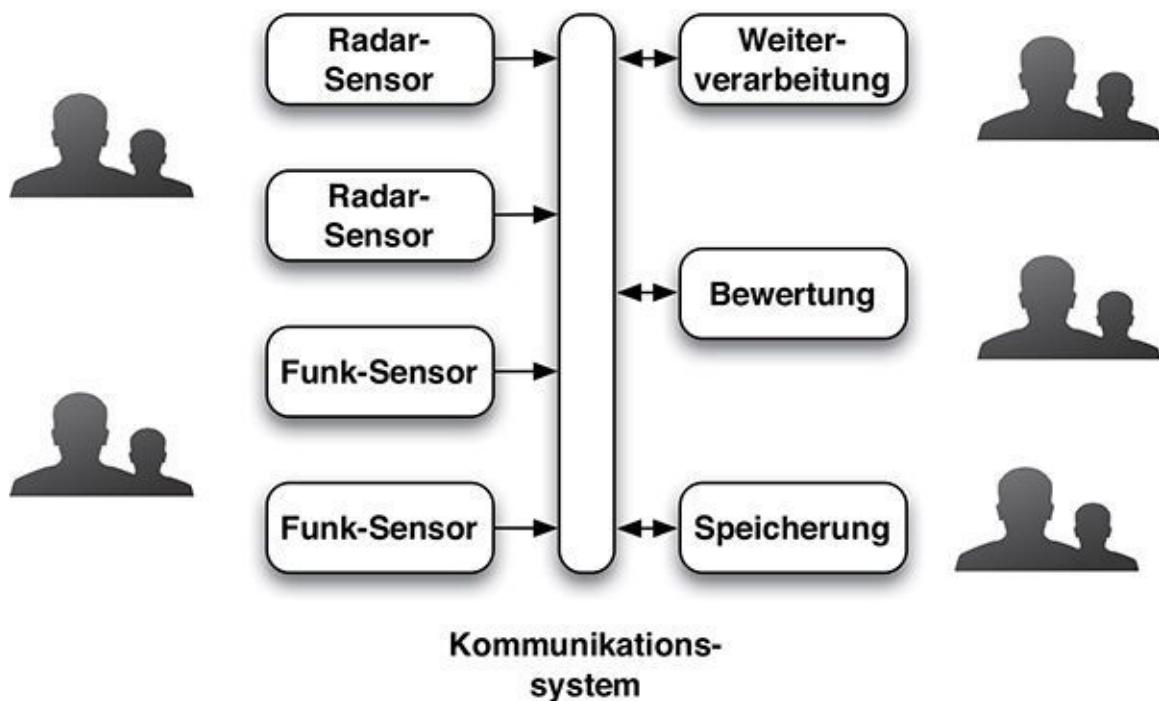
allem bei Firmen mit Internet-basiertem Geschäftsmodell. So hat der Ansatz schon lange seine praktischen Vorteile bewiesen. Außerdem haben Microservices Synergieeffekte mit anderen modernen Ansätzen wie Continuous Delivery, Cloud oder DevOps.

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.

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.

Verteiltes System

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.

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.

Technologie-Stack pro Team

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 Normalfall ist.

Integration anderer Systeme

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 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.

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 Technologievelfalt, 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.

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.

4 Was sind Microservices?

Der [Abschnitt 1.1](#) hat schon eine erste Definition des Begriffs Microservice gegeben. Es gibt aber noch andere Möglichkeiten, Microservices zu definieren. Die unterschiedlichen Definitionen zeigen die Eigenschaften von Microservices und geben damit an, aus welchen Gründen Microservices vorteilhaft sind. Am Ende des Kapitels sollte eine eigene persönliche Definition des Begriffs Microservice stehen – abhängig vom eigenen Szenario.

Den Begriff »Microservice« betrachtet das Kapitel aus unterschiedlichen Perspektiven:

- In [Abschnitt 4.1](#) steht die Größe des Microservice im Mittelpunkt.
- [Abschnitt 4.2](#) stellt eine Beziehung zwischen Microservices, Architektur und Organisation mithilfe des Gesetzes von Conway her.
- Am Ende zeigt [Abschnitt 4.3](#) eine fachliche Aufteilung von Microservices anhand von Domain-Driven Design (DDD) und BOUNDED CONTEXT.
- Der [Abschnitt 4.4](#) erläutert, warum Microservices eine UI enthalten sollten.

4.1 Größe eines Microservice

Der Name »Microservices« verrät schon, dass es um die Servicegröße geht – offensichtlich sollen die Services klein sein.

Ein Möglichkeit, die Größe eines Microservice zu definieren, sind Lines of Code (LoC, Codezeilen) [1]. Ein solcher Ansatz hat jedoch einige Probleme:

- Er hängt von der verwendeten Programmiersprache ab. Einige Sprachen benötigen mehr Code, um dasselbe auszudrücken – und Microservices sollen gerade nicht den Technologie-Stack fest definieren. Dementsprechend ist eine Definition anhand dieser Metrik kaum sinnvoll.
- Schließlich geht es bei Microservices um einen Architekturansatz. Architektur sollte sich aber nach den Gegebenheiten in der fachlichen Domäne richten und nicht so sehr nach technischen Größen wie LoC. Auch aus diesem Grund ist eine Bestimmung der Größe anhand der Codezeilen kritisch zu sehen.

LoC können trotz aller Kritik ein Indikator für die Größe eines Microservice sein. Es stellt aber dennoch die Frage nach der idealen Größe eines Microservice. Wie viele LoC darf ein Microservice haben? Auch wenn es keine absoluten Richtwerte gibt, gibt es dennoch Einflussfaktoren, die für größere oder kleinere Microservices sprechen.

Ein Faktor ist die Modularisierung. Teams entwickeln Software in Modulen, um die Komplexität handhabbar zu machen: Statt die gesamte Software zu verstehen, muss ein Entwickler nur jeweils das Modul verstehen, das er gerade ändert, und das Zusammenspiel der Module. Nur so kann ein Team trotz der Komplexität eines typischen Software-Systems überhaupt produktiv

arbeiten. Es gibt in der Praxis oft Probleme, weil Module größer werden als ursprünglich geplant. Dadurch werden die Module schwer zu verstehen und schwer zu warten, weil Änderungen ein Verständnis der Software voraussetzen. Also ist es sinnvoll, Microservices möglichst klein zu halten.

Dagegen spricht, dass Microservices im Gegensatz zu vielen anderen Modularisierungsansätzen einen Overhead haben:

Microservices laufen in eigenständigen Prozessen. Verteilte Kommunikation
Daher ist die Kommunikation zwischen Microservices verteilte Kommunikation über das Netzwerk. Für diese Art von Systemen gibt es die »erste Regel für verteilte Systeme«. Sie besagt, dass Systeme möglichst nicht verteilt werden sollen [2]. Der Grund dafür ist, dass der Aufruf eines anderen Systems über das Netzwerk einige Größenordnungen langsamer ist als der direkte Aufruf im selben Prozess. Zu der reinen Latenzzeit kommen noch der Aufwand für die Serialisierung und Deserialisierung der Parameter und Ergebnisse hinzu. Diese Operationen dauern nicht nur lange, sondern kosten auch CPU-Kapazität.

Hinzu kommt, dass verteilte Aufrufe fehlschlagen können, weil das Netzwerk gerade nicht verfügbar oder der aufgerufene Server nicht erreichbar ist – weil er beispielsweise abgestürzt ist. Das macht die Implementierung verteilter Systeme noch komplexer, denn der Aufrufer muss mit diesen Fehlern sinnvoll umgehen.

Die Erfahrung zeigt, dass Microservices-Architekturen trotz dieser Probleme funktionieren [3]. Wenn Microservices besonders klein gewählt werden, gibt es mehr verteilte Kommunikation und das System wird insgesamt langsamer. Das spricht für größere Microservices. Wenn ein Microservice eine UI enthält und einen guten funktionalen Schnitt hat, kann er in den meisten Fällen ohne den Aufruf anderer Microservices auskommen, weil alle Bestandteile der Fachlichkeit in einem Microservice umgesetzt sind. Das Vermeiden verteilter Kommunikation ist ein weiterer Grund, Systeme fachlich sauber aufzubauen.

Microservices nutzen Verteilung auch dazu, um die Nachhaltige Architektur Architektur durch Aufteilung in einzelne Microservices nachhaltig zu gestalten: Es ist viel schwieriger, einen Microservice zu nutzen als eine Klasse. Der Entwickler muss sich dazu mit der Verteilungstechnologie beschäftigen und die Schnittstelle des Microservice nutzen. Gegebenenfalls muss er für Tests Vorkehrungen treffen, um den aufgerufenen Microservice in Tests einzubeziehen oder durch einen Stub zu ersetzen. Schließlich muss der Entwickler mit dem Team Kontakt aufnehmen, das für den Microservice zuständig ist.

Eine Klasse in einem Deployment-Monolith zu nutzen, ist hingegen sehr einfach – auch wenn sie aus einem ganz anderen Bereich des Monolithen kommt und von einem anderen Team verantwortet wird. Weil es so einfach ist, eine Abhängigkeit zwischen zwei Klassen umzusetzen, schleichen sich in Deployment-Monolithen oft unbeabsichtigte Abhängigkeiten ein. Bei Microservices ist eine Abhängigkeit komplizierter umzusetzen und kann sich daher nicht einfach so einschleichen.

Allerdings führen die Grenzen zwischen den Refactoring Microservices auch zu Herausforderungen beispielsweise

beim Refactoring. Wenn sich herausstellt, dass eine bestimmte Funktionalität in einem Microservice nicht sinnvoll untergebracht ist, muss sie in einen anderen Microservice verschoben werden. Wenn der Ziel-Microservice in einer anderen Programmiersprache geschrieben ist, entspricht das letztendlich einer Neuimplementierung. Solche Probleme entfallen, wenn die Funktionalitäten innerhalb eines Microservice verschoben werden sollen. Auch dieser Faktor spricht eher für größere Microservices. Dieser Themenkomplex steht im Mittelpunkt von [Abschnitt 8.3](#).

Das unabhängige Deployment der Microservices und Teamgröße die Aufteilung in Teams ergeben eine obere Grenze für die Größe eines Microservice. Ein Team soll in einem Microservice Features unabhängig von anderen Teams implementieren und in Produktion bringen können. Dadurch erlaubt die Architektur, die Entwicklung zu skalieren, ohne dass dabei die Teams zu viel koordinieren müssen.

Ein Team muss Features unabhängig von den anderen Teams umsetzen können. Es scheint daher zunächst so, dass der Microservice so groß sein muss, dass Features sinnvoll in dem Microservice implementiert werden können. Wenn die Microservices kleiner sind, kann ein Team aber einfach für mehrere Microservices zuständig sein, die zusammen die Implementierung von fachlichen Features erlauben. Eine untere Grenze für die Größe ergibt sich durch das unabhängige Deployment und die Aufteilung in Teams nicht.

Wohl aber eine obere Grenze: Wenn der Microservice so groß ist, dass er von einem Team nicht mehr weiterentwickelt werden kann, ist er zu groß. Ein Team sollte dabei eine Größe haben, wie sie für agile Prozesse besonders gut funktioniert. Das sind typischerweise drei bis neun Personen. Also darf ein Microservice auf keinen Fall so groß werden, dass ein Team ihn nicht mehr alleine weiterentwickeln kann. Neben der Größe spielt auch die Menge an Features eine Rolle, die in einem Microservices implementiert werden müssen. Wenn gerade sehr viele Änderungen notwendig sind, kann ein Team schnell überlastet sein. [Abschnitt 13.2](#) zeigt Alternativen, um mehreren Teams die Arbeit an einem Microservice zu erlauben. Dennoch sollte ein Microservice nicht so groß sein, dass mehrere Teams an ihm arbeiten müssen.

Ein weiterer Einflussfaktor auf die Größe eines Infrastruktur Microservice ist die Infrastruktur. Jeder Microservice muss unabhängig deployt werden können. Er muss eine Continuous-Delivery-Pipeline haben und eine Infrastruktur zum Ausführen des Microservice, die nicht nur in der Produktion, sondern auch in den verschiedenen Test-Stages vorhanden sein muss. Zu der Infrastruktur können auch Datenbanken oder Application Server gehören. Und es muss auch ein Build-System für den Microservice geben. Der Code des Microservice muss unabhängig von den anderen Microservices versioniert werden. Also muss es ein Projekt in der Versionskontrolle für den Microservice geben.

Abhängig davon, wie aufwendig es ist, für einen Microservice eine Infrastruktur bereitzustellen, kann die sinnvolle Größe eines Microservice unterschiedlich sein. Wenn die Größe besonders klein gewählt wird, wird das System in viele Microservices aufgeteilt und es müssen entsprechend mehr Infrastrukturen bereitgestellt werden. Bei größeren Microservices gibt es im System weniger von ihnen und es müssen auch nicht so viele Infrastrukturen bereitgestellt werden.

Build und Deployment der Microservices sollten sowieso automatisiert sein. Es kann aber dennoch aufwendig sein, alle benötigten Bestandteile der Infrastruktur für einen Microservice bereitzustellen. Wenn das Aufsetzen einer Infrastruktur für einen neuen Microservice automatisiert ist, sinkt der Aufwand für die Bereitstellung von Infrastruktur für neue Microservices. So kann die minimal mögliche Größe eines Microservice weiter reduziert werden. Unternehmen, die schon länger mit Microservices arbeiten, vereinfachen meistens das Erstellen neuer Microservices durch die automatisierte Bereitstellung von Infrastruktur.

Außerdem gibt es Technologien, mit denen der Overhead der Infrastruktur so weit sinkt, dass wesentlich kleinere Microservices möglich sind – dann allerdings mit einigen Einschränkungen. Solche Nanoservices behandelt [Kapitel 15](#).

Ein Microservice sollte möglichst einfach zu ersetzen Ersetzbarkeit sein. Das kann sinnvoll sein, wenn die Technologie des Microservice veraltet ist oder der Code des Microservice von so schlechter Qualität ist, dass er nicht mehr weiterentwickelt werden kann. Die Ersetzbarkeit von Microservices ist ein Vorteil gegenüber monolithischen Anwendungen, die kaum zu ersetzen sind. Wenn ein Monolith nicht mehr wartbar ist, muss er entweder mit erheblichen Kosten weiterentwickelt werden oder es muss doch eine Migration stattfinden, die auch sehr kostspielig ist. Je kleiner ein Microservice ist, desto einfacher ist es, ihn durch eine Neuimplementierung zu ersetzen. Über einer bestimmten Grenze ist der Microservice kaum noch zu ersetzen, weil es dieselben Herausforderungen wie bei einem Monolithen gibt. Ersetzbarkeit begrenzt die Größe eines Microservice also nach oben.

Transaktionen haben die ACID-Eigenschaften:

Transaktionen und Konsistenz

- Atomizität bedeutet, dass die Transaktion entweder ganz oder gar nicht ausgeführt wird. Bei einem Fehler werden alle Änderungen wieder rückgängig gemacht.
- Consistency (Konsistenz) bedeutet, dass vor und nach der Ausführung der Transaktion die Daten konsistent sind – also beispielsweise keine Validierungen verletzt sind.
- Isolation bedeutet, dass die Operationen der Transaktionen voneinander getrennt sind.
- Durability steht für Dauerhaftigkeit: Die Änderungen in der Transaktion werden gespeichert und stehen auch nach einem Absturz noch zur Verfügung.

Innerhalb eines Microservice können Änderungen in einer Transaktion stattfinden. Ebenso kann die Konsistenz der Daten in einem Microservice sehr einfach garantiert werden. Über einen Microservice hinaus wird das schwierig. Dann ist eine übergreifende Koordination notwendig. Bei einem Zurückrollen einer Transaktion müssten alle Änderungen in allen Microservices rückgängig gemacht werden. Das ist aufwendig und schwer umzusetzen, weil die Entscheidung, ob Änderungen rückgängig gemacht werden sollen oder nicht, garantiert zugestellt werden muss. Kommunikation in Netzwerken ist aber unzuverlässig. Bis zur Entscheidung, ob die Änderung erfolgen darf, sind weitere Änderungen an den Daten ausgeschlossen. Gegebenenfalls ist es nach weiteren Änderungen sonst schon nicht mehr möglich, eine bestimmte Änderung noch zurückzunehmen. Wenn Microservices aber länger keine Änderungen an Daten

durchführen können, reduziert das den Durchsatz des Systems.

Allerdings sind Transaktionen bei der Kommunikation über Messaging-Systeme möglich (siehe [Abschnitt 9.4](#)). Mit diesem Ansatz sind Transaktionen auch ohne enge Bindung der Microservices möglich.

Neben Transaktionen ist auch die Konsistenz der Daten wichtig. Eine Bestellung muss beispielsweise auch irgendwann als Umsatz verbucht werden. Nur dann sind Umsatz und Bestelldaten konsistent. Die Konsistenz der Daten kann nur mit einer engen Abstimmung erreicht werden. Konsistenz von Daten kann über Microservices hinweg kaum zugesichert werden. Das bedeutet nicht, dass der Umsatz für die Bestellung nicht irgendwann gebucht wird – aber nicht genau zum selben Zeitpunkt und vielleicht auch nicht innerhalb einer Minute nach Bearbeitung der Bestellung. Schließlich erfolgt die Kommunikation über das Netzwerk – und ist damit langsam und unzuverlässig.

Änderungen von Daten innerhalb einer Transaktion und Konsistenz der Daten sind nur möglich, wenn alle betroffenen Daten in einem Microservice sind. Damit stellen sie eine untere Grenze für einen Microservice dar: Wenn Transaktionen mehrere Microservices umfassen sollen und Konsistenz der Daten über mehrere Microservices notwendig ist, sind die Microservices zu klein gewählt.

Zumindest bei Transaktionen gibt es noch eine Alternative: Wenn eine Änderung an den Daten später zurückgerollt werden muss, dann können dafür Kompensationstransaktionen genutzt werden.

Kompensationstransaktionen über Microservices hinweg

Das klassische Beispiel für eine verteilte Transaktion ist eine Reisebuchung, die aus einem Hotel, einem Mietwagen und einem Flug besteht. Nur alles zusammen soll gebucht werden oder nichts von allem. In realen Systemen und auch bei Microservices wird die Funktionalität in drei Microservices aufgeteilt, weil es drei fachlich sehr unterschiedliche Aufgaben sind. Dann wird bei den Systemen nachgefragt, ob das gewünschte Hotelzimmer, der gewünschte Mietwagen und der gewünschte Flug verfügbar sind. Danach wird dann alles reserviert. Wenn plötzlich beispielsweise das Hotel nicht mehr verfügbar ist, muss auch die Reservierung für den Flug und den Mietwagen rückgängig gemacht werden. In der realen Welt werden die Firmen für die Stornierung der Buchung jedoch wahrscheinlich eine Gebühr berechnen. Damit ist die Stornierung nicht einfach ein technisches Ereignis hinter den Kulissen wie das Zurückrollen einer Transaktion, sondern ein Geschäftsprozess. Das ist viel besser mit einer Kompensationstransaktion abbildbar. Mit diesem Ansatz sind auch Transaktionen über mehrere Elemente in Microservice-Umgebungen umsetzbar, ohne dass es eine enge technische Bindung gibt. Eine Kompensationstransaktion ist einfach ein normaler Service-Aufruf. Sowohl technische als auch geschäftliche Gründe können für eine Nutzung von Mechanismen wie Kompensationstransaktionen über Microservices sprechen.

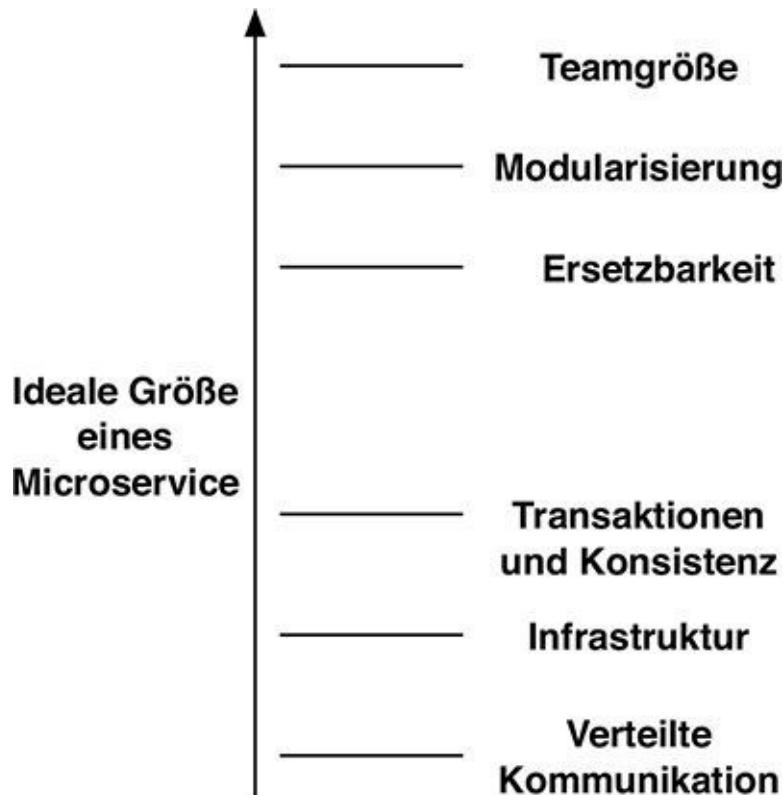
Es ergeben sich die folgenden Einflussfaktoren auf die Größe eines Microservice (siehe [Abb. 4-1](#)):

- Die Teamgröße stellt eine obere Grenze dar: Ein Microservice darf auf keinen Fall so groß sein, dass mehrere Teams an ihm arbeiten müssen. Schließlich sollen die Teams

unabhängig voneinander entwickeln und Software in Produktion bringen. Das ist nur möglich, wenn jedes Team an einer eigenen Deployment-Einheit arbeitet – also einem eigenen Microservice. Ein Team kann aber auch an mehreren Microservices arbeiten.

- Die Modularisierung schränkt die Größe eines Microservice weiter ein: Er sollte möglichst so groß sein, dass ein Entwickler ihn verstehen und weiterentwickeln kann. Noch kleiner ist natürlich besser. Diese Grenze liegt unterhalb der Teamgröße: Was ein Entwickler noch verstehen kann, sollte ein Team auch noch weiterentwickeln können.
- Die Ersetzbarkeit nimmt mit der Größe des Microservice ab. Daher kann sie die obere Grenze für die Größe eines Microservice beeinflussen. Diese Grenze liegt unterhalb der Modularisierung: Wenn jemand den Microservice ersetzen kann, muss er ihn auch verstehen können.
- Eine untere Grenze ist die Infrastruktur: Wenn es zu aufwendig ist, für einen Microservice Infrastruktur bereitzustellen, sollte die Anzahl der Microservices eher kleiner sein – und damit ergibt sich, dass die Microservices eher groß sind.
- Ebenso nimmt die verteilte Kommunikation mit der Anzahl der Microservices zu. Auch aus diesem Grund sollte die Größe des Microservice nicht zu klein gewählt werden.
- Die Konsistenz der Daten und Transaktionen kann nur in einem Microservice sichergestellt werden. Daher dürfen die Microservices nicht so klein sein, dass Konsistenz und Transaktionen mehrere Microservices umfassen.

Abb. 4–1 Einflussfaktoren für die Größe eines Microservice



Diese Faktoren beeinflussen nicht nur die Größe der Microservices, sondern sie entsprechen auch einem bestimmten Verständnis von Microservices. Hauptvorteile der

Microservices sind demnach unabhängige Deployments und das unabhängige Arbeiten der verschiedenen Teams. Dazu kommt die Ersetzbarkeit der Microservices. Daraus lassen sich Grenzen für die Größe eines Microservice ableiten.

Aber es gibt auch andere Gründe für Microservices. Wenn Microservices zum Beispiel wegen der unabhängigen Skalierung eingeführt werden, muss die Größe so gewählt werden, dass jeder Microservice eine Einheit ist, die unabhängig skalieren muss.

Wie klein oder groß ein Microservice sein kann, ist aus dieser Aufstellung alleine nicht zu ermitteln. Es hängt auch von der Technologie ab. Insbesondere der Aufwand für die Infrastruktur eines Microservice und für die verteilte Kommunikation hängt von der verwendeten Technologie ab. [Kapitel 15](#) betrachtet Technologien, mit denen sehr kleine Services möglich werden – sogenannte Nanoservices. Sie haben andere Vor- und Nachteile als Microservices, die beispielsweise mit den Technologien aus [Kapitel 14](#) umgesetzt werden.

Es gibt also keine ideale Größe, sondern die Größe hängt von der Technologie und dem Einsatzkontext der Microservices ab.

Selber ausprobieren und experimentieren

- Wie ist der Aufwand für das Deployment eines Microservice in Deiner Sprache, Plattform und Infrastruktur?
 - Ist es nur ein einfacher Prozess? Oder eine komplexe Infrastruktur mit Application Server oder anderen Infrastrukturelementen?
 - Wie kann der Aufwand für das Deployment gesenkt werden, sodass kleinere Microservices möglich werden?

Auf Basis dieser Informationen kannst du eine untere Grenze für die Größe eines Microservice definieren. Obere Grenzen sind durch die Teamgröße und die Modularisierung gegeben – auch hier solltest du dir entsprechende Grenzen überlegen.

4.2 Das Gesetz von Conway

Das Gesetz von Conway stammt von dem amerikanischen Informatiker Melvin Edward Conway [\[6\]](#)[\[7\]](#) und besagt:

Organisationen, die Systeme designen, können nur solche Designs entwerfen, welche die Kommunikationsstrukturen dieser Organisationen abbilden.

Wichtig ist, dass es nicht nur um Software geht, sondern um jegliche Art von Design. Die Kommunikationsstrukturen, von denen Conway spricht, müssen nicht mit dem Organigramm übereinstimmen. Oft gibt es informelle Kommunikationsstrukturen, die ebenfalls in diesem Kontext betrachtet werden können. Ebenso kann die geografische Verteilung der Teams die Kommunikation beeinflussen. Schließlich ist es einfacher, mit einem Kollegen im selben Raum oder zumindest am selben Standort zu sprechen, als wenn der Kollege in einer anderen Stadt oder gar einer anderen Zeitzone arbeitet.

Der Grund für das Gesetz von Conway liegt darin, dass jede organisatorische Einheit einen bestimmten Teil der Architektur entwirft. Sollen zwei Teile der Architektur eine Schnittstelle haben, ist eine Abstimmung über diese Schnittstelle notwendig – und damit eine Kommunikationsbeziehung zwischen den organisatorischen Einheiten, die für die jeweiligen Teile zuständig sind.

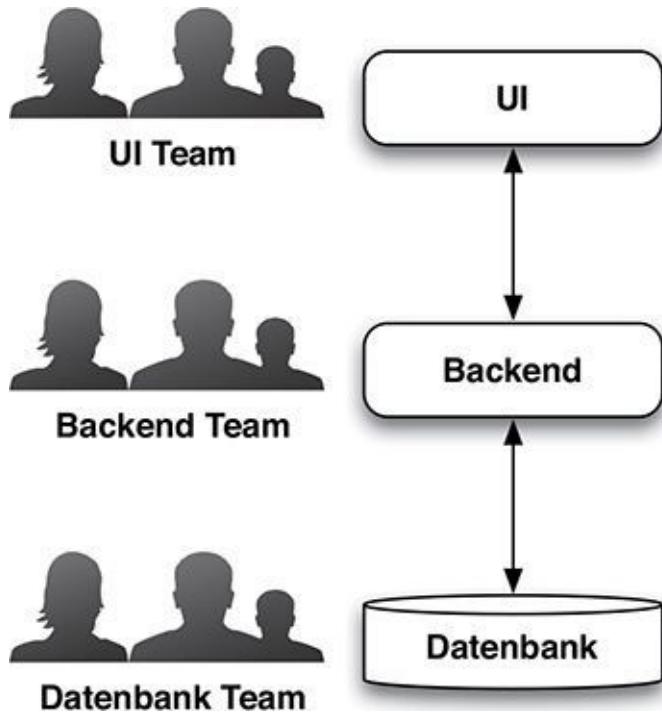
Aus dem Gesetz lässt sich auch ableiten, dass eine Modularisierung des Designs sinnvoll ist. Durch ein solches Design wird erreicht, dass sich nicht jeder im gesamten Team mit jedem anderen absprechen muss. Stattdessen können die Personen, die am selben Modul arbeiten, sich eng abstimmen, während Personen, die an unterschiedlichen Modulen arbeiten, sich nur abstimmen müssen, wenn es eine Schnittstelle gibt – und dann auch nur über die Gestaltung der Schnittstelle.

Aber die Kommunikationsbeziehungen gehen weiter. Es ist einfacher, mit einem Team im selben Gebäude zu kollaborieren als einem Team in einer anderen Stadt, einem anderen Land oder gar einer anderen Zeitzone. Daher können Teile der Architektur, die viele Kommunikationsbeziehungen haben, besser von Teams in räumlicher Nähe umgesetzt werden, weil sie leichter miteinander kommunizieren können. Es geht eben beim Gesetz von Conway nicht um das Organigramm, sondern um die realen Kommunikationsbeziehungen.

Conway stellt übrigens auch die These auf, dass eine große Organisation viele Kommunikationsbeziehungen hat. Dadurch wird die Kommunikation schwieriger oder sogar ganz unmöglich. In der Folge kann auch die Architektur immer mehr in Mitleidenschaft gezogen werden und schließlich zusammenbrechen. Zu viele Kommunikationsbeziehungen sind also letztendlich für ein Projekt ein echtes Risiko.

Normalerweise wird das Gesetz von Conway gerade in der Software-Entwicklung als eine Einschränkung angesehen. Nehmen wir an, ein Projekt wird nach technischen Aspekten aufgeteilt ([Abb. 4–2](#)). Alle Entwickler mit einer UI-Ausrichtung kommen in ein Team, die Entwickler mit Backend-Ausrichtung in ein zweites und das dritte Team sind die Datenbankexperten. Die Aufteilung hat den Vorteil, dass die drei Teams jeweils aus den Experten für die entsprechende Technologie bestehen. Dadurch ist die Organisationsform sehr einfach und klar umsetzbar. Außerdem erscheint diese Aufteilung auch logisch. Teammitglieder können sich sehr einfach unterstützen und der technische Austausch ist auch einfacher.

Abb. 4–2 Technische Aufteilung des Projekts



Aus dieser Aufteilung folgt nach dem Gesetz von Conway, dass die Teams drei technische Schichten implementieren werden: eine UI, ein Backend und eine Datenbank. Diese Aufteilung entspricht der Organisation, die auch durchaus sinnvoll aufgebaut ist. Aber sie hat einen entscheidenden Nachteil: Ein typisches Feature benötigt Änderungen an der UI, am Backend und an der Datenbank. Die UI muss die neuen Features für Nutzer verwendbar machen, das Backend muss die Logik umsetzen und die Datenbank muss Strukturen für das Speichern der entsprechenden Daten schaffen. Das hat folgende Nachteile:

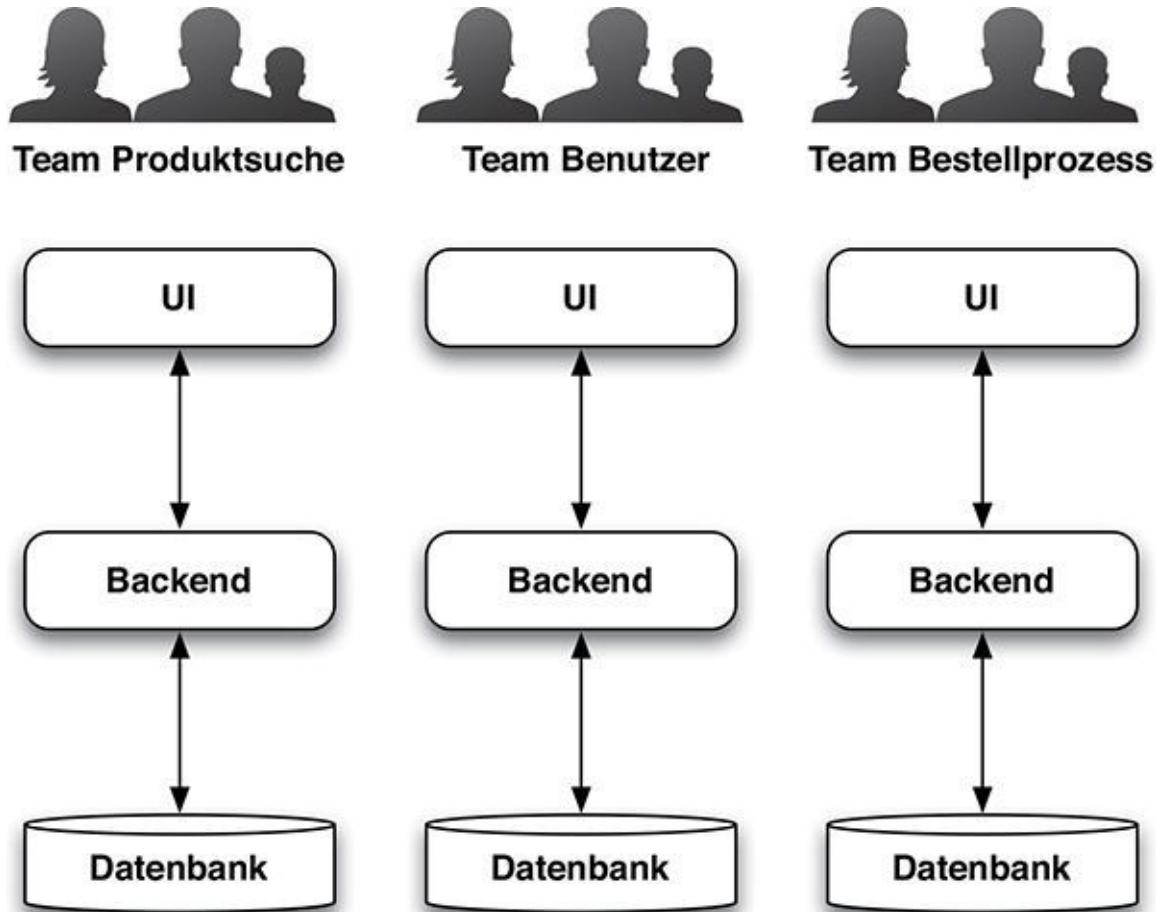
- Derjenige, der das Feature umgesetzt haben will, muss mit drei Teams sprechen.
- Die Teams müssen sich untereinander koordinieren und neue Schnittstellen schaffen.
- Die Arbeiten der Teams müssen so koordiniert sein, dass sie zeitlich zueinander passen. Das Backend kann beispielsweise ohne Zulieferung der Datenbank kaum sinnvoll arbeiten – und die UI auch nicht ohne Zulieferungen des Backends.
- Wenn die Teams in Sprints arbeiten, führen die Abhängigkeiten zu zeitlichen Verzögerungen: Das Datenbankteam erstellt im ersten Sprint die nötigen Änderungen, im zweiten Sprint implementiert das Backend-Team die Logik und im dritten Sprint kommt die UI. So dauert die Implementierung eines Features drei Sprints.

Letztendlich führt dieser Ansatz zu einer großen Menge von Abhängigkeiten, Kommunikation und Koordination. Daher ist diese Organisation nicht sehr sinnvoll, wenn es darum geht, möglichst schnell neue Features umzusetzen.

Vielen Teams, die einen solchen Ansatz nutzen, ist die Auswirkung auf die Architektur nicht klar und sie beachten diesen Aspekt nicht weiter. Im Mittelpunkt einer solchen Organisation steht eher der Aspekt, dass Mitarbeiter mit ähnlichen Skills in der Organisation an einer ähnlichen Stelle positioniert sein sollen. So wird die Organisation zu einem Hemmnis für fachlich geschnittene Module wie Microservices, dem die Aufteilung der Teams in technische Schichten entgegensteht.

Das Gesetz von Conway kann aber auch zur Unterstützung von Ansätzen wie Microservices genutzt werden. Wenn das Ziel ist, einzelne Komponenten möglichst unabhängig voneinander zu entwickeln, kann das System in fachliche Komponenten aufgeteilt werden. Anhand dieser fachlichen Komponenten können jeweils Teams gebildet werden. Abbildung 4–3 zeigt das: Es gibt jeweils ein Team für Produktsuche, Benutzer und Bestellprozess. Sie arbeiten an den jeweiligen Komponenten, die technisch in UI, Backend und Datenbank aufgeteilt sein können. Übrigens sind die fachlichen Komponenten in der Abbildung gar nicht extra benannt, weil sie identisch mit der Bezeichnung der Teams sind. Komponenten und Teams sind synonym. Dieser Ansatz entspricht der Idee von sogenannten cross functional Teams, wie Methoden wie Scrum sie fordern. Diese Teams sollen verschiedene Rollen umfassen, sodass sie ein möglichst breites Aufgabenspektrum abdecken. Nur ein solches Team kann eine Komponente verantworten – von den Anforderungen über die Implementierung bis hin zum Betrieb.

Abb. 4–3 Aufteilung nach Fachlichkeiten



Die Aufteilung in die technischen Artefakte und die Schnittstelle zwischen den Artefakten können nun innerhalb des Teams geklärt werden. Im einfachsten Fall muss dazu nur ein Entwickler mit dem Entwickler sprechen, der neben ihm am Tisch sitzt. Zwischen den Teams ist die Abstimmung komplizierter. Die ist aber auch nicht so oft notwendig, weil Fachlichkeiten idealerweise nur von einem einzigen Team umgesetzt werden müssen. Außerdem entstehen durch diesen Ansatz dünne Schnittstellen zwischen den Fachlichkeiten, weil für die Definition der Schnittstelle eine aufwendige Koordination über Teamgrenzen hinweg notwendig ist.

Letztendlich lautet die zentrale These aus dem Gesetz von Conway, dass Architektur und Organisation nur zwei Seiten derselben Medaille sind. Wenn das geschickt genutzt wird, hat das System eine recht saubere und für das Projekt nützliche Architektur. Gemeinsames Ziel von Architektur und Organisation ist die reibungslose Arbeit der Teams mit möglichst wenig Koordination.

Die saubere Aufteilung der Fachlichkeiten in die Komponenten erleichtert auch die Wartung. Und da für jede Fachlichkeit und jede Komponente nur ein Team zuständig ist, wird diese Aufteilung auch langfristig stabil und das System so auch langfristig wartbar bleiben.

Die Teams benötigen Anforderungen, an denen sie arbeiten können. Das bedeutet, dass die Teams fachliche Ansprechpartner haben müssen, die solche Anforderungen stellen können. Das hat Auswirkungen auf die Organisation über das Projekt hinaus, denn die Anforderungen kommen aus den Fachbereichen und auch diese müssen entsprechend dem Gesetz von Conway den Teamstrukturen im Projekt und der fachlichen Architektur entsprechen. Das Gesetz kann über die Software-Entwicklung hinaus auf die Kommunikationsstrukturen der gesamten Organisation einschließlich der Anwender ausgeweitet werden. Oder umgekehrt können sich die Teamstruktur eines Projekts und damit die Architektur eines Microservice-Systems aus der Organisation der Fachbereiche ergeben.

Die bisherige Betrachtung hat nur allgemein eine Beziehung zwischen der Architektur und der Organisation des Projekts gezeigt. Es wäre ohne Weiteres denkbar, die Architektur an Fachlichkeiten auszurichten und jeweils einem Team die Verantwortung für eine Fachlichkeit zu übergeben, ohne dabei Microservices zu nutzen. Dann würde das Projekt einen Deployment-Monolithen entwickeln, in dem alle Funktionalitäten umgesetzt werden. Microservices unterstützen den Ansatz aber. [Abschnitt 3.1](#) hat schon gezeigt, dass Microservices eine technische Unabhängigkeit bieten. Zusammen mit der fachlichen Aufteilung werden die Teams noch unabhängiger voneinander und müssen sich noch weniger koordinieren. Sowohl die technische als auch die fachliche Koordination der Teams wird auf das Notwendigste reduziert. Dadurch wird es deutlich einfacher, an vielen Features parallel zu arbeiten und die Features auch in Produktion zu bringen.

Microservices sind als technische Umsetzung einer Architektur besonders gut geeignet, den Ansatz einer fachlichen Aufteilung mithilfe des Gesetzes von Conway zu unterstützen. Tatsächlich ist genau dieser Aspekt eine wesentliche Eigenschaft einer Microservices-Architektur.

Allerdings bedeutet die Ausrichtung der Architektur an den Kommunikationsstrukturen, dass eine Änderung des einen auch eine Änderung des anderen bedingt. Dadurch werden Änderungen der Architektur zwischen den Microservices aufwendiger und der Prozess insgesamt weniger flexibel. Wenn eine Funktionalität von einem Microservice in einen anderen verschoben wird, kann das auch zur Folge haben, dass die Funktionalität von einem anderen Team weiter gepflegt wird. Solche organisatorischen Änderungen machen Änderungen an der Software komplizierter.

Nun muss geklärt werden, wie diese fachliche Aufteilung sinnvoll umgesetzt werden

kann. Dazu hilft Domain-Driven Design (DDD).

Selber ausprobieren und experimentieren

Betrachte ein dir bekanntes Projekt:

- Wie sieht die Teamstruktur aus?
 - Ist sie technisch oder fachlich getrieben?
 - Muss die Struktur für einen Microservices-Ansatz geändert werden?
 - Wie muss sie geändert werden?
- Ist die Architektur auf Teams sinnvoll verteilbar? Schließlich soll jedes Team unabhängige fachliche Komponenten verantworten und dort Features umsetzen können.
 - Welche Änderungen an der Architektur wären notwendig?
 - Wie aufwendig wären die Änderungen?

4.3 Domain-Driven Design und Bounded Context

In seinem gleichnamigen Buch hat Eric Evans [5] Domain-Driven Design (DDD) als eine Pattern-Sprache formuliert. Es ist eine Sammlung von zusammenhängenden Entwurfsmustern und soll die Entwicklung von Software vor allem in komplexen Domänen unterstützen. Die Namen der Entwurfsmuster sind in KAPITÄLCHEN gesetzt.

Domain-Driven Design ist für ein Verständnis von Microservices wichtig, weil es bei der Strukturierung von größeren Systemen nach Fachlichkeiten hilft. Genau so ein Modell benötigen wir für den Schnitt eines Systems in Microservices. Jeder Microservice soll eine eigene fachliche Einheit bilden, die so geschnitten ist, dass für Änderungen oder neue Features nur ein Microservice geändert werden muss. Dann ziehen wir aus der unabhängigen Entwicklung in den Teams den maximalen Nutzen, weil mehrere Features parallel ohne größere Koordinierung umgesetzt werden können.

Als Basis definiert DDD, wie ein Modell für eine *Ubiquitous Language* Domäne entworfen werden kann. Eine wesentliche Grundlage von DDD ist UBIQUITOUS LANGUAGE (allgemeingültige Sprache). Die Idee ist, dass die Software genau dieselben Begriffe nutzen soll wie die Domänenexperten. Das gilt auf allen Ebenen: sowohl für den Code und die Variablennamen wie auch für die Datenbankschemata. Dadurch wird sichergestellt, dass die Software tatsächlich die wesentlichen Elemente der Domäne erfasst und umsetzt. Nehmen wir beispielsweise an, dass es in einem E-Commerce-System Express-Bestellungen gibt. Eine Möglichkeit damit umzugehen wäre, in der Order-Tabelle einen booleschen Wert mit dem Namen »fast« (Englisch: schnell) anzulegen. Das führt zu folgendem Problem: Domänenexperten müssen den Begriff »Express-Bestellung«, mit dem sie jeden Tag umgehen, in Order übersetzen – genau genommen sogar in Order mit einem bestimmten booleschen Wert. Was boolesche Werte sind, ist ihnen möglicherweise nicht klar. Eine Diskussion des

Modells wird dadurch schwierig, weil andauernd Begriffe abgeglichen und erklärt werden müssen. Besser wäre es, wenn in dem Datenbankschema direkt eine Tabelle namens Express-Bestellung existiert. Dann ist vollkommen klar, wie die Begriffe aus der Domäne in dem System umgesetzt sind.

Die Übersetzung von Fachbegriffen vom Deutschen ins Englische ist übrigens auch ein Verstoß gegen UBIQUITOUS LANGUAGE. Eine solche Übersetzung führt nämlich eine andere Sprache mit anderen Begriffen ein, in die beispielsweise ein Gespräch mit einem Nutzer übersetzt werden muss. Da die Übersetzungen meistens nicht eindeutig sind, kann dieser Schritt zu erheblichen Verwirrungen führen.

Um ein Domänenmodell zu entwerfen, identifiziert Building Blocks DDD grundlegende Muster:

- ENTITY (Entität) ist ein Objekt mit einer eigenen Identität. In einer E-Commerce-Anwendung könnten das der Kunde oder die Ware sein. ENTITIES werden typischerweise in einer Datenbank gespeichert. Das ist aber nur eine technische Umsetzung des Konzepts ENTITY. Eine ENTITY gehört eigentlich zur fachlichen Modellierung der Domäne wie die anderen DDD-Konzepte auch.
- VALUE OBJECTS (Wertobjekte) haben keine eigene Identität. Ein Beispiel kann eine Adresse sein, die nur im Kontext mit einem Kunden sinnvoll ist und daher keine eigene Identität hat.
- AGGREGATES (Aggregate) sind zusammengesetzte Domänenobjekte. Sie ermöglichen einen einfacheren Umgang mit Invarianten und anderen Bedingungen. Beispielsweise kann eine Bestellung ein AGGREGATE aus Bestellzeilen sein. So kann gewährleistet werden, dass eine Bestellung für Neukunden einen bestimmten Betrag nicht überschreitet. Das ist eine Bedingung, die durch eine Berechnung von Werten aus den Bestellzeilen erfüllt werden muss, sodass die Bestellung als AGGREGATE diese Bedingungen kontrollieren kann.
- SERVICES enthalten Geschäftslogik. DDD fokussiert auf die Modellierung von Geschäftslogik an ENTITIES, VALUE OBJECTS und AGGREGATES. Aber Logik, die auf mehrere dieser Objekte zugreift, kann nicht sinnvoll an diesen Objekten modelliert werden. Für diese Fälle gibt es SERVICES. Der Bestellvorgang könnte ein solcher Service sein, da er Zugriff auf die Waren, den Kunden und die ENTITY Bestellung benötigt.
- REPOSITORIES dienen dazu, auf die Gesamtheit aller ENTITIES eines Typs zuzugreifen. Typischerweise verbirgt sich dahinter eine Persistenz beispielsweise in einer Datenbank.
- FACTORIES (Fabriken) sind vor allem nützlich, um komplexe Fachobjekte zusammenzubauen. Das ist vor allem der Fall, wenn sie beispielsweise viele Assoziationen enthalten.

AGGREGATES kommt im Zusammenhang mit Microservices eine besondere Bedeutung zu: In einem AGGREGATE können Konsistenzregeln erzwungen werden. Wegen der notwendigen Konsistenz müssen parallele Änderungen an einem AGGREGATE koordiniert werden. Sonst können zwei parallele Änderungen die Konsistenz gefährden. Wenn in eine

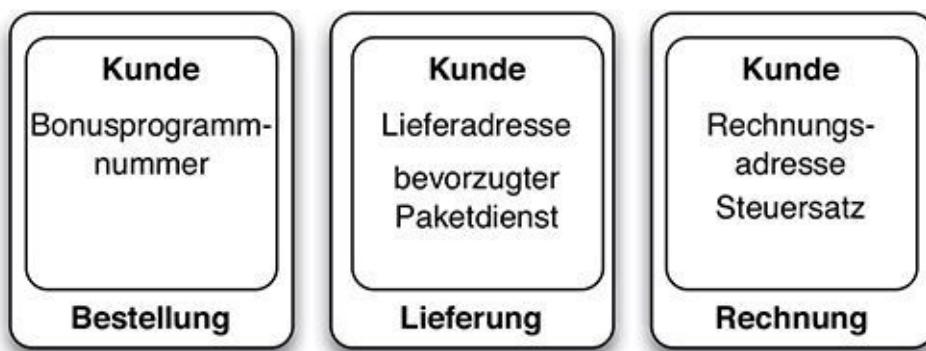
Bestellung zwei Bestellpositionen parallel aufgenommen werden, kann dadurch die Konsistenz gefährdet werden. Die Bestellung hat bereits einen Wert von 900 € und darf maximal 1000 € betragen. Wenn nun parallel zwei Bestellpositionen zu je 60 € hinzugefügt werden, kann es sein, dass beide ausgehend von den 900 € einen Gesamtstand von 960€ berechnen, der noch akzeptabel wäre. Daher müssen die Änderungen serialisiert werden, sodass das endgültige Ergebnis von 1020 € kontrolliert wird. Änderungen an AGGREGATES müssen daher serialisiert werden. Aus diesem Grund kann ein AGGREGATE nicht unter zwei Microservices aufgeteilt werden. Dann kann Konsistenz nicht mehr garantiert werden. AGGREGATES können also nicht zwischen Microservices aufgeteilt werden.

Die Building Blocks wie AGGREGATE stellen für viele den Kern von DDD dar. DDD beschreibt außerdem mit Strategic Design, wie verschiedene Domänenmodelle miteinander interagieren und so komplexere Systeme aufgebaut werden können. Dieser Aspekt von DDD ist vermutlich noch wichtiger als die Building Blocks. Auf jeden Fall ist es der Bereich von DDD, der Einfluss auf Microservices hat.

Kern des Strategic Designs ist der BOUNDED CONTEXT (Kontextgrenzen). Grundüberlegung ist, dass jedes Domänenmodell nur in bestimmten Grenzen innerhalb eines Systems sinnvoll ist. Im E-Commerce sind beispielsweise von einer Bestellung im Kontext der Lieferung die Anzahl, die Größe und die Gewichte der bestellten Waren interessant, da sich danach Lieferwege und Lieferkosten entscheiden. Für die Buchhaltung hingegen sind Preise und Steuersätze relevant. Ein komplexes System besteht aus mehreren BOUNDED CONTEXTS. Das ähnelt dem Aufbau eines komplexen biologischen Organismus aus Zellen, die auch jeweils abgeschlossen sind und ein eigenes Innenleben haben.

BOUNDED CONTEXT: Ein Beispiel

Abb. 4–4 Beispiel für BOUNDED CONTEXTS



Als Beispiel für einen BOUNDED CONTEXT soll der Kunde aus einem E-Commerce-System dienen. (Abb. 4–4) Die unterschiedlichen BOUNDED CONTEXTS sind Bestellung, Lieferung und Rechnung. Die Komponente Bestellung ist für die Abarbeitung des Bestellprozesses zuständig. Lieferung implementiert den Lieferprozess. Und Rechnung ist schließlich für das Erstellen der Rechnung zuständig.

Jeder dieser BOUNDED CONTEXTS ist an bestimmten Daten des Kunden interessiert:

- Bei der Bestellung sollen dem Kunden in einem externen Bonusprogramm Punkte gutgeschrieben werden. In diesem BOUNDED CONTEXT muss die Nummer des Kunden bei dem Bonusprogramm bekannt sein.
- Bei der Lieferung sind die Lieferadresse und der bevorzugte Paketdienst des Kunden relevant.
- Und für die Erstellung der Rechnung müssen die Rechnungsadresse und der Steuersatz für diesen Kunden bekannt sein.

So hat jeder BOUNDED CONTEXT ein eigenes Modell des Kunden. Das ermöglicht eine unabhängige Änderbarkeit der Microservices. Wenn beispielsweise für Rechnungen mehr Informationen über Kunden notwendig sind, so erfordert das nur Änderungen in dem BOUNDED CONTEXT Rechnung.

Es kann sinnvoll sein, Basisinformationen über den Kunden in einem getrennten BOUNDED CONTEXT zu speichern. Solche grundlegenden Daten sind wahrscheinlich in vielen BOUNDED CONTEXTS sinnvoll. Dazu können die BOUNDED CONTEXTS miteinander kooperieren (siehe unten).

Ein universelles Modell eines Kunden ist allerdings kaum sinnvoll. Es wäre sehr komplex, weil es alle Informationen aus alle BOUNDED CONTEXTS enthalten muss. Außerdem würde jede Änderung an den Informationen eines Kunden, die in einem bestimmten Kontext notwendig sind, dieses universelle Modell betreffen. Das würde solche Änderungen verkomplizieren und vermutlich dazu führen, dass das Modell ständig geändert wird.

Um den Aufbau des Systems in den verschiedenen BOUNDED CONTEXTS zu verdeutlichen, kann eine CONTEXT MAP (Kontextübersicht) genutzt werden (siehe [Abschnitt 8.2](#)). Jeder der BOUNDED CONTEXTS kann dann in einem oder mehreren Microservices implementiert sein.

Nun ist die Frage, wie die einzelnen BOUNDED CONTEXTS zusammenhängen. Es gibt verschiedene Möglichkeiten:

Kollaboration zwischen BOUNDED CONTEXTS

- Bei einem SHARED KERNEL (geteilter Kern) haben die Domänenmodelle einige gemeinsame Elemente, aber in anderen Bereichen unterscheiden sie sich.
- CUSTOMER/SUPPLIER (Kunde/Lieferant) bedeutet, dass ein Untersystem für den Aufrufer ein Domänenmodell anbietet. Der Aufrufer ist dann der Kunde und bestimmt den genauen Aufbau des Modells.
- Ganz anders bei CONFORMIST (Konformist): Der Aufrufer nutzt dasselbe Modell wie das Untersystem und lässt sich dadurch das andere Modell aufzwingen. Dieses Vorgehen ist recht einfach – es ist keine Übersetzung notwendig. Ein Beispiel kann eine Standard-Software für eine bestimmte Domäne sein. Die Entwickler dieser Software wissen wahrscheinlich sehr viel über die Domäne, weil sie viele verschiedene Einsatzszenarien gesehen haben. Der Aufrufer kann dieses Modell verwenden, um das Wissen aus der Modellierung nutzen zu können.
- Das ANTICORRUPTION LAYER (Antikorruptionsschicht) übersetzt ein Domänenmodell

in ein anderes, sodass beide vollständig voneinander entkoppelt sind. Dadurch können Legacy-Systeme integriert werden, ohne dass die Domänenmodelle übernommen werden müssen. Oft ist gerade die Modellierung der Daten in Legacy-Systemen wenig sinnvoll.

- SEPARATE WAYS (getrennte Wege) bedeutet, dass die beiden Systeme nicht integriert werden und unabhängig voneinander bleiben.
- Bei OPEN HOST SERVICE bietet der BOUNDED CONTEXT spezielle SERVICES an, die jeder nutzen kann. So kann jeder eine eigene Integration zusammenstellen. Das ist vor allem nützlich, wenn eine Integration mit sehr vielen anderen Systemen notwendig ist und die Implementierung dieser Integrationen zu aufwendig wäre.
- Ähnliches leistet die PUBLISHED LANGUAGE (veröffentlichte Sprache). Sie bietet eine bestimmte Modellierung der Domäne als gemeinsame Sprache zwischen den BOUNDED CONTEXTS an. Da sie breit genutzt wird, kann diese Sprache dann praktisch nicht mehr geändert werden.

Warum Sie ein kanonisches Datenmodell lieber vermeiden sollten

von Stefan Tilkov, innoQ GmbH

Wenn Sie selbst als Enterprise- oder IT-Unternehmensarchitekt tätig sind, wissen Sie, wovon ich spreche – für alle anderen hier eine kurze Einführung in diese Welt: Sie haben jede Menge Meetings, außerdem Meetings und noch viel mehr Meetings. Sie erstellen und betrachten hauptsächlich Foliensätze. Und zwar nicht die Variante, die Sie bei Presentation Zen oder einem TED-Talk sehen, sondern eher die, bei der auf jeder Folie der Text eines mittleren Romans enthalten ist. Es gibt konzeptuelle Architektur-Frameworks und Metaframeworks wie Zachman oder TOGAF, Referenzarchitekturen, unternehmensweite Klassifizierungs- und Schichtenansätze, SOA-, EAI- und ESB-Produkte und – vor allem in letzter Zeit – eine Menge Diskussion und natürlich auch wieder die unvermeidlichen Produkte rund um das Thema Web-APIs. Produktanbieter, Systemintegratoren und natürlich Berater sehen ihre Chance, sich langfristig selbst als Teil Ihrer Unternehmensstrategie darin eine feste Position zu sichern. Eine Aufgabe in diesem Umfeld kann manchmal sehr frustrierend sein. Aber hier und vielleicht auch nur hier haben Sie durchaus gelegentlich die Chance, wirklich zentrale Entscheidungen mit großen Auswirkungen zu treffen. Anders formuliert: Die Räder, die man hier drehen kann, sind wirklich groß und manchmal fast nicht zu bewegen. Aber wenn es Ihnen gelingt, ist der Effekt durchaus dramatisch.

Aber worum es hier gehen soll, ist etwas anderes: Es ist manchmal verblüffend, wie viele der Probleme, die wir in großen Individualsystemen sehen, sich auf der Unternehmens-IT-Ebene ebenfalls entdecken lassen – frei nach dem Motto: Wir machen selten Fehler, aber wenn, dann richtig große. Mein Lieblingsbeispiel dafür ist die Idee eines kanonischen Datenmodells (Canonical Data Model, CDM) für Schnittstellen.

Kurz gesagt verbirgt sich dahinter die Idee, dass Sie unabhängig von der eingesetzten Technologie – einem ESB, einer BPM-Plattform oder auch nur einfach einer Menge von Services – die Datenmodelle der Geschäftsobjekte standardisieren,

die Systeme an ihren Schnittstellen austauschen. In der Extremform, die alles andere als selten ist, gibt es dann genau eine Art von Person, Kunde, Bestellung usw. mit den identifizierenden und beschreibenden Attributen und Assoziationen, auf die man sich unternehmensweit geeinigt hat. Der Reiz dieses Ansatzes ist nicht schwer zu erkennen: Selbst ein nichttechnischer Manager versteht, dass die immer wiederkehrende Abbildung vom eigenen auf ein fremdes Datenmodell jedes Mal, wenn zwei Systeme miteinander kommunizieren wollen, Zeitverschwendug ist. Ganz offensichtlich ist die Standardisierung also eine gute Idee: Jeder muss nur noch einmal zwei Konversionen implementieren – vom internen ins kanonische Datenmodell und umgekehrt. Neu zu entwickelnde Systeme verwenden einfach direkt das CDM. Alle können problemlos miteinander kommunizieren und die Welt ist in Ordnung!

Tatsächlich ist das Schnittstellen-CDM aber eine ganz furchtbare Idee. Wenn Sie eine Chance haben, sollten Sie sie unbedingt vermeiden.

In seinem Buch über Domain-Driven Design hat Eric Evans einem Konzept einen Namen gegeben, das eigentlich jeder kennt, der schon einmal erfolgreich ein größeres System entwickelt hat: den Bounded Context (in etwa: »Abgegrenzter Kontext«). Konzeptionell dient ein solcher Kontext dazu, ein großes Modell in kleinere, einzelne Modelle aufzuteilen. Der Grund dafür ist, dass ein einzelnes Riesenmodell erstens sehr schnell unwartbar wird, aber zweitens – was noch viel schwerer wiegt – eigentlich keinen fachlichen Sinn ergibt. Mit dem Einsatz von Bounded Contexts akzeptieren Sie, dass Dinge, die ähnlich aussehen und den gleichen Namen tragen, in verschiedenen Kontexten unterschiedliche Bedeutung haben. Das ist kein »Problem« der Implementierung, sondern eine harte Realität des Lebens. Die Abteilung, die sich in Ihrem Unternehmen mit der Abwicklung von Bestellungen beschäftigt, hat ein anderes Bild vom Kunden als die Abteilung, die die eigentliche Leistung erbringt – und das ist auch gut so.

Wenn dies schon für große Systeme gilt (und glauben Sie mir, das tut es), dann gilt es für eine unternehmensweite Architektur noch viel mehr. Natürlich kann man argumentieren, dass man mit einem Schnittstellen-CDM nur die externe Schnittstelle und nicht das interne Datenmodell standardisiert, aber das ändert nichts daran: Sie versuchen damit, alle dazu zu bewegen, sich auf ein »richtiges« Modell zu einigen, anstatt zu akzeptieren, dass es gut ist, dass jedes System seine eigenen Bedürfnisse hat.

Aber ist das nicht alles nur graue Theorie? Warum sollte Sie das interessieren? Nach meiner Erfahrung sind insbesondere große Unternehmen in außerordentlichem Maße in der Lage, auf Basis falscher Grundannahmen beeindruckende Mengen von Aufgaben zu erzeugen, deren Erledigung keinerlei Mehrwert erzeugt, sondern sogar schadet. Das CDM – in der Ausprägung, die ich hier beschrieben habe – erfordert eine Koordination zwischen allen Parteien, die ein bestimmtes Objekt in ihren Schnittstellen verwenden. Es sie denn, Sie glauben daran, dass irgendjemand das Richtige im stillen Kämmerlein entwerfen könnte. Das gilt sogar dann, wenn diese Systeme gar nicht direkt miteinander kommunizieren! Allein die Tatsache, dass alle Systeme Nutzer des CDM sind, macht dieses zu einem Flaschenhals: Sie sitzen in Meetings mit anderen Unternehmensarchitekten und versuchen sich darauf zu einigen, welche Attribute zum Unternehmensgeschäftsobjekt »Kunde« gehören und welche nicht. Natürlich endet das

Ganze mit einem Entwurf, der Hunderte optionaler Attribute und Beziehungen hat, weil jeder darauf besteht, die eigenen Bedürfnisse dort erfüllt zu sehen. Diverse Entscheidungen werden allein deswegen sehr seltsam ausfallen, weil die internen Restriktionen der Systeme sich im Schnittstellenentwurf wiederfinden werden. Obwohl es schmerzhaften Aufwand kosten wird, sich endlich zu einigen, wird das Ergebnis Zombie-Qualitäten haben und von jedem gehasst werden, der damit arbeiten muss.

Ist ein CDM deswegen eine grundsätzlich schlechte Idee? Ja, wenn Sie es nicht grundsätzlich anders angehen. In vielen Fällen habe ich starke Zweifel, dass es sich überhaupt lohnt und Sie mit einer anderen und weniger ehrgeizigen Lösung besser bedient sind. Aber wenn Sie sich für ein CDM entschieden haben, können die folgenden Ansätze hilfreich sein, um einige der Probleme zu adressieren:

- Erlauben Sie, dass unabhängige Teile auch unabhängig voneinander spezifiziert werden können. Wenn nur ein System für einen bestimmten Teil des Modells verantwortlich ist, überlassen Sie es den dafür Verantwortlichen, die kanonische Variante zu entwerfen. Ersparen Sie ihnen die Teilnahme an Meetings! Wenn Sie sich nicht sicher sind, ob es eine Überlappung zwischen zwei Gruppen gibt, gibt es sie wahrscheinlich nicht.
- Standardisieren Sie Formate und evtl. noch Fragmente des Datenmodells. Versuchen Sie nicht, ein konsistentes Modell der Welt zu entwerfen. Setzen Sie stattdessen auf kleinere Bausteine, z. B. kleine XML- oder JSON-Fragmente, ähnlich Microformats aus der Webwelt (<http://microformats.org>), die eine Handvoll Geschäftsattribute bündeln und die ich noch nicht einmal als Geschäftsobjekt bezeichnen würde.
- Am wichtigsten: Verteilen Sie Ihr Modell nicht nach einem Push-Verfahren ausgehend von einem zentralen Team an die (un-)glücklichen Konsumenten, sondern sehen Sie das zentrale Team als Dienstleister, das die Modelle der anderen sammelt. Diese wiederum sollten die Modelle nach dem »Pull«-Prinzip in ihren eigenen Kontext importieren. Akzeptieren Sie, dass es trotz des grandiosen Titels nicht Sie sind, der die eigentliche Arbeit macht, sondern das Team, das ein konkretes fachliches System betreut. Betrachten Sie Ihr zentrales Modell als eine Art Suchindex, der seinen Wert dann beweist, wenn er von anderen freiwillig genutzt wird.

Da ich selbst schon häufiger in der Rolle eines Unternehmensarchitekten tätig war und dort beraten habe, habe ich gelernt, dass Ihre erste Maxime sein sollte, den anderen Beteiligten keine Steine in den Weg zu legen. Ein besonders wichtiger Weg, um das zu erreichen, ist die Vermeidung von Zentralisierung. Es muss Ihr Ziel sein, den richtigen minimalen Satz an Regeln so zu spezifizieren, dass Teams und Projekte so unabhängig wie möglich – voneinander und von einer zentralen Architekturtruppe – arbeiten können. Ein kanonisches Datenmodell, wie ich es beschrieben habe, erreicht das genaue Gegenteil.

Microservices sollen jeweils eine eigene fachliche Domäne modellieren, sodass neue Features oder Änderungen nur in einem Microservice umgesetzt werden müssen. Auf der Basis von

Bounded Context und Microservices

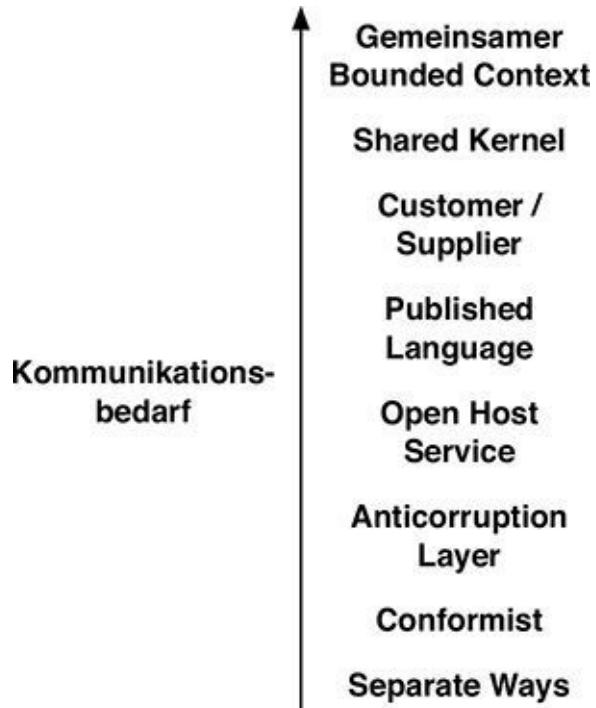
BOUNDED CONTEXT kann ein solches Modell entworfen werden.

Ein Team kann an einem oder mehreren BOUNDED CONTEXTS arbeiten, die jeweils Grundlage für einen oder mehrere Microservices sind. Änderungen und neue Features sollen typischerweise nur einen BOUNDED CONTEXT betreffen – und damit ein Team. So wird das Ziel erreicht, dass Teams weitgehend unabhängig voneinander arbeiten können. Ein BOUNDED CONTEXT kann in mehrere Microservices aufgeteilt werden, wenn das sinnvoll erscheint. Das kann technische Gründe haben. Beispielsweise muss vielleicht ein bestimmter Teil eines BOUNDED CONTEXT besonders hoch skaliert werden. Das ist einfacher, wenn dieser Teil in einen eigenen Microservice separiert wird. Es sollte aber vermieden werden, dass ein Microservice mehrere BOUNDED CONTEXTS enthält, weil dann gegebenenfalls mehrere neue Features in einem Microservice umgesetzt werden. Das widerspricht aber dem Ziel der unabhängigen Entwicklung von Features.

Es kann dennoch sein, dass eine spezielle Anforderung viele BOUNDED CONTEXTS umfasst – dann würde entsprechend zusätzliche Koordination und Kommunikation notwendig werden.

Die Koordination der Teams untereinander kann durch die unterschiedlichen Kollaborationsmöglichkeiten geregelt werden. Die beeinflussen ebenfalls die Unabhängigkeit der Teams: SEPARATE WAYS, ANTICORRUPTION LAYER oder OPEN HOST SERVICE bieten sehr viel Unabhängigkeit. CONFORMIST oder CUSTOMER/SUPPLIER hingegen binden die Domänenmodelle sehr eng aneinander und die Teams müssen sich dann eng absprechen, wenn ihre Microservices eine solche Kollaboration nutzen (siehe Abb. 4–5).

Abb. 4–5 Kommunikationsbedarf verschiedener Kollaborationsmöglichkeiten



Hier wird wie schon beim Gesetz von Conway aus Abschnitt 4.2 deutlich, dass Organisation und Architektur sehr eng zusammenhängen. Wenn die Architektur eine fachliche Aufteilung ermöglicht, bei der Features nur Änderungen an einem bestimmten Teil der Architektur benötigen, können diese Teile so auf die Teams aufgeteilt werden, dass sie weitgehend unabhängig voneinander arbeiten können. DDD und insbesondere

BOUNDED CONTEXT zeigen, wie eine solche Aufteilung aussehen kann – und wie die Teile dann miteinander zusammenarbeiten können und wie sie sich untereinander koordinieren müssen.

Mit Large-Scale Structure geht DDD auch darauf ein, Large-Scale Structure wie das Gesamtsystem aus den verschiedenen BOUNDED CONTEXTS beziehungsweise Microservices betrachtet werden kann.

- Eine SYSTEM METAPHOR (Systemmetapher) kann dazu dienen, die grundlegende Struktur des Gesamtsystems zu definieren. Beispielsweise kann sich ein E-Commerce-System an dem Einkaufsprozess orientieren: Zunächst sucht der Kunde Produkte, dann vergleicht er die Produkte, wählt eines aus und bestellt schließlich die Auswahl. Daraus können dann drei Microservices werden: Suche, Vergleich und Bestellung.
- RESPONSIBILITY LAYER (Verantwortlichkeitsschichten) teilt das System in Schichten mit unterschiedlichen Verantwortlichkeiten auf. Schichten dürfen die darunter liegenden Schichten aufrufen, aber nicht umgekehrt. Damit ist keine technische Aufteilung beispielsweise in Datenbank, UI und Logik gemeint. Fachliche Schichten könnten beispielsweise in einem E-Commerce-System der Katalog, der Bestellprozess und die Rechnungserstellung sein. Der Katalog darf den Bestellprozess aufrufen und der wiederum darf die Rechnungserstellung aufrufen. Aufrufe in die andere Richtung sind nicht erlaubt.
- EVOLVING ORDER (Entstehende Strukturen) schlägt vor, die übergeordnete Struktur nicht zu stark festzulegen. Sie soll sich aus den einzelnen Bestandteilen schrittweise ergeben.

Für Microservices können diese Ansätze ein Hinweis sein, wie die Architektur eines Systems organisiert werden kann, das aus verschiedenen Microservices besteht (siehe auch [Kap. 8](#)).

Selber ausprobieren und experimentieren

Betrachte ein dir bekanntes Projekt:

- Welche BOUNDED CONTEXTS kannst du identifizieren?
- Erstelle einen Überblick über die BOUNDED CONTEXTS in einer CONTEXT MAP. Siehe dazu auch [Abschnitt 8.2](#).
- Wie kooperieren die BOUNDED CONTEXTS miteinander? (ANTICORRUPTION LAYER, CUSTOMER/SUPPLIER etc.). Füge auch diese Information in die CONTEXT MAP ein.
 - Wären andere Mechanismen an bestimmten Stellen besser?
- Wie könnten die BOUNDED CONTEXTS sinnvoll auf Teams aufgeteilt werden, sodass Features möglichst von einem Team umgesetzt werden können?

4.4 Microservice: Mit UI?

Dieses Buch empfiehlt, Microservices mit einer UI auszustatten. Die UI sollte die Fachlichkeit des Microservice einem Benutzer anbieten. So können alle Änderungen für eine Funktionalität in einem Microservice umgesetzt werden – egal ob sie die UI, die Logik oder die Datenbank betreffen. Allerdings besteht unter Microservice-Experten keine Einigkeit darüber, ob die Integration der UI in einen Microservice zwingend ist. Schließlich sollen Microservices nicht zu groß sein. Und wenn Logik sowieso von mehreren Frontends aus genutzt werden soll, kann ein Microservice ohne UI mit reiner Logik sinnvoll sein. Und es ist auch möglich, die Logik und die UI in zwei Microservices zu implementieren, aber beide von einem Team implementieren zu lassen. So können Features ohne Koordinierung über Teams implementiert werden.

Der Fokus auf Microservices mit UI stellt statt einer technischen Aufteilung eine fachliche Aufteilung in den Mittelpunkt. Die fachliche Aufteilung ist vielen Architekten fremd, aber gerade bei Microservices sehr wichtig. Daher ist ein Entwurf, bei dem die Microservices die UI enthalten, zumindest als erster Ansatz hilfreich, um die Architektur auf Fachlichkeit zu fokussieren.

Technisch kann die UI beispielsweise als Web-UI Technische Möglichkeiten implementiert werden. Wenn die Microservices eine RESTful-HTTP-Schnittstelle haben, sind die Web-UI und die RESTful-HTTP-Schnittstelle ähnlich – beide nutzen HTTP als Protokoll. Die RESTful-HTTP-Schnittstelle liefert JSON oder XML aus, die Web-UI HTML. Ist die UI eine Single-Page-App, so wird der JavaScript-Code ebenfalls über HTTP ausgeliefert und spricht mit der Logik über RESTful HTTP. Bei mobilen Clients ist die technische Umsetzung komplizierter. [Abschnitt 9.1](#) erläutert das im Detail. Technisch kann ein deploybares Artefakt über eine HTTP-Schnittstelle JSON/XML und HTML ausliefern. So implementiert es die UI und erlaubt den Zugriff auf die Logik durch andere Microservices.

Stefan Tilkov nennt den Ansatz nicht »Microservice mit UI«, sondern ein »Self-Contained System« (SCS), also ein in sich abgeschlossenes System [4]. Er unterscheidet zwischen Microservices, die 100 Zeilen lang sein können und von denen es in einem kompletten Projekt mehr als 100 geben kann – und die damit eher dem entsprechen, was dieses Buch als Nanoservices ([Kap. 15](#)) bezeichnet. Ein SCS enthält eine UI und sollte möglichst nicht mit anderen SCS kommunizieren. In einem Gesamtsystem gibt es dann nur fünf bis 25 von diesen SCS. Ein SCS ist etwas, was ein Team gut bewältigen kann. Intern kann es in mehrere Microservices oder Nanoservices aufgeteilt sein.

Es ergeben sich folgende Definitionen:

- SCS (Self-Contained System) ist etwas, was ein Team bearbeitet und eine fachliche Einheit darstellt. Das kann ein Bestellprozess oder eine Registrierung sein. Es implementiert eine sinnvolle Funktionalität und das Team kann das SCS mit neuen Features ergänzen. Ein alternativer Name für ein SCS ist eine Vertikale. Das SCS teilt die Architektur in Fachlichkeiten auf. Das ist ein vertikaler Schnitt im Gegensatz zu einem horizontalen Schnitt. Ein solcher Schnitt würde das System in Schichten aufteilen, die technisch motiviert sind – beispielsweise UI, Logik oder Persistenz.
- Ein Microservice ist ein Teil eines SCS. Es ist eine technische Einheit und kann

unabhängig deployt werden. Das stimmt fast mit der Definition von Microservices in diesem Buch überein. Nur die Größe bei Stefan Tilkov ist so, dass es eher den Nanoservices aus diesem Buch entspricht.

- Dieses Buch spricht von Nanoservices als Einheiten, die immer noch einzeln deploybar sind, aber in einigen Bereichen technische Kompromisse eingehen, um die Größe der Deployment-Einheiten weiter zu reduzieren. Nanoservices haben daher einige technische Eigenschaften der Microservices nicht mehr.

SCS hat die Definition von Microservices in diesem Buch inspiriert. Dennoch spricht nichts dagegen, die UI in ein eigenes Artefakt zu separieren, wenn der Microservice sonst zu groß wird. Sicher ist es wichtiger, einen kleinen wartbaren Microservice zu haben, als die UI zu integrieren. Aber UI und Logik sollten mindestens im selben Team implementiert werden.

4.5 Fazit

Microservices sind ein Modularisierungsansatz. Für ein echtes Verständnis von Microservices sind die in diesem Kapitel gezeigten Perspektiven hilfreich:

- In [Abschnitt 4.1](#) stand die Größe der Microservices im Mittelpunkt. Bei genauerer Betrachtung ist die Größe eines Microservice aber gar nicht so wichtig, wenn es auch Einflussfaktoren gibt. Aber diese Perspektive gibt schon einen ersten Eindruck davon, was ein Microservice sein soll. Die Teamgröße, die Modularisierung und die Ersetzbarkeit der Microservices legen jeweils eine obere Grenze fest. Die untere Grenze kommt von den Transaktionen, der Konsistenz, der Infrastruktur und der verteilten Kommunikation.
- Das Gesetz von Conway ([Abschnitt 4.2](#)) zeigt, dass Architektur und Organisation eines Projekts sehr eng zusammenhängen – sie sind schon fast synonym. Microservices können die Unabhängigkeit von Teams weiter verbessern und so einen Architekturentwurf ideal unterstützen, der das unabhängige Entwickeln von Funktionalitäten vorsieht. Jedes Team ist für einen fachlichen Microservice zuständig, sodass die Teams fachlich weitgehend unabhängig sind. Für Fachlichkeiten ist dann kaum Koordination über Teams hinweg notwendig. Die technischen Abstimmungen können durch die mögliche technische Unabhängigkeit ebenfalls auf ein Mindestmaß reduziert werden.
- Domain-Driven Design in [Abschnitt 4.3](#) gibt schließlich einen sehr guten Eindruck davon, wie die fachliche Aufteilung eines Projekts aussehen kann und wie die einzelnen Teile dann fachlich koordiniert werden können. Jeder Microservice kann einen BOUNDED CONTEXT repräsentieren. Das ist eine abgeschlossene Fachlichkeit mit einem eigenen Domänenmodell. Zwischen den BOUNDED CONTEXTS gibt es verschiedene Kollaborationsmöglichkeiten.
- Schließlich hat [Abschnitt 4.4](#) gezeigt, dass Microservices eine UI enthalten sollten, um so die Änderungen für eine Funktionalität wirklich in einem Microservice umsetzen zu können. Das muss nicht unbedingt eine Deployment-Einheit sein, aber UI und Microservice sollten im Zuständigkeitsbereich desselben Teams sein.

Zusammen ergeben diese unterschiedlichen Perspektiven ein gutes Bild davon, was Microservices ausmacht und wie sie funktionieren können.

Wesentliche Punkte

Anders gesagt: Ein erfolgreiches Projekt benötigt drei Komponenten:

1. Eine Organisation:
Hier hilft das Gesetz von Conways.
2. Ein technisches Vorgehen:
Das können Microservices sein.
3. Einen fachlichen Schnitt, wie ihn DDD und Bounded Context bieten.

Der fachliche Schnitt ist dabei für die langfristige Wartbarkeit sehr wichtig.

Selber ausprobieren und experimentieren

Betrachte die drei Definitionsansätze für Microservices über die Größe, über das Gesetz von Conway und Domain-Driven Design.

- [Abschnitt 1.2](#) hat die wichtigsten Vorteile von Microservices gezeigt. Welches dieser Ziele unterstützen die drei Definitionen besonders gut? DDD und das Gesetz von Conway sind beispielsweise für das bessere Time-to-Market verantwortlich.
- Welcher der drei Aspekte ist deiner Meinung nach der wichtigste? Warum?

4.6 Links & Literatur

- [1] <http://yobriefca.se/blog/2013/04/28/micro-service-architecture/>
- [2] <http://martinfowler.com/bliki/FirstLaw.html>
- [3] <http://martinfowler.com/articles/distributed-objects-microservices.html>
- [4] <https://www.innoq.com/en/talks/2014/12/talk-microservices-modularization-softwarearchitecture-berlin/>
- [5] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 978-0-32112-521-7
- [6] <http://www.melconway.com/research/committees.html>
- [7] <http://sequoia.cs.byu.edu/lab/files/pubs/Bailey2013.pdf>

5 Gründe für Microservices

Microservices haben viele Vorteile, die dieses Kapitel darstellt. Das Verständnis für die Vorteile ermöglicht eine bessere Bewertung, ob in einem Einsatzkontext Microservices sinnvoll sind. Das Kapitel nimmt [Abschnitt 1.2](#) auf und stellt die Vorteile detaillierter dar.

[Abschnitt 5.1](#) zeigt die technischen Vorteile von Microservices. Aber Microservices beeinflussen auch die Organisation, wie [Abschnitt 5.2](#) zeigt. Zum Abschluss geht [Abschnitt 5.3](#) auf die Vorteile aus der geschäftlichen Sicht ein.

5.1 Technische Vorteile

Microservices sind ein effektives Modularisierungskonzept. Nur verteilte Kommunikation ermöglicht es, einen anderen Microservice aufzurufen. Das passiert nicht einfach so, sondern ein Entwickler muss dazu in der Kommunikationsinfrastruktur entsprechende Möglichkeiten schaffen. Dadurch schleichen sich Abhängigkeiten zwischen Microservices nicht einfach so ein, sondern ein Entwickler muss sie explizit einbauen. Ohne Microservices kann es sehr leicht passieren, dass ein Entwickler einfach irgendeine Klasse nutzt und so eine Abhängigkeit schafft, die architektonisch gar nicht gewünscht ist.

Nehmen wir beispielsweise an, dass in einer E-Commerce-Anwendung die Suche zwar den Bestellprozess aufrufen soll, aber nicht umgekehrt. Dadurch kann die Suche geändert werden, ohne dass es den Bestellprozess beeinflusst – denn die Suche nutzt den Bestellprozess nicht. Nun wird eine Abhängigkeit der Suche vom Bestellprozess eingeführt, weil beispielsweise ein Entwickler dort Funktionalitäten gefunden hat, die ihm helfen. Jetzt sind Suche und Bestellprozess wechselseitig voneinander abhängig. Sie können nun nur noch zusammen geändert werden.

Wenn sich einmal ungewünschte Abhängigkeiten in das System eingeschlichen haben, kommen schnell weitere Abhängigkeiten dazu. Die Architektur der Anwendung erodiert. Das können eigentlich nur Architekturmanagement-Werkzeuge vermeiden. Solche Werkzeuge haben ein Modell der gewünschten Architektur und stellen fest, wenn ein Entwickler eine ungewünschte Abhängigkeit eingeführt hat. Der Entwickler kann die Abhängigkeit sofort wieder entfernen, bevor größerer Schaden entsteht und die Architektur weiter erodiert. Werkzeuge dafür zeigt [Abschnitt 8.2](#).

In einer Microservices-Architektur wären Suche und Bestellprozess getrennte Microservices. Der Entwickler muss eine Abhängigkeit in den Kommunikationsmechanismen der Microservices einführen. Das ist aufwendig und fällt auch ohne Architekturmanagement-Werkzeug leichter auf. Also ist die Wahrscheinlichkeit geringer, dass die Architektur auf Ebene der Abhängigkeiten zwischen Microservices erodiert. Die Grenzen der Microservices sind sozusagen Firewalls, die einen Zerfall der Architektur aufhalten. Microservices bieten eine starke Modularisierung, weil es schwierig ist, die Grenzen zwischen den Modulen zu überschreiten.

Der Umgang mit alten Software-Systemen ist eine große Herausforderung: Die weitere Entwicklung der

Ersetzen von Microservices

Software ist schwierig, weil die Codequalität schlecht ist. Die Software zu ersetzen ist risikoreich. Oft ist nicht klar, wie die Software genau funktioniert, und das System ist oft sehr groß. Je größer das Software-System, desto höher der Aufwand für eine Ablösung. Wenn die Software auch noch wichtige Geschäftsprozesse unterstützt, ist es praktisch unmöglich, die Software noch zu ändern. Ein Ausfall solcher Geschäftsprozesse kann erhebliche Konsequenzen haben. Jede Änderung birgt die Gefahr eines Ausfalls.

Obwohl dieses Problem so zentral ist, hat eine Software-Architektur eigentlich nie das Ersetzen der Software als Ziel. Microservices unterstützen dieses Ziel: Sie können einzeln abgelöst werden, weil sie eine getrennte und kleine Deployment-Einheit sind. Dadurch sind die technischen Voraussetzungen für eine Ablösung besser. Schließlich muss nicht ein großes Software-System ersetzt werden, sondern nur ein kleiner Microservice. Dann können bei Bedarf weitere Microservices ersetzt werden.

Bei den neuen Microservices sind die Entwickler nicht an den alten Technologie-Stack gebunden, sondern können beliebige andere Technologien nutzen. Wenn der Microservice zusätzlich im fachlichen Sinn eigenständig ist, ist die Logik einfacher zu verstehen. Der Entwickler muss nicht das gesamte System verstehen, sondern nur die Fachlichkeit des einen Microservice. Die Kenntnis über die Fachlichkeit ist eine Voraussetzung für die erfolgreiche Ablösung eines Microservice.

Hinzu kommt, dass Microservices auch beim Ausfall eines anderen Microservice weiter funktionieren. Selbst wenn die Ablösung eines Microservice zu einem vorübergehenden Ausfall des Microservice führt, kann das Gesamtsystem dennoch weiter laufen. Das senkt das Risiko einer Ablösung weiter.

Der Start in einem neuen Projekt ist einfach: Es gibt noch nicht viel Code, die Strukturierung des Codes ist sauber und die Entwickler kommen schnell voran. Durch die Erosion der Architektur und die größere Komplexität kann die Entwicklung schwieriger werden. Irgendwann ist die Software ein Legacy-System. Wie bereits dargestellt, verhindern Microservices die Erosion der Architektur. Wenn ein Microservice ein Legacy-System geworden ist, kann er ersetzt werden. Aus diesen beiden Gründen können Microservices eine nachhaltige Software-Entwicklung ermöglichen. Damit ist gemeint, dass auch langfristig eine hohe Produktivität erreicht werden kann. Allerdings kann es auch in einem Microservice-System vorkommen, dass viel Code neu geschrieben wird. Darunter leidet die Produktivität dann natürlich.

Das Ersetzen von Microservices ist nur dann möglich, wenn das System bereits als Microservices umgesetzt worden ist. Aber auch die Ablösung und das Ergänzen vorhandener Legacy-Anwendungen ist mit Microservices einfacher. Die Legacy-Anwendungen müssen nur eine Schnittstelle bereitstellen, mit der Microservices mit der Legacy-Anwendung kommunizieren können. Umfangreiche Änderungen am Code oder eine Integration neuer Code-Bestandteile in das Legacy-System sind nicht nötig. Die Integration auf Code-Ebene ist bei Legacy-Systemen eine große Herausforderung, die so vermieden wird.

Besonders einfach ist das Ergänzen des Systems, wenn ein Microservice die Abarbeitung jedes Aufrufs abfangen und den Aufruf gegebenenfalls selber abarbeiten

kann. Solche Aufrufe können HTTP-Requests zum Aufbau von Webseiten sein oder auch REST-Aufrufe.

Dann kann der Microservice das Legacy-System ergänzen. Dazu gibt es verschiedene Möglichkeiten:

- Der Microservice kann bestimmte Anfragen selber abarbeiten und dem Legacy-System die restlichen Anfragen überlassen.
- Alternativ kann der Microservice Anfragen ändern und sie dann an die eigentliche Anwendung weiterleiten.

Dieses Vorgehen ist dem SOA-Ansatz (siehe [Kap. 7](#)) ähnlich, bei dem es um eine umfassende Integration von verschiedenen Anwendungen geht. Wenn die Anwendungen in Services aufgeteilt sind, können diese Services nicht nur neu orchestriert werden, sondern es ist ebenfalls möglich, einzelne Services zum Beispiel durch Microservices zu ersetzen.

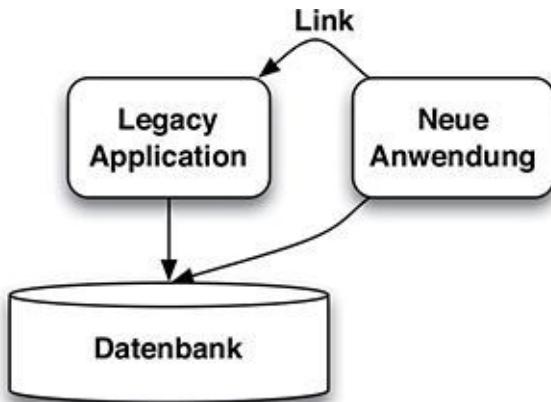
Ein Beispiel für Microservices und Legacy

In einem Projekt war die Aufgabe, in einer vorhandenen Java-E-Commerce-Anwendung eine Erneuerung durchzuführen. Dazu sollten neue Technologien wie beispielsweise neue Frameworks eingeführt werden, um in Zukunft produktiver Software zu entwickeln. Nach einiger Zeit stellte sich heraus, dass der Aufwand für die Integration der neuen und alten Technologien gewaltig sein würde. Der neue Code muss den alten aufrufen können – und umgekehrt. Dazu ist eine Integration der Technologien in beide Richtungen notwendig. Transaktionen und Datenbankverbindungen müssen gemeinsam genutzt werden. Ebenso müssen die Sicherheitsmechanismen integriert werden. Diese Integration würde auch die Entwicklung der neuen Software komplizierter machen und so das Ziel des Vorhabens gefährden.

Die Lösung zeigt [Abbildung 5–1](#) Das neue System wurde vollständig getrennt vom alten System entwickelt. Die einzige Integration waren Links, die in der alten Software entsprechendes Verhalten aufrufen – beispielsweise das Hinzufügen von Waren zum Einkaufswagen. Und das neue System hatte Zugriff auf dieselbe Datenbank wie das alte System. Eine gemeinsame Datenbank ist im Nachhinein keine besonders gute Idee, weil die Datenbank eine interne Repräsentation der Daten des alten Systems ist. Wird diese Darstellung einer anderen Anwendung zur Verfügung gestellt, verstößt das gegen das Prinzip der Kapselung [5] (siehe auch [Abschnitt 10.1](#)). Die Datenstrukturen können kaum noch geändert werden, weil neben dem alten System nun auch das neue System von den Datenstrukturen abhängt.

Der Ansatz, das System getrennt zu entwickeln, hat die Probleme bei der Integration weitgehend gelöst. Vor allem konnten die Entwickler so neue technologische Ansätze ohne Rücksicht auf den alten Code und die alten Herangehensweisen verwenden, was wesentlich elegantere Lösungen möglich machte.

Abb. 5–1 Beispiel für Legacy-Integration

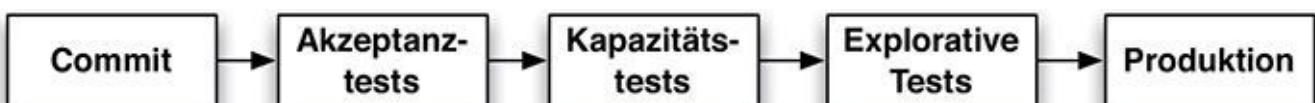


Continuous Delivery [1] bringt Software durch einen Continuous Delivery einfach reproduzierbaren Prozess regelmäßig in Produktion. Dazu dient eine Continuous-Delivery-Pipeline (siehe Abb. 5–2):

- In der Commit-Phase wird die Software kompiliert, die Unit-Tests werden durchgeführt und es wird gegebenenfalls eine statische Code-Analyse durchgeführt.
- Die automatisierten Akzeptanztests in der nächsten Phase stellen sicher, dass die Software fachlich korrekt ist, sodass sie vom Kunden akzeptiert würde.
- Kapazitätstests überprüfen, ob die Software performant genug ist, um die erwartete Anzahl Nutzer zu unterstützen. Auch diese Tests sind automatisiert.
- Die explorativen Tests hingegen sind manuell und dienen dazu, bestimmte Bereiche des Systems zu testen. Dazu können neue Features zählen oder bestimmte Aspekte wie die Sicherheit des Systems.
- Schließlich wird die Software in Produktion gebracht. Auch dieser Prozess ist idealerweise automatisiert.

Software wird in die einzelnen Phasen promotet: Sie durchläuft die einzelnen Phasen nacheinander. Ein Release kann zum Beispiel die Akzeptanztests erfolgreich bestehen. Bei den Kapazitätstests stellt sich aber heraus, dass die Software den Anforderungen bezüglich des Lastverhaltens nicht gerecht wird. Dann wird die Software nie in die weiteren Phasen wie den explorativen Test oder gar die Produktion promotet.

Abb. 5–2 Continuous-Delivery-Pipeline



Die Continuous-Delivery-Pipeline mit einer vollständigen Automatisierung ist das Optimum. Irgendwie kommt aber alle Software in Produktion. Also kann der aktuelle Prozess dem Optimum schrittweise angeglichen werden.

Continuous Delivery ist mit Microservices besonders gut umsetzbar [2]. Microservices sind eigenständige Deployment-Einheiten. Daher können sie unabhängig von anderen Services in Produktion gebracht werden. Das hat erhebliche Auswirkungen auf die Continuous-Delivery-Pipeline:

- Der Durchlauf durch die Pipeline ist schneller, weil nur ein kleiner Microservice

getestet und in Produktion gebracht werden muss. Das beschleunigt das Feedback. Schnelles Feedback ist ein wesentliches Ziel von Continuous Delivery. Wenn ein Entwickler erst nach Wochen erfährt, dass sein Code ein Problem in Produktion verursacht hat, ist es schwierig, sich wieder in den Code einzuarbeiten und das Problem zu analysieren. Schnelles Feedback ist ein zentrales Ziel von Continuous Delivery.

- Das Risiko des Deployments sinkt. Die deployten Einheiten sind kleiner und außerdem können Microservice-Systeme auch bei Ausfall einiger Microservices weitergenutzt werden. Und das Deployment kann leichter zurückgerollt werden.
- Maßnahmen zur weiteren Verringerung des Risikos sind mit kleinen Deployment-Einheiten ebenfalls leichter umzusetzen. Bei Blue/Green Deployment wird beispielsweise eine neue Umgebung mit dem neuen Release aufgebaut. Ähnliches gilt für Canary Releasing: Bei diesem Vorgehen wird nur ein Server zunächst mit der neuen Version der Software versorgt. Erst wenn dieser Server erfolgreich in Produktion ist, wird die Version auch auf den anderen Server ausgerollt. Bei einem Deployment-Monolithen kann diese Vorgehensweise schwierig oder fast unmöglich umsetzbar sein, weil dafür viele Ressourcen für die vielen Umgebungen notwendig sind. Bei Microservices hingegen sind die benötigten Umgebungen wesentlich kleiner und das Vorgehen daher einfacher.
- Eine weitere Herausforderung sind oft Testumgebungen. Wenn beispielsweise ein Drittsystem genutzt wird, muss in der Umgebung auch eine Testversion dieses Drittsystems vorhanden sein. Bei kleineren Deployment-Einheiten sind die Anforderungen an die Umgebungen geringer. Die Umgebungen für die Microservices müssen nur die Drittsysteme integrieren, die für den jeweiligen Microservice notwendig sind. Es ist ebenfalls möglich, die Systeme mit Mocks der Drittsysteme zu testen. Das vereinfacht den Test und ist auch eine interessante Methode, um Microservices unabhängig voneinander zu testen.

Continuous Delivery ist einer der wichtigsten Gründe für Microservices. Viele Projekte investieren in eine Migration zu Microservices, um so den Aufbau der Continuous-Delivery-Pipeline zu vereinfachen.

Aber Continuous Delivery ist auch eine Voraussetzung für Microservices. Ohne Continuous-Delivery-Pipelines können die vielen Microservices kaum in Produktion gebracht werden, weil es praktisch nicht möglich ist, so viele Microservices manuell in Produktion zu bringen. Also profitieren Microservices von Continuous Delivery und umgekehrt [3].

Microservices bieten über das Netzwerk erreichbare Skalierung Schnittstellen an, die beispielsweise über HTTP oder über eine Message-Lösung angesprochen werden. Jeder Microservice kann auf einem Rechner laufen – oder mehreren. Wenn der Service auf mehreren Servern läuft, kann die Last auf die Server verteilt werden. Ebenso ist es möglich, Microservices auf unterschiedlich schnellen Rechnern zu installieren und zu betreiben. Jeder Microservice kann eine eigene Skalierung umsetzen.

Hinzu kommt, dass vor einen Microservice auch ein Cache gesetzt werden kann. Bei

REST-basierten Microservices kann es ausreichen, einen generischen HTTP-Cache zu verwenden. Das reduziert den Aufwand für einen solchen Cache deutlich. Das HTTP-Protokoll hat eine umfassende Unterstützung für Caching, die in diesem Zusammenhang sehr hilfreich ist.

Ebenso kann es gegebenenfalls möglich sein, die Microservices an verschiedenen Stellen im Netzwerk zu installieren und so näher an den Aufrufer heranzubekommen. Bei einer weltweit verteilten Cloud-Umgebung ist es praktisch egal, in welchem Rechenzentrum die Microservices laufen. Wenn die Microservice-Infrastruktur mehrere Rechenzentren nutzt und Anfragen jeweils von dem nächstgelegenen Rechenzentrum aus bearbeitet, kann die Architektur die Antwortzeiten erheblich reduzieren. Statische Inhalte können außerdem von einem CDN (Content Delivery Network) ausgeliefert werden, dessen Server noch näher an den Usern positioniert sind.

Allerdings sollte man von der besseren Skalierung und der Unterstützung für Caching keine Wunder erwarten: Microservices führen zu einer verteilten Architektur. Aufrufe durch das Netzwerk sind deutlich langsamer als lokale Aufrufe. Aus einer reinen Performance-Perspektive kann es sinnvoller sein, mehrere Microservices zusammenzulegen oder Technologien zu nutzen, die auf lokale Aufrufe setzen (siehe Kap. 15).

Eigentlich müssten Microservices weniger zuverlässig sein als andere Architekturansätze. Schließlich sind Microservices ein verteiltes System. Zu den üblichen Fehlerquellen der Systeme kommt ein möglicher Ausfall des Netzwerks hinzu. Und die Microservices laufen auf mehreren Servern, sodass es auch eine größere Wahrscheinlichkeit für einen Hardware-Ausfall gibt.

Um eine hohe Verfügbarkeit zu ermöglichen, muss die Microservices-Architektur entsprechend aufgebaut werden. Die Kommunikation zwischen den Microservices muss eine Art Firewall bilden: Ein Ausfall eines Microservice darf sich nicht fortpflanzen. Dadurch wird vermieden, dass ein Problem in nur einem Microservice das gesamte System zum Ausfall bringen kann.

Dazu muss der aufrufende Microservice beim Ausfall irgendwie sinnvoll weiterarbeiten. Beispielsweise kann ein Default-Wert angenommen werden. Oder der Ausfall führt zu einem anderweitig reduzierten Service.

Schon der technische Umgang mit einem Ausfall kann entscheidend sein: Der Timeout auf Betriebssystemebene für TCP/IP-Verbindungen ist oft auf beispielsweise fünf Minuten eingestellt. Wenn durch den Ausfall eines Microservice Requests gegen diesen Timeout laufen, dann ist der Thread fünf Minuten blockiert. Irgendwann sind alle Threads blockiert. Dann kann das aufrufende System ausfallen, weil es nur noch damit beschäftigt ist, auf Timeouts zu warten. So etwas kann vermieden werden, indem die Aufrufe mit einem kleineren Timeout versehen werden. Solche Ideen gibt es wesentlich länger als Microservices. [4] stellt diese Herausforderungen und Lösungsansätze ausführlich dar. Wenn diese Ansätze tatsächlich umgesetzt werden, kann ein Microservice-System den Ausfall ganzer Microservices tolerieren und so robuster als ein Deployment-Monolith werden.

Gegenüber einem Deployment-Monolith haben Microservices außerdem den Vorteil,

dass sie das System in mehrere Prozesse aufteilen. Diese Prozesse sind besser gegeneinander isoliert. In einem Deployment-Monolithen, der nur einen Prozess startet, kann ein Speicherleck oder eine Funktionalität, die viel Rechenzeit verbraucht, das gesamte System zum Ausfall bringen. Solche Fehler sind oft sehr einfache Programmierfehler oder Flüchtigkeitsfehler. Durch die Aufteilung in Microservices sind solche Situationen unmöglich, da nur jeweils ein Microservice ausfallen würde.

Microservices bieten technologische Freiheiten. Weil Microservices nur über das Netzwerk miteinander kommunizieren, können sie in jeder beliebigen Sprache und Plattform umgesetzt werden, solange die Kommunikation mit den anderen Microservices möglich ist. Diese Wahlfreiheit kann dazu genutzt werden, neue Technologien mit einem begrenzten Risiko auszuprobieren. Es kann einfach ein Microservice mit der neuen Technologie umgesetzt werden. Wenn die Technologie sich doch nicht als tauglich erweist, muss nur dieser eine Microservice neu geschrieben werden. Auch bei einem Ausfall sind die Probleme begrenzt. Die Technologiefreiheit hat beispielsweise den Vorteil, dass Entwickler neue Technologien tatsächlich produktiv nutzen können. Das erhöht die Motivation und hat positive Auswirkungen auf die Personalsituation, da Entwickler üblicherweise gerne neue Technologien nutzen.

Außerdem kann für jedes Problem die geeignete Technologie genutzt werden. So ist es möglich, eine andere Programmiersprache oder ein bestimmtes Framework zu nutzen, um bestimmte Teile des Systems umzusetzen. Es ist sogar denkbar, dass ein einzelner Microservice eine spezielle Datenbank oder andere Persistenztechnologie nutzt. Dafür müssen aber Backup- und Desaster-Recovery-Mechanismen umgesetzt werden.

Die Technologiefreiheit ist eine Option – sie muss nicht unbedingt genutzt werden. Es können auch Technologien vorgeschrieben werden. Aber Deployment-Monolithen zwingen Entwickler in ein enges Korsett: In Java-Anwendungen kann beispielsweise jede Bibliothek nur in einer Version genutzt werden. Deswegen müssen nicht nur die Bibliotheken, sondern sogar die Versionen festgelegt werden. Microservices haben keine solchen technischen Zwänge.

Die Entscheidungen bezüglich der Technologien und bezüglich der Produktivstellung neuer Releases betreffen nur jeweils einen Microservice. Damit sind die Microservices weitgehend unabhängig voneinander. Natürlich muss es eine gemeinsame technische Basis geben. Die Installation der Microservices sollte automatisiert sein, es sollte für jeden Service eine Continuous-Delivery-Pipeline geben und die Microservices sollten sich an Vorgaben für das Monitoring halten. Aber im Rahmen dieser Einschränkungen können die Microservices praktisch beliebige technische Ansätze umsetzen. Wegen der größeren technologischen Freiheiten ist weniger Koordination zwischen den Microservices notwendig.

5.2 Organisatorische Vorteile

Microservices sind ein Architekturansatz und sollten daher eigentlich nur Vorteile für die Entwicklung und die Struktur der Software haben. Aber wegen des Gesetzes von Conway (siehe [Abschnitt 4.2](#)) hat die Architektur auch Auswirkungen auf die Teamkommunikation

und damit auf die Organisation.

Vor allem erreichen Microservices ein hohes Maß an technischer Unabhängigkeit, wie der letzte [Abschnitt 5.1](#) gezeigt hat. Wenn in der Organisation ein Team für ein Microservice voll verantwortlich ist, kann das Team die technische Unabhängigkeit komplett ausnutzen. Das Team ist dann aber voll dafür verantwortlich, wenn ein Microservice nicht funktioniert und in Produktion ausfällt.

So unterstützen Microservices die Selbstständigkeit der Teams. Die technische Basis erlaubt eine Arbeit an den verschiedenen Microservices mit wenig Koordination. Das legt das Fundament für die selbstständige Arbeit der Teams.

In anderen Projekten muss die Technologie oder die Architektur zentral vorgegeben werden, weil die einzelnen Teams und Module durch die technischen Rahmenbedingungen an diese Entscheidungen gebunden sind. Es kann schlicht unmöglich sein, zwei Bibliotheken oder zwei unterschiedliche Versionen einer Bibliothek in einem Deployment-Monolithen zu nutzen. Das erzwingt eine zentrale Koordination. Bei Microservices ist die Situation anders. Das erlaubt Selbstorganisation. Eine globale Koordination kann dennoch sinnvoll sein, um beispielsweise bei einem Sicherheitsproblem mit einer Bibliothek ein Update in allen Komponenten durchzuführen.

Teams sind für mehr verantwortlich: Sie entscheiden die Architektur der Microservices. Die Verantwortung dafür können sie nicht an eine zentrale Architektur oder Vorgaben abgeben. Daher müssen sie auch die Konsequenzen tragen, denn sie übernehmen die Verantwortung für den Microservice.

Die Scala-Entscheidung

In einem Projekt mit Microservices-Ansatz sollte die zentrale Architekturgruppe entscheiden, ob Scala als Programmiersprache in einem Team genutzt werden darf. Die Entscheidung hätte der zentralen Architekturgruppe die Verantwortung für die Entscheidung übertragen. Sie hätte entscheiden müssen, ob das Team mit dieser Sprache seine Probleme effizienter lösen kann oder ob wahrscheinlich Probleme auftauchen. Letztendlich ist die Entscheidung an das Team delegiert worden. Denn das Team muss die Verantwortung für den Microservice übernehmen. Sie müssen damit leben, wenn Scala doch nicht den Anforderungen in Produktion gerecht wird oder doch keine effizientere Entwicklung ermöglicht. Sie haben den Aufwand der Einarbeitung und müssen abschätzen, ob der Aufwand sich rentiert. Ebenso haben sie ein Problem, wenn plötzlich alle Scala-Entwickler das Projekt verlassen oder in andere Teams wechseln. Die Verantwortung an die zentrale Architekturgruppe abzugeben, ist genau genommen gar nicht möglich, denn die zentrale Architektur ist von den Konsequenzen nicht unmittelbar betroffen. Daher muss das Team zwingend die Entscheidung selbst treffen. Das Team muss alle Teammitglieder einbeziehen – auch den Product Owner, der am Ende zum Beispiel an einer niedrigen Produktivität leiden würde.

Dieses Vorgehen ist eine radikale Abkehr von alten Organisationsformen, bei denen die zentrale Architekturgruppe oder gar der Betrieb den Technologie-Stack vorschreiben. In diesen Organisationen haben die Teams nicht die Verantwortung für die Entscheidungen und nichtfunktionale Eigenschaften wie Verfügbarkeit, Performance oder Skalierbarkeit.

Die nichtfunktionalen Eigenschaften können in einer klassischen Architektur ausschließlich zentral gewährleistet werden, weil sie nur durch die gemeinsame Basis des gesamten Systems sichergestellt werden kann. Wenn Microservices eine gemeinsame Basis nicht mehr erzwingen, können diese Entscheidungen an die Teams verteilt und so eine größere Selbstständigkeit erreicht werden.

Letztendlich erlauben Microservices die Aufteilung großer Projekte in eine Menge kleiner Projekte, weil die einzelnen Microservices so unabhängig sind, dass eine zentrale Koordinierung nicht mehr so wichtig ist. Dadurch ist keine umfangreiche Projektorganisation notwendig. Große Organisationen sind problematisch, weil eine große Organisation in der Relation einen größeren Kommunikations-Overhead hat. Wenn Microservices erlauben, eine große Organisation in mehrere kleinere Organisationen zu zerschlagen, sinkt der Kommunikationsbedarf. Dadurch können sich die Teams mehr auf die Umsetzung der Anforderungen fokussieren.

Große Projekte schlagen auch eher fehl. Auch aus dieser Perspektive ist es besser, wenn ein großes Projekt in viele kleinere Projekte aufgeteilt wird. Der kleinere Umfang der einzelnen Projekte erlaubt genauere Schätzungen. Bessere Schätzungen verbessern die Planbarkeit und haben ein geringeres Risiko. Und selbst wenn die Schätzung nicht stimmt, ist der Verlust bei Fehlentscheidungen geringer. Zusammen mit der größeren Flexibilität kann das dazu führen, dass Entscheidungen schneller und einfacher getroffen werden können – weil das Risiko eben so viel geringer ist.

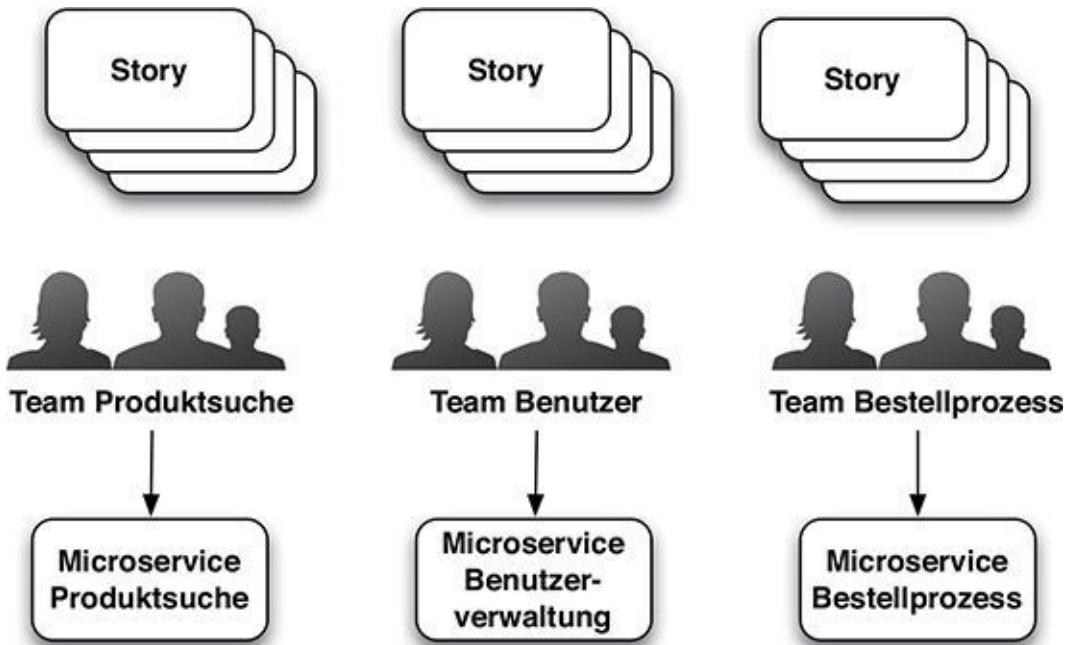
5.3 Vorteile aus Geschäftssicht

Die bereits erwähnten Vorteile aus der Organisationssicht führen zu geschäftlichen Vorteilen: Das Risiko der Projekte sinkt und die Koordination zwischen den Teams wird weniger intensiv, sodass die Teams effizienter arbeiten können.

Die Aufteilung in Microservices ermöglicht eine parallel Abarbeitung von unterschiedlichen Stories (siehe [Abb. 5–3](#)). Jedes Team arbeitet an einer Story, die nur den eigenen Microservice betrifft. Dadurch können die Teams unabhängig arbeiten und das Gesamtsystem gleichzeitig an verschiedenen Stellen erweitert werden. Das skaliert letztendlich den agilen Prozess. Aber die Skalierung findet nicht auf Ebene des Entwicklungsprozesses statt, sondern durch die unabhängige Architektur. Änderungen und Deployments der einzelnen Microservices sind ohne größere Koordinierung möglich. Daher können die Teams unabhängig voneinander arbeiten. Wenn ein Team langsamer ist oder auf Schwierigkeiten stößt, sind die anderen Teams davon kaum beeinflusst. Das reduziert das Risiko des Projekts weiter.

Durch einen eindeutigen fachlichen Schnitt und die Zuordnung von einem Entwicklerteam pro Microservice kann die Entwicklungsorganisation oder die Projektorganisation mit der Anzahl der Säulen skaliert werden.

Abb. 5–3 Mehrere Anforderungsströme



Es ist denkbar, dass Änderungen mehrere Microservices und damit mehrere Teams betreffen. Ein Beispiel: Nur bestimmte Benutzer dürfen einige Produkte bestellen – beispielsweise wegen des Jugendschutzes. Für dieses Feature würden bei der Architektur aus [Abbildung 5–3](#) Änderungen an allen Microservices notwendig. Die Benutzerverwaltung müsste speichern, ob ein Kunde volljährig ist. Die Produktsuche sollte die Produkte ausblenden oder markieren. Und der Bestellprozess muss schließlich die Bestellung der Produkte vermeiden. Die Änderungen müssen aufeinander abgestimmt werden. Das erfordert Koordination – insbesondere, wenn ein Microservice einen anderen aufruft. Dann muss der aufgerufene Microservice zuerst geändert werden, damit der Aufrufer anschließend die neuen Features nutzen kann.

Das Problem ist allerdings lösbar. Man kann argumentieren, dass die hier gezeigte Architektur nicht optimal ist. Wenn die Architektur sich an den Geschäftsprozessen orientiert, könnte die Änderung auf den Bestellprozess begrenzt werden. Schließlich soll nur das Bestellen verboten werden – nicht die Suche. Die Information, ob ein Kunde etwas bestellen kann oder nicht, müsste auch in der Verantwortung des Bestellprozesses liegen. Welche Architektur und damit welche Teamaufteilung richtig sind, hängt von den fachlichen Änderungen und den davon betroffenen Microservices und Teams ab.

Wenn die Architektur passend gewählt ist, können Microservices Agilität sehr gut unterstützen. Und das ist sicher ein guter Grund für Microservices-Architekturen aus Geschäftssicht.

5.4 Fazit

Zusammengefasst ergeben sich die folgenden technischen Vorteile ([Abschnitt 5.1](#)):

- **Starke Modularisierung:**
Abhängigkeiten zwischen Microservices können sich nicht einfach einschleichen.
- Microservices können recht *einfach ersetzt* werden.
- Die starke Modularisierung und die Ersetzbarkeit der Microservices führen zu einer

nachhaltigen Entwicklungsgeschwindigkeit: Die Architektur bleibt stabil und Microservices, die nicht mehr wartbar sind, können ersetzt werden. So bleibt die Qualität des Systems auch langfristig erhalten und daher das System wartbar.

- *Legacy-Systeme* können mit Microservices ergänzt werden, ohne dass man den Ballast des Legacy-Systems mit sich umherschleppen muss. Also sind Microservices auch ein guter Ansatz beim Umgang mit Legacy-Systemen.
- Weil Microservices kleine Deployment-Einheiten sind, kann eine *Continuous-Delivery-Pipeline* viel einfacher aufgebaut werden.
- Microservices können unabhängig voneinander *skaliert* werden.
- Wenn die Microservices entsprechend den etablierten Ansätzen implementiert werden, ist das System letztendlich *robuster*.
- Jeder Microservice kann in einer anderen Programmiersprache und mit einer *anderen Technologie* umgesetzt werden.
- Dadurch sind die Microservices auf einer technischen Ebene weitgehend *unabhängig* voneinander.

Die technische Unabhängigkeit hat Auswirkungen auf die Organisation ([Abschnitt 5.2](#)): Die Teams können selbstständig und eigenverantwortlich arbeiten. Es ist weniger zentrale Koordinierung notwendig. Ein großes Projekt wird so durch eine Sammlung kleiner Projekte ersetzt, was Risiko und Koordinierung positiv beeinflusst.

Aus der Geschäftssicht sind alleine die Auswirkungen auf das Risiko schon positiv ([Abschnitt 5.3](#)). Aber noch attraktiver ist, dass die Architektur die Skalierung agiler Prozesse erlaubt, ohne dabei übermäßig viel Koordination und Kommunikation zu benötigen.

Wesentliche Punkte

- Es gibt viele technische Vorteile – von Skalierbarkeit über Robustheit bis hin zur nachhaltigen Entwicklung.
- Die technische Unabhängigkeit ergibt Vorteile auch für die Organisation. Teams werden unabhängig.
- Die technischen und organisatorischen Vorteile zusammen ergeben Vorteile aus einer Geschäftsperspektive: niedrigeres Risiko und schnelleres Umsetzen von mehr Features.

Selber ausprobieren und experimentieren

Betrachte ein dir bekanntes Projekt:

- Warum sind Microservices für dieses Szenario sinnvoll? Bewerte jeden Vorteil mit einem Punkt (kein echter Vorteil) bis zehn Punkten (sehr großer Vorteil). Die Vorteile sind im Fazit noch einmal aufgelistet.
- Wie würde das Projekt mit und ohne die Nutzung von Microservices aussehen?

- Entwickle eine Diskussion der Vorteile von Microservices aus der Sicht eines Architekten, eines Entwicklers, eines Projektleiters und des Kunden für das Projekt. Die technischen Vorteile werden eher für Entwickler und Architekten interessant sein, die organisatorischen und wirtschaftlichen eher für Projektleiter und Kunden. Welche hast du jeweils besonders vor?
- Visualisiere den derzeitigen fachlichen Schnitt in deinem Projekt oder Produkt. Welche Teams sind für welche Teile verantwortlich? Wo gibt es Überschneidungen? Wie sollte die Zuordnung von Teams zu Produktteilen und Services für eine möglichst unabhängige Arbeitsweise aussehen?

5.5 Links & Literatur

- [1] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, dpunkt.verlag, 2014, ISBN 978-3864902086
- [2] <http://de.slideshare.net/ewolff/software-architecture-for-devops-and-continuousdelivery>
- [3] <http://de.slideshare.net/ewolff/continuous-delivery-and-micro-services-a-symbiosis>
- [4] Michael T. Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8
- [5] [http://de.wikipedia.org/wiki/Datenkapselung_\(Programmierung\)](http://de.wikipedia.org/wiki/Datenkapselung_(Programmierung))

6 Herausforderungen bei Microservices

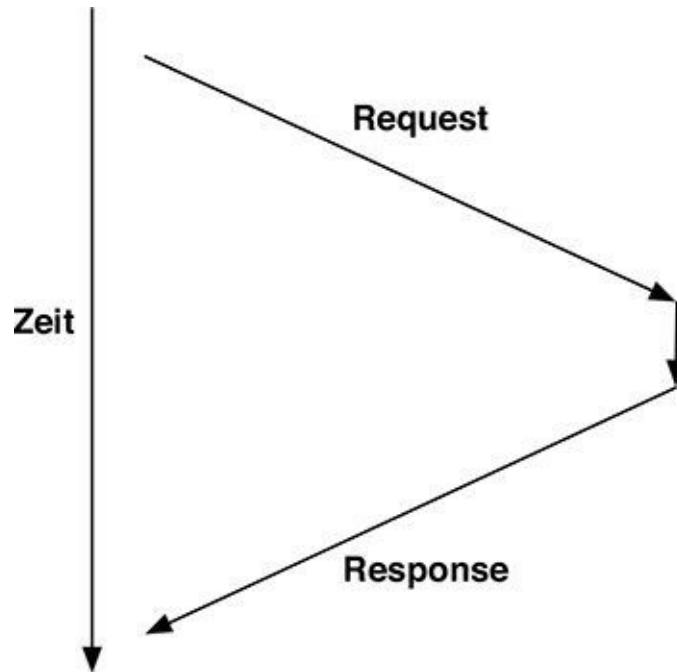
Die Aufteilung eines Systems in Microservices bedeutet eine höhere Komplexität. Dabei entstehen Herausforderungen auf der technischen Ebene (siehe [Abschnitt 6.1](#)) – beispielsweise hohe Latenzen im Netzwerk oder der Ausfall einzelner Services. Aber auch auf Ebene der Software-Architektur gibt es einiges zu beachten – unter anderem wegen der schlechten Änderbarkeit der Architektur ([Abschnitt 6.2](#)). Und schließlich gibt es viel mehr einzeln auslieferbare Bestandteile, sodass der Betrieb und die Infrastruktur komplexer werden ([Abschnitt 6.3](#)). Diese Herausforderungen müssen bei der Einführung von Microservices beachtet werden. Maßnahmen aus den folgenden Kapiteln zeigen, wie man mit den Herausforderungen umgehen kann.

6.1 Technische Herausforderungen

Microservices sind verteilte Systeme. Die Aufrufe zwischen den Microservices gehen über das Netzwerk. Das beeinflusst die Latenz und damit die Antwortzeiten der Microservices negativ. Die schon erwähnte erste Regel für verteilte Objekte besagt, dass Objekte nach Möglichkeit nicht verteilt werden sollen (siehe [Abschnitt 4.1](#)).

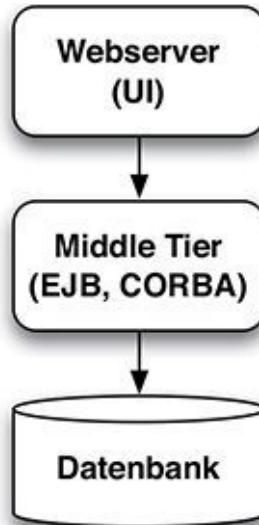
Warum das so ist, zeigt [Abbildung 6–1](#). Ein Aufruf muss über das Netz zum Server gelangen, wird dort verarbeitet und kommt zum Aufrufer zurück. Die Latenz nur für die Netzwerkkommunikation kann im selben Rechenzentrum ca. 0,5 ms betragen (siehe [2]). In dieser Zeit kann ein Prozessor, der mit 3 GHz läuft, ca. 1,5 Millionen Instruktionen bearbeiten. Wenn eine Berechnung auf einen anderen Knoten verlegt wird, sollte überprüft werden, ob das lokale Abarbeiten des Requests nicht schneller ist. Die Latenz kann durch das Marshalling und Unmarshalling der Parameter für den Aufruf und des Ergebnisses des Aufrufs noch höher ausfallen. Auf der anderen Seite können Optimierungen am Netz oder das Zusammenlegen der Knoten an demselben Switch zu Vorteilen führen.

Abb. 6–1 Latenz bei einem Aufruf über das Netz



Die erste Regel für verteilte Objekte und die Warnung vor der Latenz im Netzwerk kommt aus der Zeit, in der CORBA und EJB verwendet wurden. Diese Technologien wurden oft für verteilte Drei-Schicht-Architekturen verwendet (siehe Abb. 6–2). Für jeden Request eines Kunden implementiert der Web-Tier nur das Rendern der Daten als HTML-Seite. Die Logik ist auf einem anderen Server, der durch einen Aufruf durch das Netzwerk angesprochen wird. Die Daten liegen in der Datenbank und damit auf einem weiteren Server. Wenn nur Daten angezeigt werden sollen, passiert im Middle Tier sehr wenig. Die Daten werden nicht verarbeitet, sondern nur weitergeleitet. Für die Performance und Latenz wäre es viel besser, die Logik auf demselben Server zu halten wie den Web Tier. Die Verteilung ermöglicht es zwar, den Middle Tier einzeln zu skalieren, aber wenn er sowieso kaum etwas zu tun hat, wird das System dadurch nicht schneller.

Abb. 6–2 Verteilte Drei-Schicht-Architektur



Die Situation bei Microservices ist anders, weil die UI im Microservice enthalten ist. Aufrufe zwischen den Microservices finden nur dann statt, wenn die Microservices Funktionalitäten anderer Microservices benötigen. Wenn das oft passiert, kann das ein Hinweis auf ein Problem in der Architektur sein, denn die Microservices sollen weitgehend unabhängig sein.

In der Praxis funktionieren Microservices-Architekturen trotz der Herausforderungen bezüglich der Verteilung [3]. Dennoch sollten Microservices nicht zu viel miteinander kommunizieren, um die Performance zu verbessern und die Latenz zu reduzieren.

Der große Vorteil einer Microservices-Architektur ist die Möglichkeit des unabhängigen Deployments der einzelnen Services. Durch Code-Abhängigkeiten kann dieses Ziel zunichte gemacht werden. Wenn eine Library in mehreren Microservices genutzt wird und eine neue Version dieser Library ausgerollt werden soll, dann erzwingt das ein gemeinsames Deployment mehrerer Microservices – und das soll gerade vermieden werden. So etwas kann beispielsweise leicht durch Binärabhängigkeiten entstehen, bei denen manchmal unterschiedliche Versionen nicht mehr miteinander kompatibel sind. Das Deployment muss zeitlich so koordiniert werden, dass die Microservices alle in einem bestimmten Zeitrahmen und in einer bestimmten Reihenfolge ausgerollt werden. Außerdem muss die Code-Abhängigkeit in allen Microservices geändert werden, was ebenfalls mit den Teams koordiniert und priorisiert werden muss. Eine Abhängigkeit auf binärer Ebene ist eine sehr enge technische Kopplung und zieht auch eine enge organisatorische Kopplung nach sich.

Microservices propagieren daher einen Shared-Nothing-Ansatz, bei dem die Microservices keinen gemeinsamen Code haben. Microservices nehmen so Code-Redundanz bewusst in Kauf und widerstehen dem Anreiz der Wiederverwendung von Code, um eine enge organisatorischen Kopplung zu vermeiden.

Code-Abhängigkeiten können in bestimmten Situationen tolerierbar sein. Wenn ein Microservice beispielsweise eine Client Library anbietet, mit der Aufrufer bei der Nutzung dieses Microservice unterstützt werden, muss das nicht unbedingt negative Folgen haben. Die Library hängt von der Schnittstelle des Microservice ab. Wenn die Schnittstelle abwärtskompatibel geändert wird, kann auch ein Aufrufer mit einer alten Version der Client Library noch den Microservice nutzen. Das Deployment ist nach wie vor entkoppelt. Aber die Client Library kann ein Einstieg in eine Code-Abhängigkeit sein. Wenn die Client Library beispielsweise Domänen-Objekte enthält, kann das schon ein Problem sein. Sollte die Client Library denselben Code für die Domänen-Objekte enthalten, wie er auch intern genutzt wird, schlägt eine Änderung an den internen Objekten auf die Clients durch. Sie müssen dann gegebenenfalls neu deployt werden. Wenn in dem Domänen-Objekt Logik enthalten ist, kann diese Logik nur geändert werden, wenn alle Clients ebenfalls neu deployt werden – auch das verstößt gegen die Idee, dass Microservices unabhängig voneinander deployt werden können.

Auswirkungen von Code-Abhängigkeiten

Ein Beispiel für die Auswirkung von Code-Abhängigkeiten: Authentifizierung von Benutzern ist eine zentrale Funktion, die alle Services nutzen. Ein Projekt hat dazu einen Service entwickelt, der die Authentifizierung implementiert. Mittlerweile gibt es Open-Source-Projekte, die so etwas umsetzen ([Abschnitt 8.12](#)), sodass eine eigene Implementierung kaum noch sinnvoll ist. In dem Projekt konnte jeder Microservice eine Bibliothek verwenden, die den Umgang mit dem Authentifizierungsservice vereinfacht. Daher haben alle Microservices eine Code-Abhängigkeit zum

Authentifizierungsservice. Wenn Änderungen am Authentifizierungsservice notwendig sind, kann es sein, dass die Bibliothek neu ausgerollt werden muss. Das bedeutet, dass auch alle Microservices geändert und neu ausgerollt werden müssen. Dazu kommt die Koordination der Deployments der Microservices und des Authentifizierungsservice. Das kann leicht eine zweistellige Anzahl an Personentagen kosten. Damit ist die Authentifizierung kaum noch änderbar. Der Grund ist die Code-Abhängigkeit. Könnte der Authentifizierungsservices einfach so ausgeliefert werden und gäbe es keine Code-Abhängigkeiten, die das Deployment der Microservices und des Authentifizierungsservice miteinander verbinden, wäre das Problem gelöst.

Kommunikation zwischen Microservices verläuft über Unzuverlässige Kommunikation das Netzwerk und ist daher unzuverlässig. Außerdem können Microservices ausfallen. Um zu verhindern, dass dann das gesamte System ausfällt, müssen die anderen Microservices in so einem Fall den Ausfall kompensieren und weiter zur Verfügung stehen. Dazu muss dann allerdings die Qualität des Services eingeschränkt werden. ([Abschnitt 10.5](#))

Technisch kann dieses Problem nicht vollständig gelöst werden: Die Verfügbarkeit der Microservices kann z. B. durch hochverfügbare Hardware optimiert werden. Aber das treibt die Kosten in die Höhe. Und es ist auch keine vollständige Lösung: In gewisser Weise erhöht es sogar das Risiko. Wenn der Microservice doch ausfällt und sich der Ausfall über das gesamte System fortpflanzt, ist das Ergebnis ein kompletter Ausfall des Systems. Die Microservices selbst sollten daher eher den Ausfall kompensieren.

Dabei wird die Schwelle zwischen einem technischen Problem und einem fachlichen Problem überschritten. Als Beispiel dafür kann ein Geldautomat fungieren: Wenn der Geldautomat den Kontostand des Kunden nicht mehr abfragen kann, gibt es zwei Möglichkeiten. Der Geldautomat kann die Auszahlung verweigern. Das ist zwar sicher, verärgert aber den Kunden und kostet Umsatz. Oder der Geldautomat kann das Geld auszahlen – gegebenenfalls bis zu einer bestimmten Obergrenze. Welche der Alternativen umgesetzt wird, ist eine fachliche Entscheidung. Letztendlich geht es darum, ob man lieber auf der sicheren Seite stehen will oder auf Umsatz verzichtet und Kunden verärgert.

Die technologische Freiheit der Microservices kann Technologie-Pluralismus dazu führen, dass ein Projekt viele unterschiedliche Technologien verwendet. Die Microservices müssen keine gemeinsame technologische Basis haben. Dadurch steigt die Komplexität des Gesamtsystems. Jedes Team beherrscht noch die Technologien, die im eigenen Microservice genutzt werden. Aber das Gesamtsystem kann durch die Anzahl der genutzten Technologien und Ansätze eine Komplexität erreichen, die kein einzelner Entwickler und kein einzelnes Team mehr versteht. Das ist aber meist gar nicht notwendig, weil jedes Team nur seine Microservices verstehen muss. Wenn aber ein Verständnis des Gesamtsystems notwendig ist, und sei es nur aus einer bestimmten Perspektive – zum Beispiel aus der Perspektive des Betriebs –, ist das ein Problem. Dann kann eine Vereinheitlichung eine sinnvolle Gegenmaßnahme sein. Das bedeutet nicht, dass der Technologie-Stack vollständig einheitlich sein muss, sondern nur, dass bestimmte Teile vereinheitlicht werden oder die einzelnen Microservices sich einheitlich verhalten müssen. Es kann zum Beispiel ein einheitliches Logging-Framework definiert werden. Oder ein einheitliches Format für Logging, das

dann unterschiedliche Logging-Frameworks unterschiedlich implementieren. Oder eine gemeinsame technischen Basis wie die JVM (Java Virtual Machine) kann aus Betriebsaspekten heraus vorgeschrieben sein, ohne die Programmiersprache festzulegen.

6.2 Architektur

Die Architektur eines Microservice-Systems teilt die fachlichen Funktionalitäten unter den Microservices auf. Um die Architektur auf dieser Ebene zu verstehen, müssen Abhängigkeiten und Kommunikationsbeziehungen zwischen den Microservices bekannt sein. Die Analyse dieser Kommunikationsbeziehungen ist schwierig. Bei einem großen Deployment-Monolithen gibt es Werkzeuge, die den Quellcode oder gar nur das ausführbare System einlesen. Daraus können die Werkzeuge Schaubilder erstellen, die Module und ihre Beziehungen visualisieren. So ist es möglich, die im System umgesetzte Architektur zu überprüfen, gegen die geplante Architektur abzugleichen und die Entwicklung der Architektur über die Zeit zu verfolgen. Solch ein Werkzeug und so ein Überblick sind für die Arbeit an der Architektur zentral und bei Microservices schwieriger zu erstellen, da solche Werkzeuge fehlen – es gibt aber Lösungen. [Abschnitt 8.2](#) betrachtet dies im Detail.

Microservices basieren auf der Idee, dass die Organisation und die Architektur dasselbe sind. Diesen Umstand machen Microservices sich für die Umsetzung der Architektur zunutze. Die Organisation wird so aufgestellt, dass sie die Umsetzung der Architektur besonders einfach macht. Das bedeutet aber, dass eine Umstellung der Architektur eine Änderung der Organisation nach sich ziehen kann. Dadurch wird eine Änderung der Architektur schwieriger. Das ist eigentlich kein Problem nur von Microservices: Das Gesetz von Conway ([Abschnitt 4.2](#)) gilt für alle Projekte gleichermaßen. Andere Projekte sind sich dem Gesetz oft nicht bewusst. Sie nutzen daher das Gesetz nicht produktiv und können die organisatorischen Probleme bei einer Änderung der Architektur nicht abschätzen.

Die Architektur beeinflusst auch die unabhängige Entwicklung der einzelnen Microservices und die unabhängigen Ströme von Stories. Wenn die fachliche Aufteilung der Microservices nicht stimmt, kann es passieren, dass Anforderungen nicht nur ein Team und einen Microservice beeinflussen, sondern mehrere. Dann ist mehr Koordination zwischen den Teams und den Microservices notwendig, was die Produktivität negativ beeinflusst und damit einen der wesentlichen Gründe für die Einführung von Microservices zunichte macht.

Die Architektur beeinflusst bei Microservices nicht nur die Qualität der Software, sondern auch die Organisation und die unabhängige Arbeit der Teams und damit die Produktivität. Architektur wird noch wichtiger, weil Fehler weiter reichende Konsequenzen haben.

Fachlicher Architektur wird aber in vielen Projekten bei Weitem keine ausreichende Aufmerksamkeit geschenkt und oft auch viel weniger Aufmerksamkeit als der technischen Architektur. Außerdem sind die meisten Architekten mit fachlicher Architektur nicht so erfahren wie mit technischer Architektur. Ein solches Vorgehen kann

bei Microservices zu erheblichen Problemen führen. Die Aufteilung in Microservices und damit in Zuständigkeitsbereiche der Teams muss nämlich nach fachlichen Gesichtspunkten erfolgen.

In einem Microservice ist ein Refactoring einfach, weil der Microservice klein ist. Er kann auch leicht ersetzt und neu implementiert werden.

Refactoring

Zwischen Microservices sieht die Situation anders aus: Die Verlagerung von Funktionalitäten von einem Microservice zu einem anderen ist schwierig. Die Funktionalität muss in eine andere Deployment-Einheit verschoben werden. Das ist schon schwieriger, als die Funktionalität in derselben Einheit zu verschieben. Zwischen den Microservices sind die Technologien nicht einheitlich. Die Microservices können andere Bibliotheken nutzen oder gar andere Programmiersprachen. Dann muss die Funktionalität in einen neuen Microservice ausgelagert werden. Die Funktionalität kann auch in der Technologie des anderen Microservice neu implementiert und in den Microservice verschoben werden. Auch das ist weit komplexer als das einfache Verschieben von Code innerhalb eines Microservice.

Microservices helfen dabei, möglichst viele Änderungen in kurzer Zeit in Produktion zu bringen und eine nachhaltige Entwicklungsgeschwindigkeit zu erreichen. Das ist vor allem nützlich, wenn es sehr viele und schwer vorhersehbare Anforderungen gibt. Genau in diesem Bereich sind Microservices eigentlich zu Hause. Änderungen eines Microservice sind auch sehr einfach. Aber die Anpassung der Architektur des Gesamtsystems beispielsweise durch das Verschieben von Funktionalitäten ist nicht so einfach.

Dazu kommt, dass die Architektur eines Systems meistens beim ersten Wurf noch nicht optimal ist. Bei der Umsetzung lernt das Team sehr viel über die Domäne. In einem zweiten Wurf kann es sicher eine bessere Architektur entwerfen. Die meisten Projekte, die an einer schlechten Architektur leiden, hatten am Anfang eine gute Architektur, wenn man den damaligen Wissensstand betrachtet. Erst im Verlaufe des Projekts stellt sich heraus, dass Anforderungen doch anders gemeint sind, und es gibt neue Anforderungen, sodass die Architektur nicht mehr passt. Ein Problem ergibt sich erst, wenn daraus keine Konsequenzen gezogen werden. Dann macht das Projekt mit einer unangemessenen Architektur einfach weiter und irgendwann ist die Architektur völlig unpassend. Die Lösung ist, die Architektur schrittweise anhand des jeweiligen Wissensstandes zu verbessern. Die Änderbarkeit der Architektur und die Anpassung der Architektur an neue Anforderungen sind dafür zentral. Aber bei der Änderbarkeit der Architektur auf Ebene des Gesamtsystems haben Microservices Schwächen, während innerhalb von Microservices Änderungen sehr einfach sind.

Die Architektur ist bei Microservices noch wichtiger als bei anderen Systemen, weil sie auch die Organisation und das unabhängige Arbeiten an Anforderungen beeinflusst. Gleichzeitig haben Microservices vor allem Vorteile in Bereichen, bei denen Anforderungen unklar sind und die Architektur daher änderbar sein muss. Leider ist das Zusammenspiel der Microservices schwer veränderlich, weil die Aufteilung in Microservices durch die verteilte Kommunikation zwischen den Microservices recht fest ist und die Microservices

Zusammenfassung

in unterschiedlichen Technologien umgesetzt sein können, was das Verschieben von Funktionalitäten erschwert. Änderungen an den einzelnen Microservices oder das Ersetzen von einzelnen Microservices sind hingegen sehr einfach.

6.3 Infrastruktur und Betrieb

Microservices sollen unabhängig voneinander in Produktion gebracht werden und einen eigenen Technologie-Stack nutzen können. Daher ist jeder Microservice meistens auf einem eigenen Server beheimatet. Nur so kann eine vollständige technologische Unabhängigkeit gewährleistet werden. Die dafür notwendige Vielzahl an Systemen wäre mit Hardware-Servern nicht zu bewältigen. Aber auch mit Virtualisierung ist das Management einer solchen Umgebung nicht einfach. Die Anzahl der benötigten virtuellen Maschinen kann über dem liegen, was sonst eine ganze Unternehmens-IT nutzt. Wenn es Hunderte von Microservices gibt, gibt es auch Hunderte von virtuellen Maschinen und einige mehrfach zum Beispiel für Lastverteilung. Das setzt eine Automatisierung voraus und passende Infrastrukturen, die eine große Anzahl virtueller Maschinen erzeugen können.

Über den Betrieb hinaus benötigt jeder Microservice Continuous-Delivery-Pipelines noch mehr Infrastruktur: Er muss eine eigene Continuous-Delivery-Pipeline haben, damit er unabhängig von den anderen Microservices in Produktion gebracht werden kann. Das bedeutet, dass es entsprechende Testumgebungen und Automatisierungsskripte geben muss. Die große Anzahl Pipelines führt zu weiteren Herausforderungen: Sie müssen aufgebaut und gewartet werden. Dazu müssen die Pipelines weitgehend vereinheitlicht werden, um die Aufwände zu reduzieren.

Jeder der Microservices muss außerdem ein Monitoring Monitoring haben. Nur so können Probleme in dem Service erkannt werden. Bei einem Deployment-Monolithen ist es noch recht einfach, die Systeme zu überwachen: Bei einem Problem kann der Administrator sich auf dem System einloggen und Probleme durch entsprechende Werkzeuge untersuchen. Bei einem Microservice-System sind es so viele Systeme, dass dieser Ansatz nicht mehr möglich ist. Also muss es ein Monitoring geben, das alle Systeme umfasst. Dabei sollten nicht nur die typischen Informationen aus dem Betriebssystem und dem I/O zur Festplatte und zum Netzwerk ausgewertet werden, sondern auch eine Sicht in die Anwendung anhand von Anwendungsmetriken möglich sein. Nur so können die Entwickler herausfinden, wo die Anwendung optimiert werden muss oder aktuell Probleme existieren.

Schließlich muss jeder Microservice unabhängig von den anderen Microservices in der Versionskontrolle abgelegt werden. Nur Software, die einzeln versioniert wird, kann getrennt in Produktion gebracht werden. Werden zwei Software-Module nämlich gemeinsam versioniert, sollten sie immer gemeinsam in Produktion gebracht werden. Sonst kann es geschehen, dass eine Änderung sich doch auf beide Module ausgewirkt hat und eigentlich beide Services neu ausgeliefert werden müssen. Ebenso ist bei einer alten Version eines der Services in Produktion nicht klar, ob ein Update notwendig ist oder die neue Version keine Änderungen hat – schließlich könnte die neue Version nur Änderungen im anderen

Microservice gehabt haben.

Für Deployment-Monolithen müsste es weniger Server, Umgebungen und Projekte in der Versionskontrolle geben. Das verringert die Komplexität. Die Anforderungen an den Betrieb und die Infrastruktur sind in einer Microservices-Umgebung viel höher. Mit dieser Komplexität umzugehen ist die größte Herausforderung bei der Adaption von Microservices.

6.4 Fazit

Dieses Kapitel hat verschiedene Herausforderungen bei Microservices aufgezeigt. Auf der technischen Seite ([Abschnitt 6.1](#)) sind die Herausforderungen vor allem eine Folge davon, dass Microservices verteilte Systeme sind: Dadurch werden Performance und Zuverlässigkeit des Systems schwieriger umsetzbar. Ebenso wird die technische Komplexität durch die größere Anzahl Technologien höher und Code-Abhängigkeiten können das unabhängige Deployment von Microservices unmöglich machen.

Die Architektur eines Microservice-Systems ([Abschnitt 6.2](#)) ist wegen der Auswirkungen auf die Organisation und der parallelen Abarbeitung von mehreren Stories sehr wichtig. Gleichzeitig sind Änderungen am Zusammenspiel der Microservices schwieriger. Funktionalitäten können nicht so einfach von einem Microservice zu einem anderen verschoben werden. In einem Projekt können Klassen meist sogar automatisch verschoben werden. Zwischen Microservices ist manuelle Arbeit notwendig. Die Schnittstelle zum Code ändert sich von lokalen Aufrufen hin zu Kommunikation zwischen Microservices. Das erhöht den Aufwand. Schließlich können Microservices in unterschiedlichen Sprachen geschrieben sein – dann ist das Verschieben des Codes im Prinzip ein Neuschreiben.

Aber Änderungen an der Architektur des Gesamtsystems sind wegen unklarer Anforderungen notwendig. Das Team lernt außerdem ständig mehr über das System. Gerade da, wo Microservices wegen des schnellen und unabhängigen Deployments besonders vorteilhaft sind, müsste die Architektur besonders einfach änderbar sein. In Microservices sind die Änderungen auch einfach, aber zwischen Microservices sehr aufwendig.

Schließlich wird durch die größere Anzahl an Services die Infrastruktur komplexer ([Abschnitt 6.3](#)), weil mehr Server, mehr Projekte in der Versionskontrolle und mehr Continuous-Delivery-Pipelines benötigt werden. Das ist eine wesentliche Herausforderung der Microservices-Architekturen.

[Teil III](#) des Buchs wird Lösungen für diese Herausforderungen zeigen.

Wesentliche Punkte

- Microservices sind verteilte Systeme. Sie sind daher technisch komplexer.
- Die Architektur ist wegen der Auswirkungen auf die Organisation sehr wichtig. Gleichzeitig ist sie zwar innerhalb von Microservices einfach änderbar, aber das Zusammenspiel der Microservices ist nur schwer änderbar.

- Wegen der Anzahl ist mehr Infrastruktur wie Serverumgebungen, Continuous-Delivery-Pipelines oder Projekte in der Versionskontrolle notwendig.

Selber ausprobieren und experimentieren

Wähle eines der Szenarien aus dem [Kapitel 3](#) oder ein dir bekanntes Projekt.

- Mit welchen Herausforderungen muss man rechnen? Bewerte die Herausforderungen entsprechend. Das Fazit stellt die Herausforderungen noch einmal komprimiert zusammen.
- Welche der Herausforderungen birgt das größte Risiko in sich? Warum?
- Gibt es Möglichkeiten, Microservices so zu nutzen, dass die Vorteile maximiert werden und die Herausforderungen vermieden? Beispielsweise könnte man darauf verzichten, heterogene Technologie-Stacks zu nutzen.

6.5 Links & Literatur

- [1] <http://martinfowler.com/bliki/FirstLaw.html>
- [2] <https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>
- [3] <http://martinfowler.com/articles/distributed-objects-microservices.html>

7 Microservices und SOA

Auf den ersten Blick scheinen Microservices und SOA viel gemeinsam zu haben: Im Fokus beider Ansätze steht die Aufteilung von großen Systemen in Services. Sind SOA und Microservices dasselbe oder gibt es Unterschiede? Die Antwort auf diese Frage trägt zu einem tieferen Verständnis von Microservices bei. Außerdem ist ein Teil der Ideen aus dem SOA-Bereich für Microservices-Architekturen interessant. Ein SOA-Ansatz kann vor allem bei der Migration zu Microservices einen Vorteil haben. Er teilt die alten Anwendungen in Services auf, die Microservices dann ersetzen oder ergänzen können.

[Abschnitt 7.1](#) definiert den Begriff SOA und den Service-Begriff im SOA-Kontext. Darauf baut [Abschnitt 7.2](#) auf und zeigt die Unterschiede zwischen SOA und Microservices.

7.1 Was ist SOA?

SOA (Service-Oriented Architecture) hat eines mit Microservices gemeinsam: Es ist ein Begriff, der nicht eindeutig definiert ist. Dieser Abschnitt stellt daher nur eine mögliche Definition dar. Andere Definitionen sehen durchaus Microservices und SOA als identische Ansätze. Schließlich basieren beide auf Services und die Aufteilung von Anwendungen in Services.

Zentral ist für SOA der Begriff des Services [1]. Laut Wikipedia soll ein SOA-Service die folgenden Eigenschaften haben:

- Ein Service soll eine fachliche Funktionalität umfassen.
- Der Service kann eigenständig genutzt werden.
- Er ist im Netzwerk verfügbar.
- Jeder Service hat eine Schnittstelle. Kenntnisse über die Schnittstelle reichen zur Nutzung des Service aus.
- Die Nutzung des Service ist von verschiedenen Programmiersprachen und Plattformen aus möglich.
- Um den Service einfach zu nutzen, ist er in einem Verzeichnis registriert. Darüber suchen Clients den Service zur Laufzeit und nutzt ihn.
- Der Service sollte grobgranular sein, um die Abhängigkeiten zu reduzieren. Kleine Services können nur zusammen mit anderen Services sinnvolle Fachlichkeiten abbilden. Daher fokussiert SOA eher auf größere Services.

Die SOA-Services werden nicht neu implementiert, sondern sind schon in den Anwendungen im Unternehmen vorhanden. Eine SOA-Einführung macht diese Services außerhalb der Anwendungen verfügbar. Durch die Aufteilung der Anwendungen in Services wird deren Nutzung in verschiedenen Zusammenhängen ermöglicht. Das soll die Flexibilität der IT insgesamt verbessern – das ist das Ziel einer SOA. Auf Basis der

einzelnen Services ist es möglich, Geschäftsprozesse abzubilden und dabei die Services wiederzuverwenden. Dazu ist nur eine Orchestrierung der einzelnen Services notwendig.

Abb. 7–1 Überblick über eine SOA-Landschaft

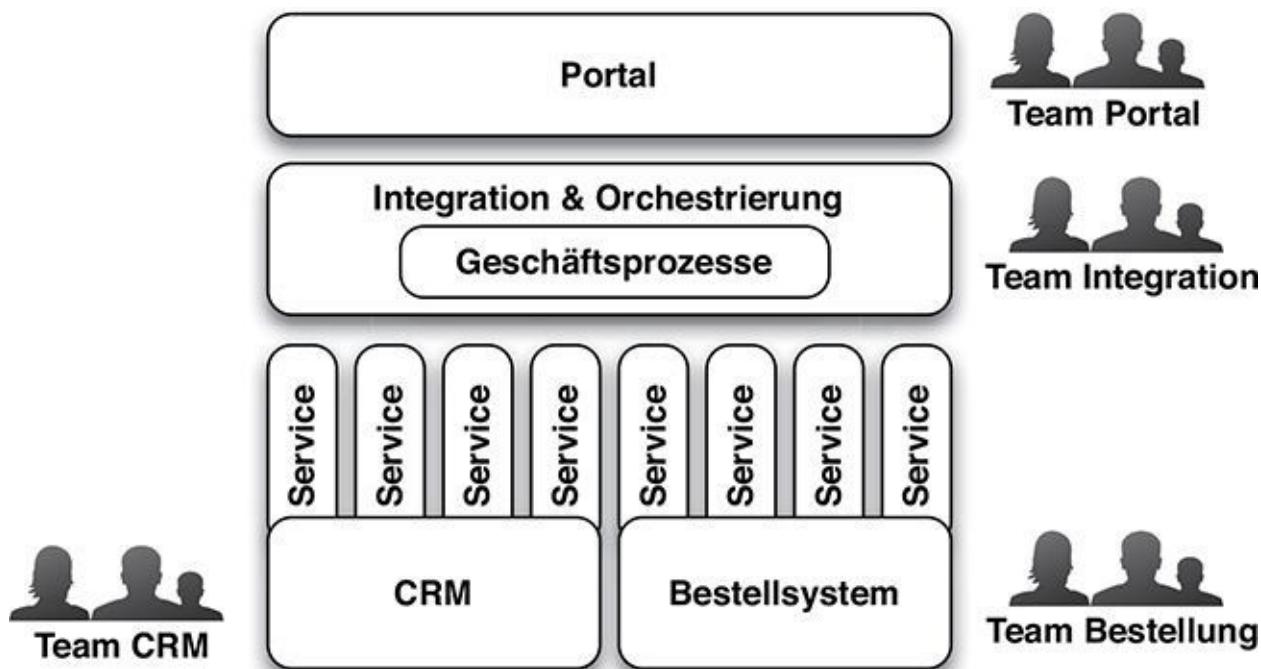


Abbildung 7–1 zeigt eine mögliche SOA-Landschaft. Wie die vorhergehenden Beispiele kommt es aus dem E-Commerce-Umfeld. In der SOA-Landschaft gibt es verschiedene Systeme:

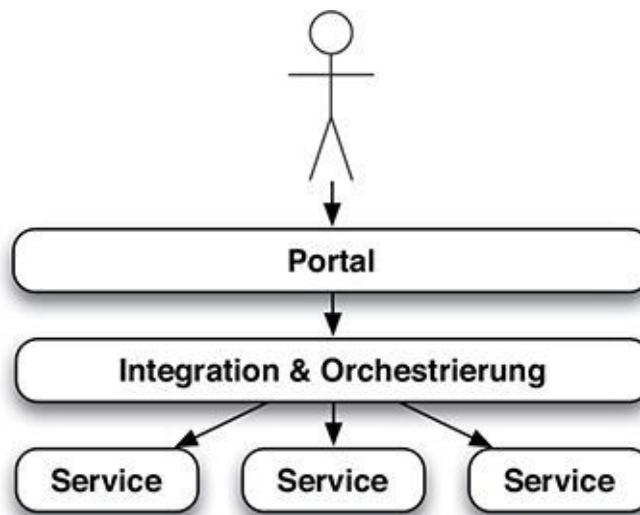
- Das *CRM* (Customer Relationship Management) ist eine Anwendung, die wesentliche Informationen zu den Kunden speichert. Dazu zählen nicht nur die Kontaktdaten, sondern auch die Historie aller Interaktionen mit dem Kunden – Anrufe genauso wie E-Mails oder Bestellungen. Das CRM exponiert Services, die beispielsweise das Anlegen eines neuen Kunden unterstützen, Informationen zu einem Kunden anbieten oder Reports über alle Kunden generieren.
- Das *Bestellungssystem* ist für die Abarbeitung der Bestellungen zuständig. Es kann neue Bestellungen entgegennehmen, Auskunft über den Zustand einer Bestellung geben oder eine Bestellung stornieren. Auch dieses System gibt Zugriff auf die verschiedenen Funktionalitäten durch einzelne Services. Diese Services können durchaus erst später als zusätzliche Schnittstelle in das System integriert worden sein.
- In der Abbildung sind das CRM und das Bestellsystem die einzigen Systeme. In Wirklichkeit gäbe es sicher noch weitere Systeme, beispielsweise um den Produktkatalog zur Verfügung zu stellen. Als Illustration der SOA-Landschaft reichen jedoch die beiden Systeme.
- Damit die Systeme sich gegenseitig aufrufen können, gibt es eine *Integrationsplattform*. Diese Plattform ermöglicht die Kommunikation der Services untereinander. Die Plattform kann die Services mithilfe der Orchestrierung neu komponieren. Die Orchestrierung kann durch eine Technologie erfolgen, die Geschäftsprozesse modelliert und zur Ausführung der Prozesse die einzelnen Services aufruft.

- Daher ist die *Orchestrierung* dafür zuständig, die einzelnen Services zu koordinieren. Die Infrastruktur ist intelligent und kann je nach Nachricht geeignet reagieren. Sie enthält die Modellierung der Geschäftsprozesse und damit einen wichtigen Teil der Geschäftslogik.
- Die Nutzung der SOA ist über ein *Portal* möglich. Es ist dafür zuständig, Nutzern eine Oberfläche zur Verfügung zu stellen, mit der die Services verwendet werden können. Es kann mehrere Portale geben: beispielsweise eines für Endkunden, eines für den Support und eines für interne Mitarbeiter. Es ist auch möglich, das System über Rich-Client-Anwendungen oder mobile Apps anzusprechen. Von der Architektur her ist das aber kein Unterschied: Auch solche Systeme nutzen die verschiedenen Services, um sie für einen Nutzer verwendbar zu machen. Letztendlich sind alle diese Systeme jeweils eine universelle UI, um alle anderen Systeme nutzen zu können.

Jedes dieser Systeme kann von einem eigenen Team betrieben und weiterentwickelt werden. Im Beispiel könnte es je ein Team für das CRM und das Bestellsystem geben, je ein weiteres Team für jedes Portal und schließlich ein Team, das sich um die Integration und Orchestrierung kümmert.

[Abbildung 7–2](#) zeigt, wie die Kommunikation in einer SOA-Architektur strukturiert ist. Nutzer arbeiten mit der SOA typischerweise über das Portal. Dadurch können Geschäftsprozesse angestoßen werden, die dann in der Orchestrierung umgesetzt sind. Diese Prozesse nutzen die Services. Gerade bei der Migration von Monolithen zu einer SOA ist es auch möglich, dass Nutzer direkt einen Monolithen mit der eigenen Benutzeroberfläche verwenden. Aber die Zielarchitektur sieht ein Portal als zentrale Benutzerschnittstelle und eine Orchestrierung für die Umsetzung der Prozesse vor.

Abb. 7–2 Kommunikation in einer SOA-Architektur



Die Einführung einer SOA ist eine strategische Initiative, SOA einführen an der verschiedene Teams beteiligt werden müssen.

Letztendlich ist das Ziel, die gesamte Unternehmens-IT in einzelne Service aufzuteilen. Die Aufteilung unterstützt das Zusammenstellen von Services zu neuen Funktionalitäten besser. Das ist aber nur möglich, wenn alle Systeme in der gesamten Organisation angepasst worden sind. Und erst, wenn so viele Services zur Verfügung stehen, dass Geschäftsprozesse durch einfache Orchestrierung umgesetzt werden können, kann die

SOA ihre Vorteile ausspielen. Daher muss die verwendete Integrations- und Orchestrierungstechnologie in der gesamten IT verwendet werden, um die Kommunikation und Integration der Services zu ermöglichen. Das führt zu hohen Investitionskosten, da die gesamte IT-Landschaft geändert werden muss. Genau das ist auch ein Hauptkritikpunkt an SOA [4].

Die Services können auch im Internet oder über private Netzwerke anderen Unternehmen und Nutzern angeboten werden. Dann unterstützt die SOA ein Unternehmenskonzept, das auf Outsourcing von Services basiert oder externe Services einbezieht. In einer E-Commerce-Anwendung könnte beispielsweise ein externer Anbieter einfache Services wie die Adressvalidierung anbieten oder komplexe Services wie eine Bonitätsprüfung.

Zumindest bei einer Einführung einer SOA auf Basis alter Systeme sind die Services einer SOA nur Schnittstellen von großen Deployment-Monolithen. Ein Monolith bietet jeweils mehrere Services an. Die Services bauen auf den vorhandenen Anwendungen auf. Oft sind für das Anbieten der Services nicht einmal Anpassungen an den Interna des Systems notwendig. Ein solcher Service hat typischerweise keine UI, sondern bietet nur eine Schnittstelle für andere Anwendungen. Eine UI gibt es nur über alle Systeme übergreifend und auch nicht als Bestandteile eines Service, sondern eigenständig beispielsweise im Portal.

Es ist ebenfalls möglich, in einer SOA kleinere Deployment-Einheiten umzusetzen. Die Definition des SOA-Service sagt nichts über die Größe der Deployment-Einheiten aus – im Gegensatz zu Microservices, bei denen die Größe der Deployment-Einheiten zentral ist.

Die Versionierung der Services in einer SOA stellt eine besondere Herausforderung dar. Eine Änderung an einem Service muss mit den Nutzern des Service abgesprochen werden. Wegen dieser Kommunikation sind Schnittstellenänderungen an Services aufwendig. Die Nutzer des Service werden kaum ihre eigene Software anpassen, wenn sie nicht Vorteile aus der neuen Schnittstelle ziehen. Daher müssen meistens auch alte Versionen der Schnittstelle unterstützt werden. Wenn es viele Nutzer eines Service gibt, kann es sein, dass sehr viele Versionen der Schnittstelle unterstützt werden müssen. Das erhöht die Komplexität der Software und macht Änderungen schwieriger. Schließlich muss bei einem neuen Release der Software sichergestellt sein, dass auch die alten Schnittstellen noch funktionieren. Werden Daten ergänzt, gibt es Herausforderungen, weil die alten Schnittstellen diese Daten nicht unterstützen. Beim Lesen ist das kein großes Problem. Beim Schreiben kann es aber schwierig sein, ohne die zusätzlichen Daten neue Datensätze anzulegen.

Wenn gar externe Nutzer außerhalb der eigenen Firma den Service verwenden, sind Schnittstellenänderungen noch schwieriger. Im Extremfall weiß der Anbieter des Service nicht einmal genau, wer den Service nutzt, weil er anonymen Nutzern im Internet zur Verfügung steht. Eine Abstimmung von Änderungen ist dann nahezu unmöglich. Das Abschalten einer alten Version eines Service wird dadurch fast unmöglich. Und so entstehen immer mehr Versionen der Schnittstellen und Änderungen an den Services werden immer schwieriger. Dieses Problem betrifft Microservices aber auch (siehe [Abschnitt 9.6](#)).

Für die Nutzer einer Schnittstelle ergeben sich auch Herausforderungen: Wenn sie eine Änderung an einer Schnittstelle benötigen, müssen sie das mit dem Team abstimmen, das den Service anbietet. Dort müssen die Änderungen gegen alle anderen Anforderungen und die Wünsche anderer Teams priorisiert werden. Und wie schon dargestellt, ist die Änderung einer Schnittstelle kein einfaches Unterfangen. Daher kann es länger dauern, die Änderungen tatsächlich umzusetzen. Das behindert die Fortentwicklung des Systems.

Nach einer Änderung einer Schnittstelle muss das Deployment der Services koordiniert werden: Zunächst muss der Service deployt werden, der die neue Schnittstellenversion anbietet. Erst danach kann der Service deployt werden, der die neue Schnittstelle benutzt. Da die Anwendungen bei SOA meist Deployment-Monolithen sind, können immer nur mehrere Services gemeinsam deployt werden. Die Koordination der Services wird dadurch erschwert. Außerdem steigt das Risiko, weil das Deployment eines Monolithen lange dauert und nur schwer rückgängig gemacht werden kann – einfach weil die Änderungen so umfangreich sind.

Schnittstellen erzwingen eine Koordination des Deployments.

Die Koordinierung der SOA mithilfe einer Orchestrierung in der Integrationsschicht führt zu einigen Herausforderungen. In gewisser Weise entsteht ein Monolith: Alle Geschäftsprozesse finden sich in dieser Orchestrierung wieder. Dieser Monolith ist oft sogar schlimmer als die sonst üblichen Monolithen, weil er alle Systeme in der IT nutzt. Im Extremfall kann es so weit kommen, dass die Services nur die Verwaltung von Daten übernehmen, während jegliche Logik in der Orchestrierung zu finden ist. Dann ist die gesamte SOA eigentlich nichts anderes als ein Monolith, bei dem die gesamte Logik in der Orchestrierung liegt.

Koordinierung und Orchestrierung

Aber auch sonst sind Änderungen in einer SOA nicht unbedingt einfacher: Fachlichkeiten sind in Services in den verschiedenen Systemen und in Geschäftsprozesse in der Orchestrierung aufgetrennt. Wenn eine Änderung an einer Funktionalität auch die Services oder die Oberfläche betrifft, wird es schwierig: Die Änderung des Geschäftsprozesses ist recht einfach, aber die Änderung am Service ist nur durch Schreiben von Code und ein Deployment der Anwendung möglich, die den Service bereitstellt. Die Änderungen am Code und das Deployment können sehr aufwendig sein. Dann geht die Flexibilität der SOA verloren, die durch einfache Orchestrierung der Services entstehen soll. Umgestaltungen der Oberfläche verursachen Änderungen am Portal oder den anderen Oberflächensystemen und benötigen ebenfalls ein neues Deployment.

SOA ist ein Architekturansatz und unabhängig von einer konkreten Technologie. Allerdings muss eine SOA eine gemeinsame Technologie für die Kommunikation der Services untereinander definieren, wie dies Microservices auch tun. Außerdem muss eine konkrete Technologie für die Orchestrierung der Services festgelegt werden. Oft führt die Einführung einer SOA zur Einführung einer komplexen Technologie, um die Integration und Orchestrierung der Services zu ermöglichen. Es gibt spezielle Technologieprodukte, die alle Aspekte einer SOA unterstützen – aber auch entsprechend komplex sind und deren Features meistens nicht ausgenutzt werden.

Technologien

Diese Technologie kann schnell zu einem Flaschenhals werden. Viele Probleme mit

diesen Technologien werden SOA zugerechnet, obwohl SOA auch mit anderen Technologien umsetzbar wäre. Zu den Problemen gehört die Komplexität der Webservices-Protokolle. SOAP alleine ist noch recht einfach, aber zusammen mit den Erweiterungen aus dem WS-* Umfeld entsteht ein komplexer Protokoll-Stack. WS-* ist für Transaktionen, Sicherheit oder andere Erweiterungen notwendig. Komplexe Protokolle erschweren die Interoperabilität – aber Interoperabilität ist eine Voraussetzung für eine SOA.

Eine Aktion auf der Oberfläche muss durch die Orchestrierung und die verschiedenen Services abgearbeitet werden. Das sind jeweils verteilte Aufrufe im Netzwerk mit dem entsprechenden Overhead und entsprechender Latenz. Außerdem läuft diese Kommunikation durch die zentrale Integrations- und Orchestrierungstechnologie, die daher sehr viele Aufrufe bewältigen muss.

7.2 Unterschiede zwischen SOA und Microservices

SOA und Microservices sind verwandt: Beide streben die Aufteilung von Anwendungen in Services an. Die Unterscheidung von SOA und Microservices nur aufgrund dessen, was im Netzwerk passiert, ist nicht einfach. Immerhin tauschen in beiden Architekturansätzen viele Services Informationen über das Netz aus.

Genau wie Microservices können auch SOA auf asynchroner Kommunikation oder synchroner Kommunikation basieren. SOAs können entkoppelt werden, indem lediglich Events wie »neue Bestellung« verschickt werden. Dann kann jeder SOA-Service mit unterschiedlicher Logik auf das Event reagieren. Ein Service kann eine Rechnung schreiben und ein anderer die Lieferung veranlassen. Die Services sind stark entkoppelt, weil sie nur auf Events reagieren, ohne den Auslöser des Events zu kennen. Neue Services können einfach in das System integriert werden, indem sie auch auf solche Events reagieren.

Aber schon auf Ebene der Integration fangen die Unterschiede zwischen SOA und Microservices an: In einer SOA ist die Integrationslösung auch für die Orchestrierung der Services zuständig. Ein Geschäftsprozess wird aus den Services zusammengestellt. In einer Microservices-Architektur besitzt die Integrationslösung keine eigene Intelligenz. Die Microservices sind dafür zuständig, mit den anderen Services zu kommunizieren. SOA versucht durch die Orchestrierung eine zusätzliche Flexibilität bei der Umsetzung von Geschäftsprozessen zu erreichen. Das funktioniert nur, wenn die Services und die Oberfläche stabil sind und nicht häufig geändert werden müssen.

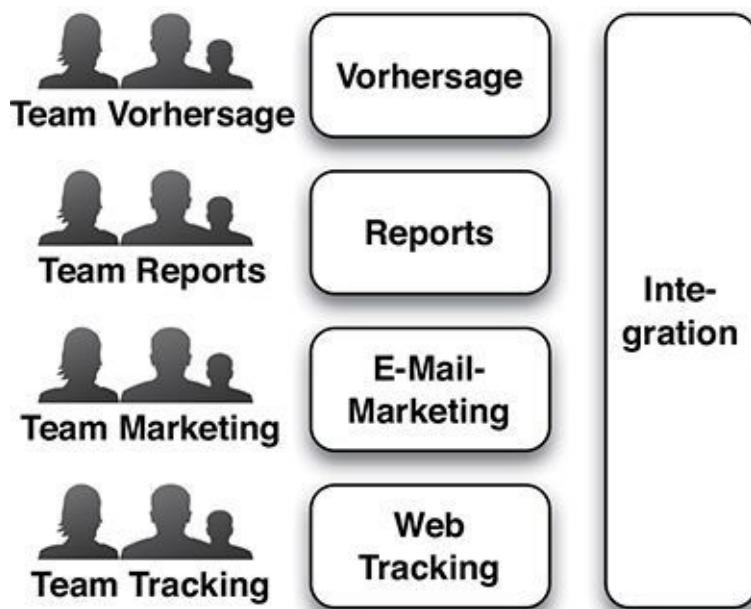
Microservices hingegen setzen darauf, dass jeder Microservice einfach geändert und in Produktion gebracht werden kann und so die notwendige Flexibilität erreicht wird. Wenn die flexiblen Geschäftsprozesse der SOA nicht ausreichend sind, erzwingt eine SOA die Änderung an Services in Deployment-Monolithen oder Oberflächen in einem weiteren Deployment-Monolithen.

Microservices setzen auf Isolation: Idealerweise wird eine Benutzerinteraktion vollständig in einem Microservice abgearbeitet und es ist kein Aufruf eines anderen Microservice notwendig. Dann sind Änderungen für neue Features auf einen Microservice begrenzt. Eine SOA verteilt die Logik auf das Portal, die Orchestrierung und die einzelnen Services.

Aber der wichtigste Unterschied zwischen SOA und Microservices ist die Ebene, auf der die Architekturen ansetzen. Die Architektur einer SOA betrachtet das gesamte Unternehmen. SOA definiert, wie eine Vielzahl von Systemen in der Enterprise-IT interagiert. Microservices hingegen sind eine Architektur für ein einzelnes System. Sie sind eine Alternative zu anderen Modularisierungstechnologien. Es wäre denkbar, ein Microservice-System mit einer anderen Modularisierungstechnologie umzusetzen und dann das System Deployment-Monolith ohne verteilte Services in Produktion zu bringen. Eine vollständige SOA umfasst die gesamte Unternehmens-IT. Sie muss verschiedene Systeme betrachten. Eine Alternative zu einem verteilten Ansatz ist nicht denkbar. Dementsprechend ist die Entscheidung für eine Microservices-Architektur eine Entscheidung, die ein Projekt treffen und auf ein Projekt begrenzt sein kann, während die Einführung und Umsetzung einer SOA das gesamte Unternehmen betreffen.

Microservices: Projektebene

Abb. 7–3 CRM als Sammlung von Microservices



Für das SOA-Szenario aus [Abbildung 7–1](#) ergibt sich eine fundamental andere Architektur (siehe [Abb. 7–3](#)) [2][3]:

- Da sich Microservices nur auf ein einziges System beziehen, muss die Architektur nicht die gesamte IT mit verschiedenen Systemen umfassen, sondern kann auf ein einziges System begrenzt sein. In der Abbildung ist dies das CRM. Dadurch ist das Umsetzen von Microservices recht einfach und wenig kostenintensiv, weil es ausreicht, ein Projekt entsprechend umzusetzen, statt die gesamte IT-Landschaft des Unternehmens zu ändern.
- Dementsprechend erzwingt eine Microservices-Architektur auch keine Integrationstechnologie, die unternehmensweit eingeführt und genutzt wird. Der

Einsatz der Technologie für die Integration und Kommunikation ist auf das Microservice-System begrenzt – oder es können sogar mehrere Ansätze verwendet werden. Beispielsweise kann durch eine Replikation der Daten in der Datenbank ein performanter Zugriff auch auf große Datenmengen umgesetzt werden. Für den Zugriff von anderen Systemen können wiederum andere Technologien verwendet werden. Bei einer SOA müssen alle Services im gesamten Unternehmen mit einer einheitlichen Technologie zugreifbar sein. Dazu ist ein einheitlicher Technologie-Stack notwendig. Microservices setzen auf einfache Technologien, die keinen so komplexen Anforderungen genügen müssen wie SOA-Suites.

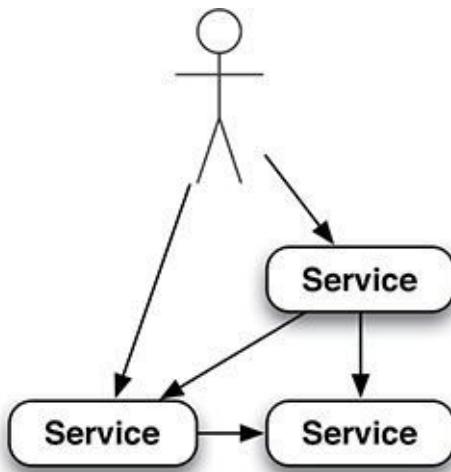
- Auch die Kommunikation zwischen den Microservices ist anders: Microservices setzen auf einfache Kommunikationssysteme ohne Eigenintelligenz. Die Microservices rufen sich gegenseitig auf oder verschicken Nachrichten. Die Integrationstechnologie setzt keine Orchestrierung um. Ein Microservice kann mehrere andere Microservices aufrufen und selber eine Orchestrierung implementieren. Dann liegt die Logik für die Orchestrierung aber in dem Microservice und nicht in einem Integrationslayer. Die Integrationslösung enthält bei Microservices keine Logik, weil sie aus verschiedenen Fachlichkeiten kommen würde. Das widerspricht dem Schnitt nach Fachlichkeiten, den Microservices anstreben.
- Die Nutzung der Integration ist auch ganz anders. Microservices vermeiden durch die integrierte UI und den fachlichen Schnitt die Kommunikation mit anderen Microservices. Eine SOA legt den Fokus auf die Kommunikation. Durch die Orchestrierung erreicht eine SOA ihre Flexibilität – und damit geht eine Kommunikation zwischen den Services einher. Und die Kommunikation bei Microservices muss nicht unbedingt über Messaging oder REST umgesetzt werden: Eine Integration auf der UI-Ebene oder durch Datenreplikation ist genauso machbar.
- Das CRM als Gesamtsystem gibt es in der Microservices-Architektur nicht mehr wirklich. Stattdessen gibt es eine Sammlung von Microservices, die jeweils bestimmte Funktionalitäten abdecken, wie die Reports oder die Umsatz-Vorhersage.
- Während in der SOA alle Funktionalitäten des CRM-Systems in einer einzigen Deployment-Einheit zusammengefasst sind, ist bei dem Microservices-Ansatz jeder Service eine eigene Deployment-Einheit und kann unabhängig von den anderen in Produktion gebracht werden. Abhängig von der konkreten technischen Infrastruktur können die Services sogar noch kleiner sein als die in [Abbildung 7–3](#) dargestellten.
- Schließlich ist der Umgang mit der UI anders: Bei einem Microservice ist die UI Teil des Microservice, während eine SOA typischerweise nur Services anbietet, die dann von einem Portal genutzt werden können.
- Die Trennung in UI und Service bei SOA hat weitreichende Konsequenzen: Um bei einer SOA eine neue Funktionalität einschließlich der UI umzusetzen, muss mindestens der Service geändert und die UI angepasst werden. Das bedeutet, es müssen mindestens zwei Teams koordiniert werden. Wenn andere Services in anderen Anwendungen genutzt werden, sind es noch mehr Teams und der Koordinierungsaufwand ist entsprechend höher. Hinzu kommen noch die

Änderungen in der Orchestrierung, die ebenfalls ein getrenntes Team umsetzt. Microservices hingegen streben an, dass ein Team eine neue Funktionalität mit möglichst wenig Koordinierung mit anderen Teams in Produktion bringen kann. Durch die Microservices-Architektur sind Schnittstellen zwischen Schichten, die sonst zwischen den Teams bestehen, innerhalb eines Teams. Das erleichtert die Umsetzung von Änderungen. Die Änderungen können im selben Team bearbeitet werden. Wäre ein anderes Team involviert, müssten die Änderungen gegenüber anderen Anforderungen priorisiert werden.

- Jeder Microservice kann von einem eigenen Team entwickelt und betrieben werden. Dieses Team ist dann für eine fachliche Aufgabe zuständig und kann neue Anforderungen oder Änderungen an der Fachlichkeit vollständig unabhängig von den anderen Teams umsetzen.
- Auch das Vorgehen bei SOA und Microservices ist anders: SOA führt nur eine neue Schicht über den vorhandenen Services ein, um die Anwendungen neu zu kombinieren. Es zielt auf eine flexible Integration der vorhandenen Anwendungen. Microservices dienen dazu, den Aufbau der Anwendungen selbst zu ändern – mit dem Ziel, dass Änderungen an den Anwendungen einfacher werden.

Die Kommunikationsbeziehungen bei Microservices zeigt [Abbildung 7–4](#). Der Nutzer interagiert mit der UI, die von den verschiedenen Microservices umgesetzt wird. Außerdem sprechen die Microservices miteinander. Eine zentrale UI oder Orchestrierung gibt es nicht.

Abb. 7–4 Kommunikation bei Microservices



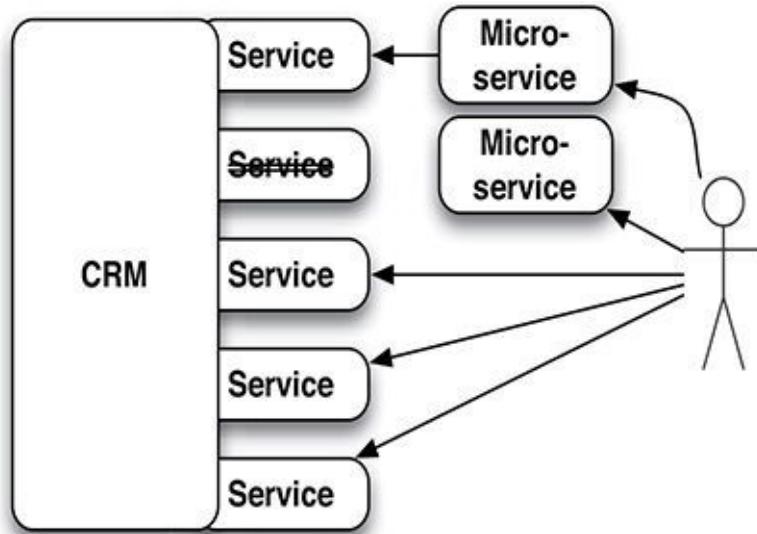
Es gibt durchaus Bereiche, in denen Microservices und SOA Synergien haben. Schließlich verfolgen beide Ansätze das Ziel, Anwendungen in Services aufzulösen. Ein solcher Schritt kann bei der Migration einer Anwendung zu Microservices hilfreich sein: Wenn die Anwendung in SOA-Services aufgeteilt ist, können einzelne Services durch Microservices ersetzt oder ergänzt werden. Bestimmte Anfragen können von einem Microservice bearbeitet werden, während die anderen Anfragen immer noch von der Anwendung bearbeitet werden. Dadurch ist es möglich, Anwendungen schrittweise zu migrieren und die Microservices schrittweise zu implementieren.

Berührungspunkte

[Abbildung 7–5](#) zeigt ein Beispiel: Der oberste Service des CRM wird durch einen

Microservice ergänzt. Er nimmt nun alle Anfragen entgegen und kann gegebenenfalls das CRM aufrufen. Der zweite Service des CRM ist vollständig durch einen Microservice ersetzt. So kann das CRM um neue Funktionalitäten ergänzt werden. Gleichzeitig muss nicht das gesamte CRM neu implementiert werden, sondern es kann an ausgewählten Stellen durch Microservices ergänzt werden. [Abschnitt 8.5](#) stellt weitere Ansätze vor, wie Legacy-Anwendungen durch Microservices abgelöst werden können.

Abb. 7–5 SOA zur Migration von Microservices



7.3 Fazit

SOA und Microservices sind unterschiedliche Herangehensweisen mit einem gemeinsamen Ziel (siehe [Tab. 7–1](#)): die Umsetzung neuer Anforderungen schneller und einfacher zu machen. SOA geht davon aus, dass Services relativ selten geändert werden und die Flexibilität durch die Orchestrierung der Services wesentlich für eine schnellere Umsetzung von Anforderungen ist. Neben der Orchestrierung wird von den Services auch die UI abgetrennt. Microservices hingegen setzen darauf, eine fachliche Aufteilung zu erreichen, bei der Änderungen nur in einem Service vorgenommen werden müssen. Dieser Service muss dann die UI und die Logik enthalten. So isoliert der Microservice eine bestimmte Fachlichkeit in einem Deployment-Artefakt, sodass Änderungen an einer bestimmten Funktionalität nur Änderungen an einem Microservice bedeuten, der dann auch unabhängig von den anderen Microservices deployt werden kann.

Tab. 7–1 Unterschiede zwischen SOA und Microservices

	SOA	Microservices
Anwendungsbereich	Unternehmensweite Architektur	Architektur für ein Projekt
Umsetzung Flexibilität	von Flexibilität durch Orchestrierung	Flexibilität durch schnelles Deployment und schnelle unabhängige Entwicklung von Microservices

Organisation	Services werden von verschiedenen Organisationseinheiten umgesetzt	Services werden von Teams im selben Projekt umgesetzt
Deployment	Monolithisches Deployment mehrerer Services	Jeder Microservice kann einzeln deployt werden
UI	Portal als universelle UI aller Services	Service enthält UI

Auf organisatorischer Ebene sind die Ansätze sehr unterschiedlich: SOA setzt bei der Struktur der gesamten IT des Unternehmens an, Microservices können in einem einzelnen Projekt genutzt werden. SOAs fokussieren auf eine Organisation, bei der einige Teams Backend-Services entwickeln, während die UI in einem eigenen Team umgesetzt wird. Bei einem Microservice-Ansatz soll möglichst alles in einem Team umgesetzt werden, um die Kommunikation zu erleichtern und dadurch Features schneller umsetzen zu können. Das ist in einer SOA kein Ziel. In einer SOA kann ein neues Feature Änderungen an vielen Services benötigen und damit auch Kommunikation zwischen sehr vielen Teams. Microservices versuchen das zu vermeiden.

Auf technischer Ebene sind Gemeinsamkeiten zu erkennen: Beide Konzepte setzen auf Services. Die Granularität der Services kann sogar ähnlich sein. Wegen dieser technischen Gemeinsamkeiten scheint die Abgrenzung nicht so einfach zu sein. Aber konzeptionell, architektonisch und organisatorisch haben die beiden Ansätze sehr unterschiedliche Auswirkungen.

Wesentliche Punkte

- SOA und Microservices teilen Anwendungen in Services auf, die im Netz zur Verfügung stehen. Dazu können ähnliche Technologien genutzt werden.
- SOA zielt auf Flexibilität auf der Ebene der Enterprise-IT durch eine Orchestrierung der Services. Das ist ein komplexes Unterfangen und funktioniert nur, wenn keine Änderungen an den Services notwendig sind.
- Microservices betrachten ein einzelnes Projekt und zielen darauf, das Deployment zu vereinfachen und parallel Arbeit an verschiedenen Services zu ermöglichen.

Selber ausprobieren und experimentieren

In die SOA-Landschaft aus [Abbildung 7–1](#) soll eine neue Funktionalität eingebaut werden. Das CRM hat keine Unterstützung für E-Mail-Kampagnen. Daher muss ein System für E-Mail-Kampagnen implementiert werden. Es soll einen Service für das Anlegen und Ausführen von Kampagnen und einen Service für die Auswertung haben.

Als Architekt sind nun folgende Fragen zu beantworten:

- Wird zur Integration der beiden Services die SOA-Infrastruktur benutzt? Der Service für die Auswertung der Kampagnen braucht sehr viele Daten.
- Würde man für den Zugriff auf die große Datenmenge eher Datenreplikation, Integration auf der UI-Ebene oder Aufrufe der Services nutzen?
- Welche dieser Integrationsmöglichkeiten bietet eine SOA-Infrastruktur typischerweise?
- Soll der Service sich in das vorhandene Portal integrieren oder bekommt er eine eigene Oberfläche? Was spricht für die eine oder andere Lösung?
- Soll die neue Funktionalität vom CRM-Team umgesetzt werden?

7.4 Links & Literatur

- [1] https://de.wikipedia.org/wiki/Serviceorientierte_Architektur
- [2] https://blogs.oracle.com/archbeat/entry/podcast_show_notes_micro-services_roundtable
- [3] <http://de.slideshare.net/ewolff/micro-services-neither-micro-nor-service>
- [4] <http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html>

Microservices umsetzen

Dieser Teil des Buchs zeigt, wie Microservices umgesetzt werden können. Nach dem Studium dieses Teils kann man nicht nur Microservices-Architekturen entwerfen, sondern sie auch technisch umsetzen und die organisatorischen Auswirkungen bewerten.

Das [Kapitel 8](#) beschreibt die Architektur von Microservice-Systemen. Dabei geht es um das Zusammenspiel der einzelnen Microservices.

Die fachliche Architektur geht auf Domain-Driven Design als Basis einer Microservices-Architektur ein und zeigt Metriken, die Qualität der Architektur messbar machen. Eine Herausforderung ist das Architekturmanagement: Den Überblick über die Vielzahl an Microservices zu behalten, kann schwierig sein. Allerdings reicht oft ein Verständnis darüber, wie ein bestimmter Use Case implementiert ist und welche Microservices dafür zusammenarbeiten.

Kapitel 8: Architektur von Microservice-Systemen

Praktisch alle IT-Systeme sind einem mehr oder minder starken Wandel unterworfen. Daher muss die Architektur eines Microservice-Systems angepasst und das System muss weiterentwickelt werden. Dazu sind einige Herausforderungen zu lösen, die es bei Deployment-Monolithen so nicht gibt – beispielsweise ist eine Änderung der Aufteilung in Microservices nur schwer möglich. Die Änderung an einzelnen Microservices hingegen ist einfach.

Ebenso müssen Microservice-Systeme Legacy-Systeme integrieren. Das ist recht einfach möglich, weil Microservices Legacy-Systeme als Blackbox ansprechen können. Eine Ablösung kann zunehmend mehr Funktionalitäten in die Microservices verlagern, ohne die innere Struktur des Legacy-Systems anzupassen oder den Code im Detail zu verstehen.

Die technische Architektur umfasst typische Herausforderungen für die Implementierung von Microservices. Meistens gibt es eine zentrale Konfiguration und Koordination für alle Microservices. Ebenfalls teilt ein Load Balancer die Last zwischen den einzelnen Instanzen der Microservices auf. Die Sicherheitsarchitektur muss jedem Microservice die Freiheit lassen, eigene Berechtigungen im System umzusetzen, aber auch dafür sorgen, dass ein Benutzer sich nur einmal einloggen muss. Schließlich sollten die Microservices Informationen über sich selbst als Dokumentation und als Metadaten zurückgeben.

[Kapitel 9](#) zeigt die verschiedenen Möglichkeiten zur Integration und Kommunikation zwischen Microservices. Konkret gibt es drei mögliche Ebenen für die Integration:

Kapitel 9: Integration und Kommunikation

- Microservices können sich auf der Webebene integrieren. Jeder Microservice liefert dann ein Teil der Web-UI.

- Auf der Logik-Schicht können die Microservices mithilfe von REST oder Messaging kommunizieren.
- Eine Replikation von Daten ist ebenfalls möglich.

Mit diesen Technologien haben die Microservices interne Schnittstellen für andere Microservices. Das Gesamtsystem kann eine Schnittstelle nach außen haben. Änderungen an den verschiedenen Schnittstellen führen zu unterschiedlichen Herausforderungen. Versionierung der Schnittstellen und die Auswirkungen der Versionierung betrachtet das Kapitel daher ebenfalls.

[Kapitel 10](#) zeigt Möglichkeiten zur Architektur eines einzelnen Microservice. Für einen einzelnen Microservice gibt es unterschiedliche Ansätze:

Kapitel 10: Architektur eines Microservice

- CQRS trennt lesende und schreibende Zugriffe in zwei einzelne Services auf. Das ermöglicht kleinere Services und eine unabhängige Skalierung der beiden Teile.
- Event Sourcing verwaltet den Zustand eines Microservice durch einen Strom von Ereignissen, aus dem der aktuelle Zustand ermittelt werden kann.
- In einer *hexagonalen Architektur* hat der Microservice einen Kern, auf den verschiedene Adapter den Zugriff erlauben und der selber über solche Adapter mit anderen Microservices oder der Infrastruktur kommuniziert.

Jeder Microservice kann einer eigenen Architektur folgen.

Schließlich müssen Microservices technische Herausforderungen wie Resilience und Stabilität umsetzen – das muss eine technische Architektur lösen.

Das Testen steht im Mittelpunkt von [Kapitel 11](#). Auch Tests müssen Rücksicht auf die speziellen Herausforderungen der Microservices nehmen.

Kapitel 11: Testen von Microservices und Microservice-Systemen

Zunächst klärt das Kapitel, warum Tests überhaupt notwendig sind und wie man ein System grundsätzlich testen kann.

Microservices sind kleine Deployment-Einheiten. Das senkt das Risiko eines Deployments. Also können neben Tests auch Optimierungen beim Deployment zu einer niedrigeren Zahl an Fehlern führen.

Die Tests des Gesamtsystems stellen bei Microservices ein besonderes Problem dar, weil nur ein Microservice zur Zeit durch diese Phase gehen kann. Wenn die Tests eine Stunde dauern, gibt es nur acht Deployments an einem Arbeitstag. Bei 50 Microservices sind das viel zu wenig. Daher gilt es, diese Tests auf ein Mindestmaß zu reduzieren.

Oft lösen Microservices Legacy-Systeme ab. Die Microservices und das Legacy-System müssen beide getestet werden – auch im Zusammenspiel. Die Tests der einzelnen Microservices unterscheiden sich an einigen Stellen entscheidend von den Tests anderer Software-Systeme.

Consumer-Driven Contract Tests sind ein wesentlicher Bestandteil von Microservice-Tests: Sie testen die Erwartungen eines Microservice an eine Schnittstelle. So kann gewährleistet werden, dass Microservices zusammenspielen, ohne dass dazu die

Microservices zusammen in einem Integrationstest getestet werden müssen. Stattdessen hinterlegt der nutzende Microservice seine Anforderungen an die Schnittstelle in einem Test, den dann der genutzte Microservice ausführen kann.

Microservices müssen bestimmte Standards bezüglich Monitoring oder Logging einhalten. Auch die Einhaltung der Standards kann mit Tests überprüft werden.

Der Betrieb und Continuous Delivery stehen im Mittelpunkt von [Kapitel 12](#). Gerade die Infrastruktur ist eine wesentliche Herausforderung bei der Einführung von Microservices. Logging und Monitoring müssen einheitlich über alle Microservices umgesetzt werden, sonst ist der Aufwand dafür zu hoch. Ebenso sollte es ein einheitliches Deployment geben. Und schließlich sollten das Starten und Stoppen von Microservices einheitlich möglich sein – also eine einfache Steuerung. Für diese Bereiche stellt das Kapitel konkrete Technologien und Ansätze dar. Ebenso stellt das Kapitel einige Infrastrukturen vor, mit denen der Betrieb einer Microservices-Umgebung besonders einfach ist.

Kapitel 12: Betrieb und Continuous Delivery von Microservices

Schließlich zeigt [Kapitel 13](#), wie Microservices die Organisation beeinflussen. Microservices erlauben eine einfachere Aufteilung der Aufgaben auf unabhängige Teams und somit eine Parallelisierung der Arbeit an verschiedenen Features. Dazu müssen die Aufgaben auf die Teams verteilt werden, die dann die eigenen Microservices passend ändern. Allerdings können neue Features auch mehrere Microservices umfassen. Dann müssen Anforderungen von einem Team an ein anderes gestellt werden – und das führt zu viel Koordination und verzögert die Umsetzung von Features. Daher kann es sinnvoll sein, dass Teams auch die Microservices anderer Teams ändern.

Kapitel 13: Organisatorische Auswirkungen der Architektur

Microservices trennen die Architektur in Mikro- und MakroArchitektur auf: Die Teams können im Rahmen der Mikro-Architektur eigene Entscheidungen treffen, während die Makro-Architektur zentral verantwortet und koordiniert wird. In Bereichen wie Technologien, Betrieb, Architektur und Test können einzelne Aspekte der Mikro- oder Makro-Architektur zugeschlagen werden.

DevOps als Organisationsform passt gut zu Microservices, da eine enge Zusammenarbeit von Betrieb und Entwicklung gerade bei den infrastrukturlastigen Microservices nützlich ist.

Die unabhängigen Teams benötigen jeweils eigene unabhängige Anforderungen, die letztendlich von den Fachbereichen kommen müssen. Also haben Microservices auch in diesen Bereichen noch Auswirkungen.

Wiederverwendung von Code ist ebenfalls ein organisatorisches Problem: Wie koordinieren die Teams die unterschiedlichen Anforderungen an gemeinsame Komponenten? Ein Modell, das durch Open-Source-Projekte inspiriert ist, kann helfen.

Und natürlich ist da die Frage, ob Microservices ohne Organisationsänderung überhaupt möglich sind – immerhin sind die unabhängigen Teams einer der wesentlichen Gründe für Microservices.

8 Architektur von Microservice-Systemen

Dieses Kapitel handelt davon, wie die Microservices sich von außen verhalten sollen und wie das gesamte Microservice-System entwickelt werden kann. [Kapitel 9](#) zeigt mögliche Kommunikationstechnologien, die ein wichtiger Bestandteil der Technologien sind. In [Kapitel 10](#) geht es um den Aufbau einzelner Microservices.

[Abschnitt 8.1](#) zeigt, wie die fachliche Architektur eines Microservice-Systems beschaffen sein sollte. In [Abschnitt 8.2](#) geht es um geeignete Werkzeuge, um die Architektur zu visualisieren und zu managen. Der [Abschnitt 8.3](#) zeigt auf, wie die Architektur schrittweise angepasst werden kann. Nur die Anpassung der Architektur sorgt dafür, dass das System langfristig wartbar bleibt und weiterentwickelt werden kann. Welche Ziele und welches Vorgehen für die Weiterentwicklung wichtig sind, zeigt [Abschnitt 8.4](#).

Es folgen einige Ansätze für die Architektur eines Microservice-Systems. Event-driven Architecture beschreibt [Abschnitt 8.6](#). Damit sind sehr stark entkoppelte Architekturen möglich. [Abschnitt 8.5](#) geht auf die besonderen Herausforderungen ein, wenn eine Legacy-Anwendung durch Microservices ergänzt oder abgelöst werden soll.

Schließlich geht es in [Abschnitt 8.7](#) darum, welche technischen Aspekte in der Architektur eines Systems aus Microservices relevant sind. Dazu zählen Mechanismen zur Koordination und Konfiguration ([Abschnitt 8.8](#)), für Service Discovery ([Abschnitt 8.9](#)), Load Balancing ([Abschnitt 8.10](#)), Skalierbarkeit ([Abschnitt 8.11](#)), Sicherheit ([Abschnitt 8.12](#)) und schließlich Dokumentation und Metadaten ([Abschnitt 8.13](#)).

8.1 Fachliche Architektur

Die fachliche Architektur des Microservice-Systems bestimmt, welche Microservices in dem System welche Fachlichkeiten abbilden sollen. Sie liefert eine Aufteilung der gesamten Domäne in verschiedene Bereiche, die jeweils ein Microservice und damit auch ein Team umsetzt. Diese Aufteilung stellt eine zentrale Herausforderung bei Microservices dar. Schließlich ist eine wichtige Motivation für die Nutzung von Microservices, dass Änderungen an der Fachlichkeit idealerweise von einem Team an einem Microservice vorgenommen werden können – und damit wenig teamübergreifende Koordination und Kommunikation notwendig ist. So unterstützen Microservices eine Skalierung der Software-Entwicklung, weil auch bei großen Teams nur wenig Kommunikation notwendig ist und so auch große Teams produktiv arbeiten können.

Zentral ist dafür eine fachliche Aufteilung der Microservices, bei der die Änderungen tatsächlich auf einen Microservice und damit auf ein Team begrenzt sind. Wenn die Aufteilung in Microservices das nicht erlaubt, benötigen Änderungen mehr Koordination und Kommunikation. Der Microservice-Ansatz kann dann seine Vorteile nicht voll ausspielen.

[Abschnitt 4.3](#) hat bereits eine Aufteilung von Microservices anhand des Strategic Designs aus dem

Strategic Design und Domain-Driven Design

Domain-Driven-Design diskutiert. Zentral ist, dass die Microservices in Kontexte aufgeteilt werden – also Bereiche, die jeweils eine abgeschlossene Funktionalität darstellen.

Oft entwickeln Architekten eine Aufteilung von Microservices anhand von Daten in einem Domänenmodell. Die Hoheit über bestimmte Arten von Daten obliegt dann bestimmten Microservices. Dann gibt es beispielsweise einen Microservice für Kunden, einen für Waren und einen für Lieferungen. Dieser Ansatz widerspricht der BOUNDED-CONTEXT-Idee, nach der eine einheitliche Modellierung von Daten unmöglich ist. Der Ansatz isoliert Änderungen auch nur sehr schlecht. Wenn ein Prozess geändert werden soll und dazu Datenschemata ebenfalls angepasst werden müssen, ist die Änderung über verschiedene Services verteilt. So kann eine Änderung am Bestellprozess die Datenmodellierung für Kunden, Waren und Lieferungen betreffen. Wenn das der Fall ist, werden neben dem Microservice für den Bestellprozess auch die drei Microservices für die verschiedenen Datenmodelle geändert. Es kann daher sinnvoll sein, bestimmte Teile der Daten für Kunden, Waren und Lieferungen im Microservice für den Bestellprozess zu halten. Dann sind Änderungen an dem Bestellprozess auch auf nur einen Microservice beschränkt, wenn die Datenmodellierung geändert werden muss.

Es kann dennoch Services geben, deren Aufgabe die Verwaltung bestimmter Daten ist. Beispielsweise kann es notwendig sein, zumindest die grundlegenden Daten einer bestimmten Business-Entität in einem Service zu verwalten. So kann ein Dienst durchaus die Kundendaten verwalten, aber spezielle Daten des Kunden wie eine Nummer für ein Bonusprogramm anderen Microservices überlassen – beispielsweise dem Microservice für den Bestellprozess, der diese Nummer vermutlich kennen muss.

Ein Beispiel – nämlich die Architektur des Otto-Shops [1] – verdeutlicht das. Es gibt Bereiche wie User, Order oder Product, die eher an Daten orientiert sind, aber auch Tracking, Search & Navigation oder Personalisation, die sich nicht an Daten, sondern an Funktionalitäten orientieren. Genau ein solcher fachlicher Schnitt sollte in einem Microservice-System angestrebt werden.

Eine fachliche Architektur basiert auf einem genauen Verständnis der Fachlichkeit. Die fachliche Architektur beinhaltet nicht nur die Aufteilung des Systems in Microservices, sondern auch die Abhängigkeiten. Eine Abhängigkeit ergibt sich, wenn ein Microservice einen anderen nutzt – also beispielsweise den Microservice aufruft, ein Element aus der UI des anderen Microservice wiederverwendet oder Daten repliziert. Eine solche Abhängigkeit bedeutet, dass Änderungen an dem genutzten Microservice auch den nutzenden Microservice beeinflussen können. Ändert der genutzte Microservice beispielsweise seine Schnittstelle, muss der nutzende Microservice an diese Änderungen angepasst werden. Außerdem können neue Anforderungen an den nutzenden Microservice notwendig machen, dass der genutzte Microservice seine Schnittstelle ändert. Wenn für die Umsetzung der Anforderungen mehr Daten notwendig sind, muss der genutzte Microservice diese Daten anbieten und die Schnittstelle entsprechend anpassen.

Bei Microservices haben die Abhängigkeiten Nachteile über die Software-Architektur hinaus: Schließlich können Microservices von unterschiedlichen Teams implementiert

werden. Dann erzwingt eine Änderung an einer Schnittstelle auch Kollaboration zwischen Teams – und das ist wiederum aufwendig und langwierig.

Solche Abhängigkeiten zwischen den Microservices sind zentral für die Architektur des Systems. Zu viele Abhängigkeiten führen dazu, dass die Microservices kaum noch isoliert änderbar sind – was dem Ziel der unabhängigen Entwicklung der Microservices widerspricht. Hier gelten die beiden grundlegenden Regeln für gute Architekturen:

- Zwischen Komponenten wie Microservices soll es eine *lose Kopplung* geben. Das bedeutet, dass sie nur wenige Abhängigkeiten zu anderen Microservices haben. Dadurch sind sie leichter änderbar, weil Änderungen nur Auswirkungen auf einen Microservice haben.
- Innerhalb einer Komponente wie einem Microservice sollen die Bestandteile eng zusammenarbeiten. Man spricht von einer *hohen Kohäsion*. Dadurch ist sichergestellt, dass alle Bestandteile innerhalb eines Microservice tatsächlich zusammengehören.

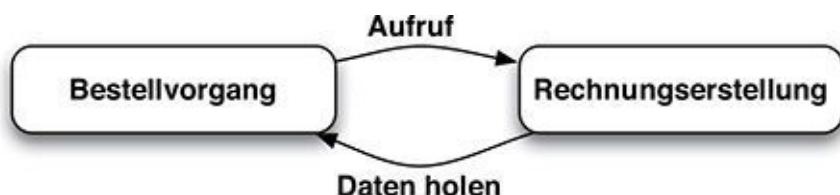
Wenn diese beiden Voraussetzungen nicht gegeben sind, sind Änderungen an dem Microservice kaum isoliert umsetzbar und Änderungen müssen über mehrere Teams und Microservices koordiniert sein – was Microservices-Architekturen gerade vermeiden sollen. Aber das ist eigentlich nur ein Symptom: Das grundlegende Problem ist der fachliche Schnitt zwischen den Microservices, bei dem offensichtlich Funktionalitäten über verschiedene Microservices verteilt worden sind, die eigentlich zusammen in einen Microservice gehören. Beispielsweise muss ein Bestellprozess auch eine Rechnung erzeugen. Diese Funktionalitäten sind so verschieden, dass sie in mindestens zwei Microservices aufgeteilt werden müssen. Wenn aber jede Änderung im Bestellprozess auch den Microservice für die Rechnungsstellung betrifft, dann ist die fachliche Modellierung nicht optimal und sollte angepasst werden. Die Funktionalitäten müssen anders auf die Microservices verteilt werden, wie wir noch sehen werden.

Nicht nur die Anzahl der Abhängigkeiten ist ein *Fachlich ungewollte Abhängigkeiten* Problem. Bestimmte fachliche Abhängigkeiten können schlicht unsinnig sein. Wenn beispielsweise in einem E-Commerce-System das Team für die Produktsuche plötzlich Schnittstellen mit der Rechnungsstellung hat, ist das überraschend, denn fachlich sollte das nicht so sein. Aber es gibt gerade bei der Fachlichkeit für Laien immer wieder Überraschungen. Wenn die Abhängigkeit fachlich nicht sinnvoll ist, muss etwas im Schnitt der Microservices falsch sein. Die Microservices implementieren vielleicht Features, die fachlich in andere Microservices gehören. Vielleicht wird bei der Produktsuche ein Scoring des Kunden benötigt, das in der Rechnungsstellung implementiert ist. Dann ist die Frage, ob diese Funktionalität dort wirklich sinnvoll untergebracht ist. Um das System langfristig wartbar zu halten, müssen solche Abhängigkeiten hinterfragt und gegebenenfalls aus dem System entfernt werden. Beispielsweise kann das Scoring in einen eigenen Microservice ausgelagert oder in einen anderen Microservice verschoben werden.

Weitere Probleme bei der übergreifenden Architektur können *Zyklische Abhängigkeiten* sein. Nehmen wir an,

dass in einem Microservice-System der Microservice für den Bestellvorgang den Microservice für die Rechnungserstellung aufruft. Der Microservice für die Rechnungserstellung holt sich Daten aus dem Bestellung-Microservice ab. Wenn nun der Bestellung-Microservice geändert wird, können Änderungen am Rechnung-Microservice notwendig sein, weil sich der Rechnung-Microservice Daten von dem Bestellung-Microservice holt. Umgekehrt können Änderungen am Rechnung-Microservice auch Änderungen am Bestellung-Microservice notwendig machen, da der Bestellung-Microservice den Rechnung-Microservice aufruft. Aus diesem Grund sind zyklische Abhängigkeiten problematisch: Die Komponenten sind nicht mehr isoliert änderbar, was eigentlich das Ziel bei einer Aufteilung in Komponenten ist. Bei Microservices wird besonders viel Wert auf die Unabhängigkeit gelegt, die in diesem Fall verletzt wird. Neben einer Koordination der Änderungen kann es auch vorkommen, dass das Deployment koordiniert werden muss. Wenn eine neue Version eines Microservice ausgerollt wird, muss gegebenenfalls auch eine neue Version eines anderen Microservice ausgerollt werden, wenn sie zyklisch voneinander abhängen.

Abb. 8–1 Zyklische Abhängigkeit



Der Rest des Kapitels zeigt Ansätze, mit denen Microservices-Architekturen so aufgebaut werden können, dass sie fachlich gut strukturiert sind. Dann sollten Metriken wie Kohäsion und lose Kopplung zeigen, dass die Architektur wirklich passt. Microservices in Ansätzen wie Event-driven Architecture ([Abschnitt 8.6](#)) haben technisch kaum direkte Abhängigkeiten, weil nur Nachrichten verschickt werden. Wer die Nachrichten verschiickt und wer sie verarbeitet, ist aus dem Code heraus kaum zu ermitteln, sodass die Metriken sehr gut aussehen. Auf fachlicher Ebene kann das System dennoch viel zu kompliziert sein, weil die fachlichen Abhängigkeiten nicht bei den Metriken betrachtet werden. Die fachlichen Abhängigkeiten entstehen, wenn zwei Microservices Nachrichten austauschen. Das ist aber durch Code-Analyse kaum zu ermitteln, sodass die Metriken immer recht gut aussehen werden. Metriken können also nur ein Hinweis auf ein Problem sein. Einfach nur die Metriken optimieren, optimiert die Symptome, löst aber das Problem nicht. Schlimmer noch: Selbst Systeme mit guten Metriken können in der Architektur Schwächen haben.

Ein besonderes Problem bei Microservices ist, dass Abhängigkeiten zwischen Microservices auch das unabhängige Deployment beeinflussen können. Wenn ein Microservice eine neue Version eines anderen Microservice benötigt, weil er beispielsweise eine neue Version einer Schnittstelle nutzt, ist auch das Deployment aneinander gekoppelt: Der genutzte Microservice muss deployt werden, bevor der andere Microservice deployt werden kann. Im Extremfall kann das dazu führen, dass eine große Zahl Microservices koordiniert deployt werden muss – und das ist eigentlich, was vermieden werden soll. Microservices sollen unabhängig voneinander deployt werden. Daher können Abhängigkeiten zwischen Microservices noch ein größeres Problem sein, als dies bei Modulen innerhalb eines Deployment-Monolithen der Fall wäre.

8.2 Architekturmanagement

Für die fachliche Architektur ist es entscheidend, welche Microservices es gibt und wie die Kommunikationsbeziehungen zwischen den Microservices aussehen. Auch in anderen Systemen sind die Beziehungen zwischen Komponenten sehr wichtig. Wenn fachliche Komponenten auf Module, Klassen, Java-Packages, JAR-Dateien oder DLLs abgebildet werden, können Werkzeuge die Beziehungen zwischen den Komponenten darstellen und die Einhaltung bestimmter Regeln kontrollieren. Dabei wird eine statische Code-Analyse verwendet.

Wenn ein solches Architekturmanagement nicht stattfindet, schleichen sich schnell ungewollte Abhängigkeiten ein. Die Architektur wird dann zunehmend komplexer und schwieriger zu durchschauen. Nur mithilfe von Architekturmanagement-Werkzeugen können Entwickler und Architekten überhaupt den Überblick über ein System behalten. In einer Entwicklungsumgebung sehen Entwickler nur einzelne Klassen. Die Abhängigkeiten zwischen den Klassen finden sich nur im Quelltext und sind nicht so ohne Weiteres direkt erkennbar.

Abb. 8–2 Screenshot des Architekturmanagement-Werkzeugs Structure 101

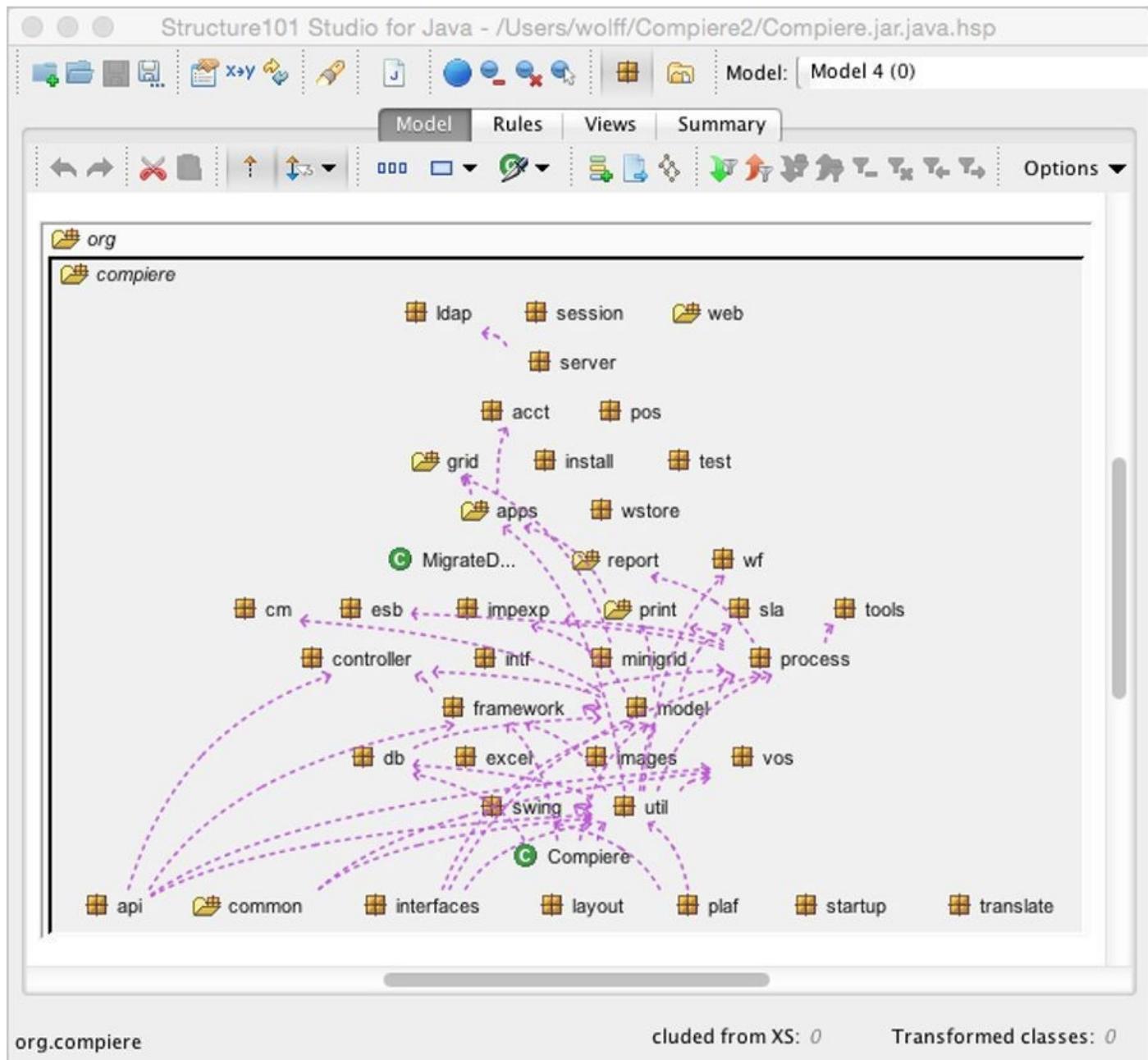


Abbildung 8–2 zeigt die Analyse eines Java-Projekts im Architekturmanagement-Werkzeug Structure 101. Die Abbildung zeigt Klassen und Java-Packages, die Klassen enthalten. Eine Levelized Structure Map (LSM) stellt sie im Überblick dar. Klassen und Packages, die weiter oben in einer LSM angeordnet sind, nutzen Klassen und Packages, die weiter unten in der LSM angeordnet sind. Diese Beziehungen zeigt das Diagramm nicht, um so die Übersicht zu erhöhen.

Architekturen sollten zyklenfrei sein. Zyklische Abhängigkeiten bedeuten, dass zwei Artefakte sich wechselseitig nutzen. In dem Screenshot werden solche Zyklen durch gestrichelte Linien dargestellt. Sie gehen immer von unten nach oben. Die andere Beziehung in dem Zyklus würde von oben nach unten gehen und wird daher nicht dargestellt.

Neben den Zyklen fallen auch Packages auf, die an der falschen Stelle sind. So gibt es ein Package `util`, das dem Namen nach Hilfsklassen enthalten sollte. Es ist aber nicht ganz unten in dem Diagramm. Es muss Abhängigkeiten zu Packages oder Klassen weiter unten haben – was eigentlich nicht der Fall sein sollte. Hilfsklassen sollten unabhängig von allen

anderen Bestandteilen des Systems sein und daher in einer LSM ganz unten stehen.

Architekturmanagement-Werkzeuge wie Structure 101 können nicht nur Architekturen analysieren, sondern Architekten können mit dem Werkzeug auch erlaubte Beziehungen zwischen Packages und Klassen definieren. Verstößt ein Entwickler gegen diese Regeln, bekommt er eine Fehlermeldung und kann den Code umstellen.

Mit Werkzeugen wie Structure 101 ist eine Visualisierung der Architektur eines Systems kein Problem. Man muss lediglich den kompilierten Code in das Werkzeug laden und analysieren lassen. So ist die Sichtbarkeit in die Architektur gewährleistet.

Bei Microservices ist das Problem viel schwieriger: Die Beziehungen zwischen den Microservices sind nicht so einfach zu ermitteln wie die Beziehungen zwischen Code-Bestandteilen. Schließlich können die Microservices in unterschiedlichen Technologien implementiert sein. Sie kommunizieren nur über das Netzwerk miteinander. Diese Beziehungen entziehen sich einem Management auf Code-Ebene, weil sie im Code nur indirekt auftauchen. Wenn die Beziehungen zwischen den Microservices nicht bekannt sind, wird das Architekturmanagement unmöglich.

Microservices und Architekturmanagement

Um die Architektur zu visualisieren und zu managen, gibt es unterschiedliche Möglichkeiten:

- Jeder Microservice kann eine Dokumentation (siehe [Abschnitt 8.13](#)) mitbringen, in der die genutzten Microservices aufgelistet sind. Diese Dokumentation muss in einem fest definierten Format vorliegen, das für eine Visualisierung genutzt werden kann.
- Die Kommunikationsinfrastruktur kann die notwendigen Daten liefern. Wenn eine Service Discovery ([Abschnitt 8.9](#)) genutzt wird, kennt sie alle Microservices und weiß auch, welche Microservices Zugriff auf welche anderen Microservices haben. Diese Daten können wieder für die Visualisierung der Beziehungen genutzt werden.
- Wenn der Zugriff der Microservices untereinander durch eine Firewall abgesichert ist, verraten die Regeln für die Firewall zumindest, welcher Microservice mit welchem Microservice sprechen kann. Das kann wiederum als Basis für eine Visualisierung genutzt werden.
- Der Netzwerkverkehr gibt Aufschluss darüber, welche Microservices mit welchen anderen Microservices sprechen. Dazu können Werkzeuge wie Packetbeat (siehe [Abschnitt 12.3](#)) hilfreich sein, die auf Basis des mitgeschnittenen Netzwerkverkehrs die Beziehungen zwischen den Microservices darstellen.
- Die Aufteilung in Microservices sollte auch der Aufteilung in Teams entsprechen. Wenn zwei Teams kaum noch getrennt voneinander arbeiten können, dann ist vermutlich ein Problem in der Architektur daran schuld: Die Microservices der Teams sind so stark voneinander abhängig, dass sie nur noch gemeinsam geändert werden können. Den beteiligten Teams ist wahrscheinlich schon aufgrund der Kommunikation klar, welche Microservices problematisch sind. Zur Verifikation kann nun noch ein Architekturmanagement-Werkzeug oder eine Visualisierung verwendet werden. Dazu können gegebenenfalls sogar manuell erhobene

Informationen ausreichend sein.

Um die Daten über die Abhängigkeiten auszuwerten, sind verschiedene Werkzeuge nützlich:

Werkzeuge

- Es gibt Versionen von Structure 101 [2], die als Eingabe eigene Datenstrukturen nutzen können. Einen passenden Importer muss man dann selbst schreiben. Structure 101 erkennt dann zyklische Abhängigkeiten und kann die Abhängigkeiten als Grafiken aufbereiten.
- Gephi [21] kann komplexe Graphen generieren, die für eine Visualisierung der Abhängigkeiten von Microservices hilfreich sind. Dazu muss ein eigener Importer geschrieben werden, der die Abhängigkeiten der Microservices aus einer geeigneten Quelle in Gephi importiert.
- jQAssistant [22] basiert auf der Graphen-Datenbank neo4j. Es kann durch eigene Importer erweitert werden. Das Datenmodell kann dann mit Regeln überprüft werden.

Für alle diese Werkzeuge ist eine eigene Entwicklung notwendig. Man kann keine Microservices-Architektur einfach so analysieren, sondern das ist nur mit entsprechendem Aufwand möglich. Weil die Kommunikation der Microservices nicht standardisiert werden kann, wird man wahrscheinlich auch in Zukunft nicht um eigene Entwicklungen umhinkommen.

Das Architekturmanagement von Microservices ist wichtig, weil nur so Wildwuchs in den Beziehungen zwischen Microservices verhindert werden kann. Microservices sind diesbezüglich eine Herausforderung: Einen Deployment-Monolithen zu analysieren ist mit modernen Werkzeugen sehr schnell und einfach möglich. Für eine Microservices-Architektur gibt es noch nicht einmal ein Werkzeug, das die gesamte Struktur einfach analysieren kann. Die Teams müssen die Voraussetzungen für eine Analyse erst einmal schaffen. Änderungen an den Beziehungen zwischen den Microservices sind schwierig – wie der nächste Abschnitt zeigen wird. Umso wichtiger ist es, die Architektur der Microservices von Anfang an zu überprüfen, um so bei Problemen möglichst frühzeitig eingreifen zu können. Zugute kommt den Microservices-Architekturen, dass die Architektur sich auch in der Organisation niederschlägt. Probleme in der Kommunikation weisen also auf ein Architekturproblem hin. Auch ohne ein Architekturmanagement werden Probleme mit der Architektur so offensichtlich.

Ist Architekturmanagement wichtig?

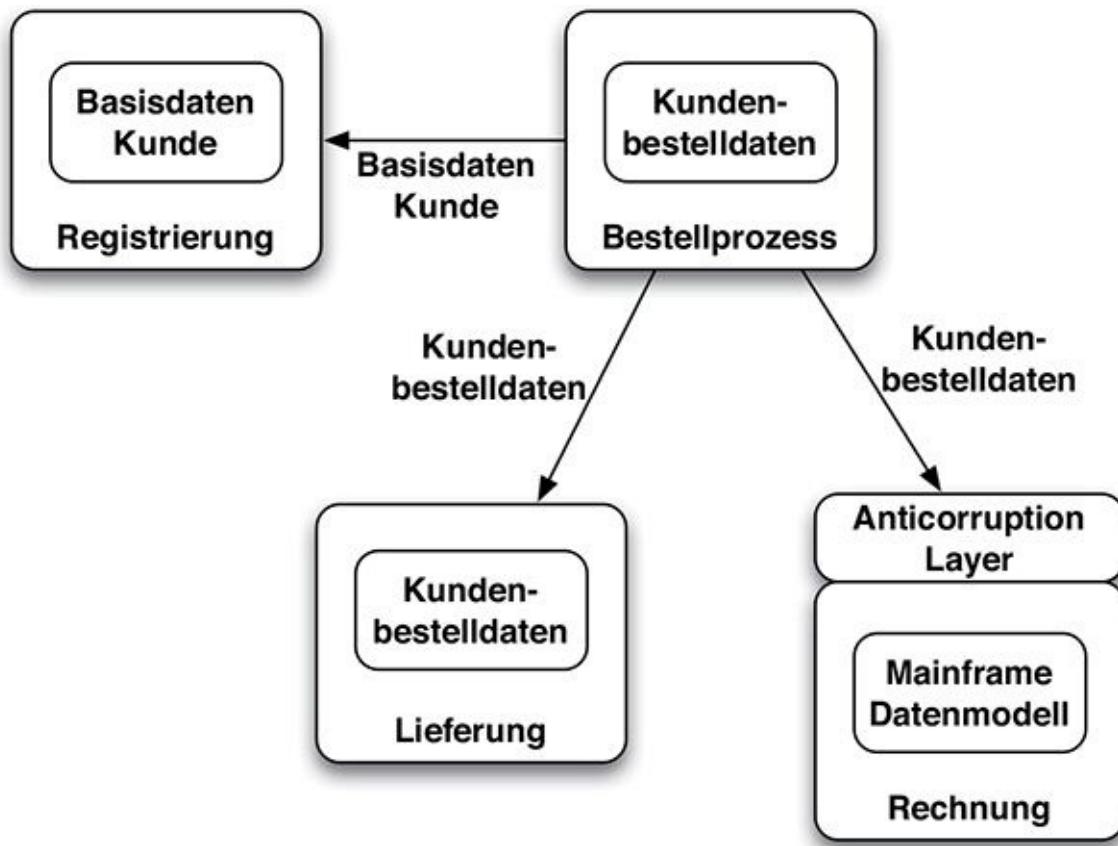
Auf der anderen Seite zeigt die Erfahrung mit komplexen Microservice-Systemen, dass in solchen Systemen niemand den Überblick über die komplette Architektur hat. Das ist auch nicht notwendig, weil die meisten Änderungen auf einen Microservice begrenzt sind. Wenn ein bestimmter Anwendungsfall geändert werden soll, an dem mehrere Microservices beteiligt sind, reicht es aus, diese Interaktion und die Microservices zu verstehen. Ein globaler Überblick ist nicht mehr unbedingt notwendig. Das ist eine Konsequenz der Unabhängigkeit der Microservices.

Eine Möglichkeit, um einen Überblick über die Architektur eines Microservice-Systems zu bekommen, sind Context Maps [5]. Sie stellen dar, welche Domänenmodelle in welchen

Context Map

Microservices genutzt werden, und visualisieren so die verschiedenen BOUNDED CONTEXTS (siehe [Abschnitt 4.3](#)). Die BOUNDED CONTEXTS beeinflussen nicht nur die interne Darstellung der Daten in den Microservices. Auch bei den Aufrufen der Microservices untereinander werden Daten ausgetauscht. Sie müssen irgendwelchen Modellen entsprechen. Allerdings können die Datenmodelle bei der Kommunikation durchaus anders als die internen Repräsentationen sein. Wenn ein Microservice beispielsweise Empfehlungen für Kunden in einem E-Commerce-Shop ermitteln soll, können intern dazu komplexe Modelle genutzt werden, die viele Informationen über Kunden, Produkte und Bestellungen enthalten und sie in komplexe Beziehungen setzen. Nach außen sind die Modelle vermutlich wesentlich einfacher.

Abb. 8–3 Eine beispielhafte Context Map



Ein Beispiel für eine Context Map zeigt [Abbildung 8–3](#)

- Die *Registrierung* erfasst die *Basisdaten* eines Kunden. Der *Bestellprozess* nutzt dieses Datenformat auch, um mit der *Registrierung* zu kommunizieren.
- Im *Bestellprozess* werden die Basisdaten des Kunden um Daten wie die Rechnungs- oder Lieferadressen zu den *Kundenbestelldaten* ergänzt. Das entspricht einem geteilten Kern (SHARED KERNEL) (siehe [Abschnitt 4.3](#)). Der *Bestellprozess* teilt sich mit der *Registrierung* den Kern der Kundendaten.
- Die *Kundenbestelldaten* nutzen auch die *Lieferung* und die *Rechnung* zur Kommunikation, die *Lieferung* sogar zur internen Repräsentation der Kunden. Damit ist dieses Modell eine Art Standard-modell für die Kommunikation der Kundendaten.
- Die *Rechnung* nutzt ein altes *Mainframe-Datenmodell*. Daher werden durch eine Antikorruptionsschicht die Kundenbestelldaten aus der Kommunikation nach außen von der internen Repräsentation entkoppelt. Das Datenmodell stellt nämlich eine sehr

schlechte Abstraktion dar, die auf keinen Fall Auswirkungen auf die anderen Microservices haben soll.

In diesem Modell fällt auf, dass die interne Repräsentation der Daten in der Registrierung sich in den Bestellprozess fortpflanzt. Sie dient dort als Basis für die Kundenbestelldaten. Dieses Modell wird in der Lieferung als internes Datenmodell genutzt und in der Kommunikation mit der Rechnung und der Lieferung. So wird das Modell schwer änderbar, weil viele Services dieses Modell nutzen. Bei einer Änderung des Modells müssten alle diese Services modifiziert werden.

Es hat aber auch Vorteile. Wenn alle diese Services dieselbe Änderung an dem Datenmodell vornehmen müssten, ist nur eine Änderung notwendig, die dann gleich alle Microservices auf den aktuellen Stand bringen. Allerdings widerspricht das der Idee, dass Änderungen immer nur einen Microservice betreffen sollten. Wenn die Änderung auf das Modell beschränkt bleibt, ist das gemeinsame Modell vorteilhaft, weil alle die aktuelle Modellierung automatisch nutzen. Wenn die Änderung aber Änderungen in den Microservices nach sich zieht, müssen nun viele Microservices geändert werden – und koordiniert in Produktion gebracht werden. Das widerspricht einem unabhängigen Deployment der Microservices.

Selber ausprobieren und experimentieren

Lade ein Werkzeug zur Analyse von Architekturen herunter. Kandidaten sind Structure 101 (<http://structure101.com>), Gephi (<http://gephi.github.io/>) oder jQAssistant (<http://jqassistant.org>). Verschaffe dir mit dem Werkzeug einen Überblick über eine vorhandene Code-Basis. Welche Möglichkeiten gibt es, eigene Abhängigkeitsgraphen in das Werkzeug einzufügen? Damit wäre es möglich, die Abhängigkeiten in einer Microservices-Architektur ebenfalls mit diesem Werkzeug zu analysieren.

<https://github.com/adrianco/spigo> ist eine Simulation für die Kommunikation zwischen Microservices. Sie kann genutzt werden, um einen Eindruck von komplexeren Microservices-Architekturen zu bekommen.

8.3 Techniken zum Anpassen der Architektur

Microservices sind vor allem interessant für Software, die vielen Änderungen unterworfen ist. Durch die Aufteilung in Microservices zerfällt das System in Deployment-Einheiten, die unabhängig voneinander weiterentwickelt werden können. So kann jeder Microservice einen eigenen Strom von Stories oder Anforderungen umsetzen. Dadurch kann an mehreren Änderungen gleichzeitig gearbeitet werden, ohne dass dabei viel Koordination notwendig ist.

Die Erfahrung zeigt, dass die Architektur eines Systems Änderungen unterworfen ist. Eine bestimmte Aufteilung in fachliche Komponenten mag zunächst sinnvoll erscheinen. Wenn der Architekt aber die Domäne genauer kennlernt, kommt er oft zu der Überzeugung, dass eine andere Aufteilung besser wäre. Neue Anforderungen lassen sich nur schwer mit der Architektur umsetzen, weil sie unter anderen Voraussetzungen entworfen worden ist. Das passiert gerade bei agilen Prozessen, die weniger Planung und

mehr Flexibilität mit sich bringen.

Ein System mit einer schlechten Architektur ist meistens nicht dadurch entstanden, dass am Anfang die falsche Architektur gewählt worden ist. Auf Basis der Informationen, die am Anfang des Projekts bekannt waren, war die Architektur oft gut und schlüssig. Das Problem ist, dass die Architektur nicht geändert wird, wenn es neue Erkenntnisse gibt, die eine Änderung der Architektur nahe legen. Das Symptom stand schon im letzten Absatz: Neue Anforderungen lassen sich nicht mehr schnell und einfach umsetzen. Dafür müsste die Architektur geändert werden. Wenn dieser Änderungsdruck zu lange ignoriert wird, passt die Architektur irgendwann überhaupt nicht mehr. Die ständige Anpassung und Änderung der Architektur sind die wesentliche Voraussetzung dafür, dass die Architektur tatsächlich tragfähig bleibt.

Woher kommt schlechte Architektur?

Dieser Abschnitt zeigt, mit welchen Techniken das Zusammenspiel der Microservices geändert werden kann, um die Architektur des Gesamtsystems anzupassen.

Innerhalb eines Microservice sind Anpassungen *Änderungen in Microservices* einfach. Die Microservices sind klein und überschaubar. Strukturen anzupassen ist kein großes Problem. Und wenn die Architektur des Microservice völlig unzureichend ist, kann er neu geschrieben werden, weil er nicht besonders groß ist. Innerhalb eines Microservice ist es auch kein Problem, Bestandteile zu verschieben oder den Code anderweitig umzustrukturieren. Unter dem Begriff Refactoring [3] sind Techniken bekannt, um die Strukturierung des Codes zu verbessern. Viele davon automatisieren Entwicklungswerkzeuge sogar. So kann Code eines Microservice sehr einfach angepasst werden.

Änderungen der Gesamtarchitektur

Wenn aber die Aufteilung der Aufgaben auf die Microservices nicht mehr den Anforderungen entspricht, reicht die Änderung eines Microservice nicht. Für solche Anpassung der Gesamtarchitektur müssen Funktionalitäten zwischen den Microservices verschoben werden. Dafür kann es verschiedene Gründe geben:

- Der Microservice ist zu groß und muss aufgeteilt werden. Indizien können sein, dass der Microservice kaum noch verständlich oder sogar so groß ist, dass er von einem Team nicht mehr weiterentwickelt werden kann. Ein weiteres Indiz kann sein, dass der Microservice mehr als einen BOUNDED CONTEXT umfasst.
- Funktionalität gehört eigentlich in einen anderen Microservice. Ein Indikator dafür kann sein, dass bestimmte Teile eines Microservice sehr viel mit einem anderen Microservice kommunizieren. Dann haben die Microservices untereinander keine lose Kopplung. Solche intensive Kommunikation kann darauf hindeuten, dass der Bestandteil in einen anderen Microservice gehört. Ebenso kann eine niedrige Kohäsion in einem Microservice darauf hindeuten, dass der Microservice aufgeteilt werden muss. Dann gibt es in dem Microservice Bereiche, die wenig voneinander abhängen. Die müssen dann auch nicht unbedingt in einem Microservice sein.
- Funktionalitäten sollen von mehreren Microservices gemeinsam genutzt werden. Das kann beispielsweise notwendig werden, wenn ein Microservice durch eine neue

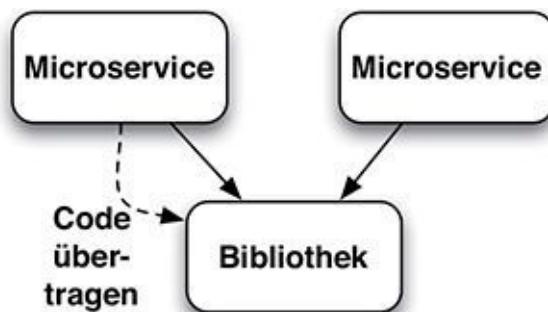
Funktionalität Logik aus einem anderen Microservice nutzen muss.

Es geht um drei Herausforderungen: Microservices müssen aufgeteilt werden, Code muss von einem Microservice in einen anderen verschoben werden und mehrere Microservices sollen Code gemeinsam nutzen.

Gemeinsame Bibliothek

Wenn die beiden Microservices Code gemeinsam nutzen sollen, kann der Code in eine gemeinsame Bibliothek ausgelagert werden (siehe Abb. 8–4). Der Code wird aus dem Microservice entfernt und so verpackt, dass er auch im anderen Microservice genutzt werden kann. Voraussetzung dafür ist, dass die Microservices in Technologien geschrieben sind, mit denen die Nutzung einer gemeinsamen Bibliothek möglich ist. Das kann bedeuten, dass sie in derselben Sprache geschrieben sein oder zumindest dieselbe Plattform verwenden müssen – z. B. die JVM (Java Virtual Machine) oder die .NET Common Language Runtime (CLR).

Abb. 8–4 Gemeinsame Bibliothek



Eine gemeinsame Bibliothek bedeutet, dass die Microservices voneinander abhängig werden. Die Arbeit an der Bibliothek muss koordiniert werden. Features für beide Microservices müssen in die Bibliothek aufgenommen werden. Durch die Hintertür bekommt so jeder Microservice Änderungen mit, die eigentlich für den anderen Microservice bestimmt sind. Das kann zu Fehlern führen. Daher müssen die Teams die Entwicklung der Bibliothek und Änderungen an der Bibliothek miteinander koordinieren. Unter bestimmten Bedingungen kann eine Änderung an einer Bibliothek sogar erzwingen, dass ein Microservice neu deployt werden muss – zum Beispiel weil in der Bibliothek eine Sicherheitslücke geschlossen worden ist.

Auch bekommen die Microservices gegebenenfalls durch die Bibliothek weitere Code-Abhängigkeiten zu 3rd Party Libraries. In einer Java-JVM können 3rd Party Libraries nur in einer Version vorgehalten werden. Wenn die gemeinsame Bibliothek eine bestimmte Version einer 3rd Party Library erzwingt, ist auch der Microservice auf diese Version festgelegt. Außerdem haben Bibliotheken oft ein bestimmtes Programmiermodell. So können die Bibliotheken Code bereitstellen, der aufgerufen werden kann, oder ein Framework, in das eigener Code integriert werden kann, der durch das Framework aufgerufen wird. Die Bibliothek kann ein asynchrones Modell verfolgen oder ein synchrones Modell. Solche Ansätze können mehr oder weniger gut zu den jeweiligen Microservices passen.

Microservices fokussieren nicht auf die Wiederverwendung von Code, weil dadurch

neue Abhängigkeiten zwischen den Microservices entstehen. Ein wichtiges Ziel von Microservices ist Unabhängigkeit, sodass Wiederverwendung von Code oft mehr Nachteile als Vorteile mit sich bringt. Das ist eine Abkehr von dem Ideal der Code-Wiederverwendung. Darauf haben die Entwickler noch in den Neunzigerjahren große Hoffnung gesetzt, um die Produktivität zu steigern. Das Auslagern des Codes in eine Bibliothek hat auch Vorteile. Fehler und Sicherheitslücken müssen nur einmal behoben werden. Die Microservices nutzen jeweils die aktuelle Version der Bibliothek und bekommen so die Lösung der Fehler gleich mit.

Ein weiteres Problem bei der Wiederverwendung von Code ist, dass dazu detailliertes Wissen über den Code notwendig ist – gerade bei Frameworks, in die sich der eigene Code einbetten muss. Diese Art der Wiederverwendung nennt man Whitebox-Reuse: Die internen Strukturen des Codes müssen bekannt sein – nicht nur die Schnittstelle. Diese Art der Wiederverwendung setzt ein genaues Verständnis des Codes voraus, der wiederverwendet werden soll, und damit ist die Hürde für die Benutzung sehr hoch.

Ein Beispiel kann eine Bibliothek sein, die das Erstellen von Metriken für das Monitoring des Systems vereinfacht. Sie wird im Rechnung-Microservice verwendet. Andere Teams wollen den Code auch nutzen. Der Code wird in eine Bibliothek extrahiert. Weil es technischer Code ist, wird er bei fachlichen Änderungen nicht modifiziert. Dadurch beeinflusst diese Bibliothek das unabhängige Deployment und die unabhängige Entwicklung fachlicher Features nicht. Aus der Bibliothek sollte ein internes Open-Source-Projekt (siehe [Abschnitt 13.7](#)) werden. Fachlichen Code in eine gemeinsame Bibliothek auszulagern ist problematisch, weil dann eher Deployments aller Microservices notwendig sind. Wenn beispielsweise die Modellierung eines Kunden in einer Bibliothek umgesetzt ist, dann muss jede Änderung an der Datenstruktur an alle Microservices weitergegeben werden und sie müssen neu deployt werden. Außerdem ist wegen BOUNDED CONTEXT eine einheitliche Modellierung einer Datenstruktur wie Kunde sowieso kaum möglich.

Code übertragen

Eine andere Möglichkeit für eine Änderung an der Architektur ist es, Code von einem Microservice in einen anderen zu verschieben. Das ist sinnvoll, wenn dadurch die lose Kopplung und die hohe Kohäsion des Gesamtsystems sichergestellt werden können. Wenn zwei Microservices viel miteinander kommunizieren, ist die lose Kopplung nicht gewährleistet. Wenn der Teil des Microservice verschoben wird, der viel mit dem anderen Microservice kommuniziert, kann so das Problem gelöst werden.

Der Ansatz ist dem Auslagern in eine gemeinsame Bibliothek ähnlich. Aber der Code ist keine gemeinsame Abhängigkeit, was das Problem der Kopplung zwischen den Microservices löst. Allerdings kann es sein, dass die Microservices eine Schnittstelle untereinander haben müssen, damit beide auch nach der Übertragung des Codes die Funktionalitäten noch nutzen können. Das ist eine Blackbox-Abhängigkeit: Nur die Schnittstelle muss bekannt sein und nicht die internen Code-Strukturen.

Es ist auch möglich, den Code in den anderen Microservice zu übertragen und im ursprünglichen Microservice beizubehalten. Das führt zu Redundanzen. Fehler müssen dann in beiden Versionen behoben werden. Und die beiden Versionen können sich in

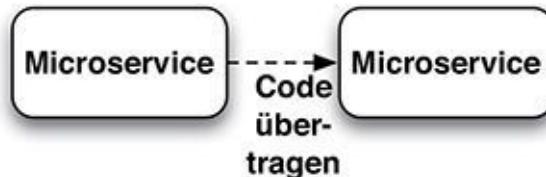
unterschiedliche Richtungen weiterentwickeln. Aber dafür sind die Microservices unabhängig, insbesondere beim Deployment.

Die technologischen Einschränkungen sind immer noch dieselben – die beiden Microservices müssen ähnliche Technologien nutzen, weil sonst der Code nicht übertragen werden kann. Allerdings kann der Code zur Not auch in einer anderen Programmiersprache oder mit einem anderen Programmiermodell neu geschrieben werden. Microservices sind nicht besonders groß. Der neu zu schreibende Code ist nur ein Teil eines Microservice. Also ist der Aufwand überschaubar.

Ein Problem ist allerdings, dass die Größe des Microservice zunimmt, in den der Code verschoben wird. So entsteht die Gefahr, dass der Microservice langfristig ein Monolith wird.

Ein Beispiel: Der Microservice für den Bestellvorgang ruft den Rechnung-Microservice häufig auf, um die Preise für den Versand zu berechnen. Beide Services sind in derselben Programmiersprache geschrieben. Der Code wird von dem einen Microservice in den anderen verlagert. Fachlich stellt sich heraus, dass die Berechnung der Versandkosten auch eher in den Bestellvorgang-Microservice gehört. Das Verschieben ist nur möglich, wenn die beiden Services dieselbe Plattform und Programmiersprache nutzen. Außerdem muss die Kommunikation über Microservice-Grenzen hinweg durch lokale Kommunikation ersetzt werden.

Abb. 8–5 Code übertragen



Statt gemeinsam genutzten Code dem einen oder anderen Microservice zuzuschlagen, kann der Code auch in beiden Microservices weitergepflegt werden. Das hört sich zunächst einmal gefährlich an – schließlich ist der Code dann redundant an zwei Stellen und Bug Fixes müssen dementsprechend auch an mehreren Stellen vorgenommen werden. Meistens versuchen Entwickler, solche Situationen zu verhindern. Eine etablierte Best Practice ist »Don't Repeat Yourself« (DRY). Jede Entscheidung und damit jeder Code sollen nur an genau einer Stelle im System abgelegt sein. In einer Microservices-Architektur hat Redundanz aber einen entscheidenden Vorteil: Die beiden Microservices sind unabhängig voneinander und können unabhängig voneinander deployt und weiterentwickelt werden. Dadurch wird diese zentrale Eigenschaft der Microservices bewahrt.

Außerdem ist es fraglich, ob Systeme vollständig redundanzfrei aufgebaut werden können. Viele Projekte vor allem in der Anfangszeit der Objektorientierung haben viel Aufwand investiert, um gemeinsamen Code in Frameworks und Bibliotheken auszulagern. Das sollte den Aufwand bei der Erstellung der einzelnen Projekte reduzieren. In der Realität war der Code, der wiederverwendet werden soll, oft schwer zu verstehen und zu nutzen. Eine redundante Implementierung in den verschiedenen Projekten wäre vielleicht die bessere Alternative gewesen. Es kann weniger aufwendig sein, den Code mehrmals zu implementieren, als ihn wiederverwendbar zu gestalten und dann

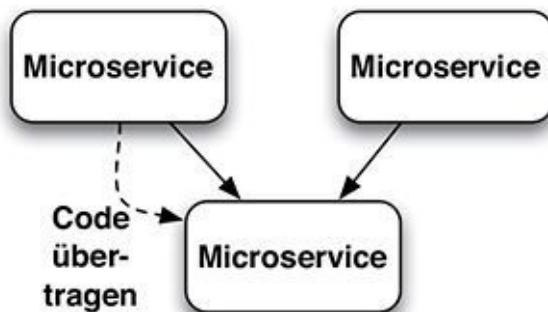
wiederzuverwenden.

Erfolgreiche Wiederverwendung von Code gibt es natürlich: Kaum ein Projekt kommt heute ohne Open-Source-Bibliotheken aus. Auf dieser Basis findet Code-Wiederverwendung ständig statt. Dieser Ansatz kann eine gute Vorlage für die Wiederverwendung von Code zwischen Microservices sein. Das hat allerdings Auswirkungen auf die Organisation. Der [Abschnitt 13.7](#) betrachtet die Organisation und damit auch die Wiederverwendung von Code nach einem Open-Source-Modell.

Gemeinsamer Service

Statt den Code in eine Bibliothek auszulagern, kann er auch in einen neuen Microservice überführt werden (siehe [Abb. 8–6](#)). Dadurch ergeben sich die typischen Vorteile einer Microservices-Architektur: Die Technologie des neuen Microservice ist egal. Solange er die allgemein definierten Kommunikationstechnologien nutzt und wie die anderen Microservices betrieben werden kann, kann der interne Aufbau beliebig sein – bis hin zur Programmiersprache.

Abb. 8–6 Gemeinsame Microservices



Die Verwendung des Microservice ist einfacher als die einer Bibliothek. Nur die Schnittstelle des Microservice muss bekannt sein – der interne Aufbau ist egal. Das Auslagern von Code in einen neuen Service verringert die durchschnittliche Größe eines Microservice – und damit wird die Verständlichkeit und Ersetzbarkeit der Microservices verbessert. Aber die Auslagerung ersetzt lokale Aufrufe durch Aufrufe über das Netzwerk. Änderungen für neue Features sind gegebenenfalls nicht mehr auf einen Microservice begrenzt.

In der Software-Entwicklung sind große Module oft ein Problem, sodass die Auslagerung in neue Microservices sinnvoll sein kann, um die Module klein zu halten. Außerdem kann der neue Microservice vom Team weiterentwickelt werden, das für den ursprünglichen Microservice zuständig war. So wird eine enge Abstimmung des neuen und alten Microservices vereinfacht, weil die Kommunikation dazu nur in einem Team stattfindet.

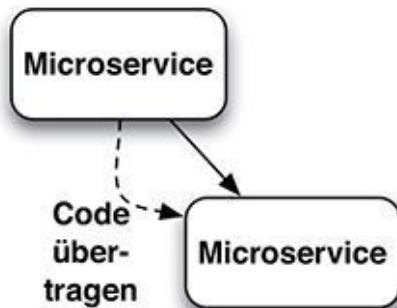
Die Aufteilung der Microservices führt auch dazu, dass eine Anfrage an das Microservice-System nicht durch einen einzigen Microservice bearbeitet wird, sondern mehrere Microservices. Sie rufen sich gegenseitig auf. Einige Microservices haben dann auch keine GUI, sondern sind reine Backend-Services.

Als Beispiel soll wieder der Bestellvorgang dienen, der den Rechnung-Microservice häufig auruft, um die Versandkosten zu berechnen. Die Berechnung der Versandkosten

kann in einen eigenen Microservice separiert werden. Das ist möglich, selbst wenn der Rechnungsservice und der Bestellvorgang-Microservice unterschiedliche Plattformen und Technologien nutzen. Allerdings muss eine neue Schnittstelle etabliert werden, die den neuen Versandkosten-Microservice mit dem Rest des Rechnung-Service kommunizieren lässt.

Neuen Microservice erzeugen

Abb. 8–7 Neuen Microservice erzeugen



Es ist es auch möglich, einen Teil des Codes aus einem Microservice zu nutzen, um einen neuen Microservice zu erstellen (siehe Abb. 8–7). Die Vor- und Nachteile sind identisch mit dem Szenario, in dem der Code in einen gemeinsam genutzten Microservice verlagert wird. Aber die Motivation ist in diesem Fall eine andere: Die Größe der Microservices soll reduziert werden, um die Wartbarkeit zu erhöhen oder vielleicht, um die Zuständigkeit für eine bestimmte Funktionalität an ein anderes Team zu übergeben. Der neue Microservice soll aber nicht gemeinsam von mehreren Microservices genutzt werden.

Zum Beispiel kann der Service für die Registrierung mittlerweile zu komplex geworden sein. Er wird in mehrere Services aufgeteilt, die jeweils bestimmte Benutzergruppen behandeln. Auch eine technische Aufteilung ist denkbar – zum Beispiel nach CQRS (siehe Abschnitt 10.2), Event Sourcing (Abschnitt 10.3) oder in hexagonale Architektur (Abschnitt 10.4).

Neu schreiben

Schließlich ist eine weitere Möglichkeit zum Umgang mit Microservices, deren Strukturierung nicht mehr passt, ein Neuschreiben dieser Microservices. Durch die geringe Größe der Microservices und wegen der Nutzung durch definierte Schnittstellen ist dieser Ansatz viel einfacher, als es in anderen Architekturansätzen der Fall ist. Denn es muss nicht das gesamte System neu geschrieben werden, sondern nur ein Teil. Es ist auch möglich, den neuen Microservice in einer anderen Programmiersprache umzusetzen, die vielleicht besser geeignet ist. Das Neuschreiben der Microservices kann auch vorteilhaft sein, weil so neue Erkenntnisse über die Domäne in die neue Implementierung einfließen können.

Die Erfahrung mit Microservice-Systemen zeigt, dass über die Projektlaufzeit ständig neue Microservices entstehen. Das bedeutet einen höheren Aufwand für die Infrastruktur und für den Betrieb des Systems. Die Anzahl der deployten Services steigt ständig. Eine solche Entwicklung ist für klassische Projekte ungewöhnlich und scheint daher problematisch. Wie dieser

Die Anzahl der Microservices steigt.

Abschnitt aber gezeigt hat, ist das Anlegen neuer Microservices für die gemeinsame Verwendung von Logik und für die Weiterentwicklung eines Systems die beste Alternative. Außerdem bleibt durch die wachsende Zahl die mittlere Größe eines Microservice konstant und damit bleiben die positiven Eigenschaften der Microservices erhalten.

Das Anlegen neuer Microservices sollte möglichst einfach sein, denn so können die Eigenschaften der Microservice-Systeme erhalten bleiben. Optimierungspotenziale gibt es vor allem beim Etablieren der Continuous-Delivery-Pipelines, der Build-Infrastruktur und der benötigten Server für einen neuen Microservice. Wenn das automatisiert möglich ist, können neue Microservices recht problemlos angelegt werden.

Microservice-Systeme sind schwer änderbar

Dieser Abschnitt hat gezeigt, dass es schwierig ist, die Architektur eines Microservice-Systems anzupassen. Es müssen neue Microservices erstellt werden, was Änderungen in der Infrastruktur und zusätzliche Continuous-Deployment-Pipelines bedeutet. Gemeinsamer Code in Bibliotheken ist kaum eine sinnvolle Option.

In einem Deployment-Monolithen wären solche Änderungen schnell gemacht: Oft automatisieren sogar die integrierten Entwicklungsumgebungen das Verschieben von Code oder andere Strukturänderungen. Durch die Automatisierung sind die Änderungen weniger aufwendig und weniger fehleranfällig. Auswirkungen in die Infrastruktur oder die Continuous-Delivery-Pipelines gibt es beim Deployment-Monolithen überhaupt nicht.

Auf Ebene des gesamten Systems sind Änderungen also schwierig – weil Funktionalitäten zwischen den Microservices nur schwer verlagert werden können. Letztendlich ist das genau der Effekt, den [Abschnitt 1.2](#) als »Starke Modularisierung« bezeichnet und als Vorteil aufgeführt hat: Das Überschreiten der Grenzen zwischen Microservices ist schwierig, sodass die Architektur auf der Ebene zwischen den Microservices auch längerfristig erhalten bleibt. Daraus folgt aber auch, dass die Architektur auf dieser Ebene schwer anpassbar ist.

Selber ausprobieren und experimentieren

Ein Entwickler hat eine Hilfsklasse geschrieben, die den Umgang mit einem Logging-Framework vereinfacht, das auch andere Teams nutzen. Sie ist nicht besonders groß und komplex. Sollte sie von anderen Teams verwendet werden? Sollte aus der Hilfsklasse eine Bibliothek werden, ein eigener Microservice oder soll der Code einfach kopiert werden?

8.4 Microservice-Systeme weiterentwickeln

Microservices haben vor allem Vorteile in sehr dynamischen Umgebungen. Durch das unabhängige Deployment der Microservices können Teams an verschiedenen Features parallel ohne große Koordination arbeiten. Das ist vor allem von Vorteil, wenn unklar ist, welche Features wirklich sinnvoll sind und nur durch Experimente am Markt die

erfolgversprechenden Ansätze herausgefunden werden können.

Gerade in einem solchen Umfeld ist es kaum möglich, am Anfang eine gute Aufteilung der Domäne in Microservices zu planen. Die Architektur muss sich den Gegebenheiten anpassen.

Architektur planen?

Für Microservices bietet sich folgendes Bild:

- Die Aufteilung nach fachlichen Gesichtspunkten ist noch wichtiger, als dies bei einer klassischen Architektur der Fall wäre. Denn die fachliche Aufteilung beeinflusst auch die Aufteilung in Teams und damit das unabhängige Arbeiten der Teams – den zentralen Vorteil von Microservices ([Abschnitt 8.1](#)).
- [Abschnitt 8.2](#) hat gezeigt, dass Werkzeuge für das Architekturmanagement in Microservices-Architekturen nicht ohne Weiteres genutzt werden können.
- Wie [Abschnitt 8.3](#) gezeigt hat, ist die Anpassung der Architektur von Microservices schwierig – vor allem im Vergleich zu Deployment-Monolithen.
- Microservices sind vor allem in dynamischen Umgebungen von Vorteil – wo es noch schwieriger ist, eine sinnvolle Architektur gleich am Anfang festzulegen.

Die Architektur muss änderbar sein, aber das ist wegen der technischen Gegebenheiten schwierig. Dieser Abschnitt zeigt, wie dennoch die Architektur eines Microservice-Systems schrittweise weiterentwickelt werden kann.

Eine Möglichkeit, mit diesem Dilemma umzugehen, ist Start Big es, mit einigen großen Systemen zu starten und dann schrittweise das System in Microservices zu zerlegen. [Abschnitt 4.1](#) hat als obere Grenze der Größe eines Microservice die Codemenge angegeben, die ein Team noch handhaben kann. Zumindest am Anfang eines Projekts kann man kaum gegen diese obere Schwelle verstossen. Dasselbe gilt für die anderen oberen Schwellen: die Modularisierung und die Ersetzbarkeit.

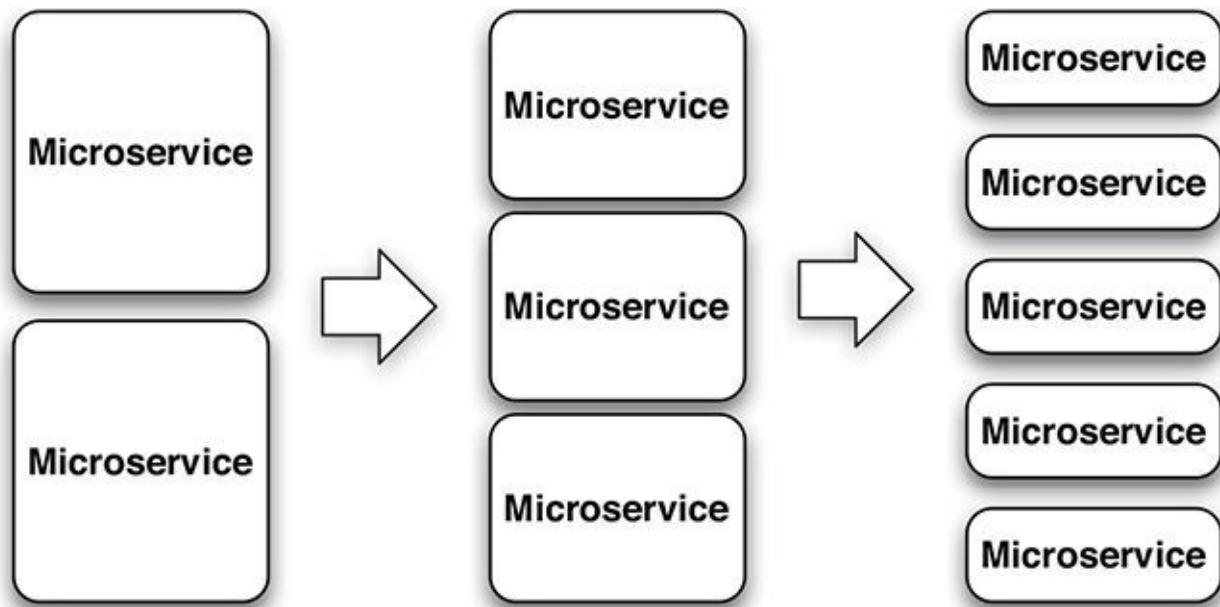
Wenn das gesamte Projekt nur aus einem oder einigen wenigen Services besteht, ist es noch einfach, Funktionalitäten zu verlagern, denn die Verschiebung wird meistens innerhalb eines Service stattfinden und nicht zwischen Services. Schrittweise können dann mehr Personen in das Projekt gezogen werden. Dann können weitere Teams aufgebaut werden. Parallel kann das System in immer mehr Microservices aufgeteilt werden, damit die Teams unabhängig voneinander arbeiten können. Ein solcher Ramp-up ist auch aus organisatorischen Gründen sinnvoll, da so die Teams schrittweise aufgebaut werden können.

Natürlich könnte man auch mit einem Deployment-Monolithen beginnen. Der Start mit einem Monolithen hat aber einen entscheidenden Nachteil: Es besteht die Gefahr, dass sich Abhängigkeiten und Probleme in die Architektur einschleichen, die eine spätere Aufteilung in Microservices unmöglich machen. Außerdem gibt es dann nur eine Continuous-Delivery-Pipeline. Wenn der Monolith in Microservices aufgeteilt wird, müssen die Teams neue Continuous-Delivery-Pipelines erstellen. Das kann aufwendig sein, vor allem wenn die Continuous-Delivery-Pipeline für den Deployment-Monolithen manuell erstellt worden ist. Dann müssten nämlich alle weiteren Continuous-Delivery-Pipelines ebenfalls aufwendig manuell erstellt werden.

Wenn das Projekt gleich mit mehreren Microservices beginnt, umgeht man das Problem. Der Monolith muss nicht später aufgeteilt werden und einen Ansatz für das Erzeugen neuer Continuous-Delivery-Pipelines muss es auch geben. Die Teams können so von Anfang an unabhängig voneinander an eigenen Microservices arbeiten. Im Laufe des Projekts werden die anfänglichen Microservices in weitere kleinere Microservices aufgeteilt.

Start Big entspricht der Beobachtung, dass die Anzahl der Microservices über die Projektlaufzeit steigen wird. Dazu passt es, mit wenigen großen Microservices anzufangen und dann schrittweise neue Microservices abzuspalten. So können jeweils die aktuellen Erkenntnisse in die Aufteilung in Microservices einfließen. Es ist nicht möglich, die perfekte Architektur am Anfang zu definieren. Stattdessen sollten die Teams die Architektur schrittweise den Erkenntnissen anpassen und auch den Mut haben, die notwendigen Änderungen umzusetzen.

Abb. 8–8 Start Big: Aus wenigen Microservices entstehen zunehmend mehr Microservices.



Dieser Ansatz erzeugt einen einheitlichen Technologie-Stack – was Betrieb und Deployment vereinfacht. Entwickler können auch einfacher an anderen Microservices arbeiten.

Es ist auch denkbar, mit einer Aufteilung in eine große Zahl Microservices zu starten und diese Aufteilung als Basis für die Entwicklung zu nutzen. Allerdings ist die Aufteilung der Services sehr schwierig. [4] gibt ein Beispiel, bei dem ein Team ein Werkzeug zur Unterstützung von Continuous Delivery als Microservice-System entwickeln sollte. Das Team kannte die Domäne sehr gut, hatte schon Produkte in diesem Umfeld gebaut und hat daher eine Architektur gewählt, die das System schon frühzeitig in viele Microservices aufgeteilt hat. Da das neue Produkt aber in der Cloud angeboten werden sollte, war die Architektur an einigen Stellen aus subtilen Gründen nicht passend. Änderungen wurden schwierig, weil Anpassungen für Features in mehreren Microservices vorgenommen werden mussten. Als Lösung wurden die Microservices wieder zu einem Monolithen zusammengefasst, um die Software leichter änderbar zu machen. Ein Jahr später hat das Team den Monolithen wieder in Microservices aufgeteilt und damit die endgültige Architektur festgelegt. Dieses

Start Small?

Beispiel zeigt, dass eine vorschnelle Aufteilung in Microservices problematisch sein kann – selbst wenn das Team die Domäne sehr gut versteht.

Aber eigentlich ist das eine Limitierung der Grenzen der Technologie Technologie. Wäre es einfacher, Funktionalitäten zwischen Microservices zu verschieben, könnte die Aufteilung in Microservices korrigiert werden. Dann wäre es auch weniger riskant, mit einer Aufteilung in kleine Microservices loszulegen. Wenn alle Microservices dieselbe Technologie nutzen, ist ein Verschieben von Funktionalitäten einfacher. [Kapitel 15](#) zeigt Technologien für Nanoservices, die einige Kompromisse eingehen, aber dafür kleinere Services und ein leichteres Verschieben von Funktionalitäten erlauben.

Ein Vorteil des Microservices-Ansatzes ist Ersetzbarkeit als Qualitätskriterium Ersetzbarkeit der Microservices. Das geht nur, wenn die Microservices eine bestimmte Größe und interne Komplexität nicht überschreiten. Ein Ziel bei der Weiterentwicklung von Microservices ist, dass Microservices ersetzbar bleiben. So kann ein Microservice durch eine andere Implementierung ersetzt werden – wenn beispielsweise die Weiterentwicklung wegen der schlechten Struktur des Microservice nicht mehr realistisch möglich ist. Ersetzbarkeit als Ziel ist aber auch sinnvoll, um die Verständlichkeit und Wartbarkeit des Microservice zu erhalten. Wenn der Microservice nicht mehr ersetzt werden kann, ist er vermutlich auch kaum noch zu verstehen und daher auch kaum noch weiterentwickelbar.

Ein Problem ist, dass große Services Änderungen und neue Features anziehen. Sie decken schon viele Features ab. Daher liegt es nahe, neue Features auch in diesen Services umzusetzen. Das gilt für zu große Microservices, aber noch mehr für Deployment-Monolithen. Ziel einer Microservices-Architektur kann sein, einen Monolithen abzulösen. Da der Monolith aber so viele Funktionalitäten hat, muss man aufpassen, dass nicht zu viele Änderungen in dem Monolithen vorgenommen werden. Microservices können dazu neu erstellt werden, auch wenn sie zunächst kaum Funktionalitäten enthalten. Einfach Änderungen und Erweiterungen an dem Monolithen vorzunehmen, ist genau das Vorgehen, das den Deployment-Monolithen unvorbereitet gemacht und zu einer Ablösung durch Microservices geführt hat.

Wie schon erwähnt, ist das Problem einer Architektur meistens nicht, dass sie ursprünglich so geplant wurde, dass sie nicht auf das Problem passt. Meistens ist das Problem, dass die Architektur nicht mit der Änderung an der Umgebung Schritt gehalten hat. Auch eine Microservices-Architektur muss ständig angepasst werden, sonst wird sie früher oder später den Anforderungen nicht mehr gerecht werden. Dazu zählt ein Management der fachlichen Aufteilung, aber auch der Größe der einzelnen Microservices. Nur so kann sichergestellt werden, dass die Vorteile der Microservices-Architektur langfristig erhalten bleiben. Da die Code-Menge eines Systems üblicherweise wächst, wird die Anzahl der Microservices auch wachsen, um die durchschnittliche Größe konstant zu halten. Eine Zunahme der Anzahl der Microservices ist kein Problem, sondern ein gutes Zeichen.

Aber nicht nur die Größe der Microservices kann ein Problem sein. Auch die Abhängigkeiten der Globale Architektur?

Microservices können zu Problemen führen (siehe [Abschnitt 8.1](#)). Solche Probleme können meistens gelöst werden, indem einige Microservices angepasst werden – nämlich jene, die problematische Abhängigkeiten haben. Dazu müssen nur die Teams involviert sein, die an diesen Microservices arbeiten. Die Teams können solche Probleme erkennen, denn sie haben durch schlechte Architektur und größere Koordination auch die Probleme. Durch Änderungen an der Architektur können sie die Probleme auch lösen. Dann ist ein globales Management der Abhängigkeiten nicht notwendig. Metriken wie eine hohe Anzahl von Abhängigkeiten oder zyklische Abhängigkeiten können nur ein Indiz sein, dass es Probleme gibt. Ob solche Metriken tatsächlich auf ein Problem hinweisen, kann nur geklärt werden, wenn die Metriken zusammen mit den beteiligten Teams bewertet werden. Werden die problematischen Bestandteile beispielsweise in Zukunft nicht mehr weiterentwickelt, ist es egal, ob die Metriken auf ein Problem hinweisen. Oder vielleicht sind aus anderen Gründen bei der Entwicklung nie Probleme aufgetreten. Selbst wenn es ein globales Management der Architektur gibt, kann es nur in enger Abstimmung mit den Teams effektiv arbeiten.

Verpasste nicht den Absprung oder: Wie vermeidet man die Erosion eines Microservice?

von Lars Gentsch, E-Post Development GmbH

Einen Microservice zu entwickeln, stellt sich in der Praxis nicht als allzu schwierig dar. Wie schaffe ich es aber, dass ein Microservice ein Microservice bleibt und nicht klamm und heimlich zu einen Monolithen wächst? An einem Beispiel soll beleuchtet werden, wann sich ein Service in die falsche Richtung entwickelt und welche Maßnahmen möglich sind, sodass der Microservice ein Microservice bleibt.

Stellen wir uns eine kleine Webanwendung zur Registrierung eines Nutzers vor. Dieses Szenario findet sich in fast jeder Webanwendung wieder. Ein Nutzer möchte ein Produkt in einem Internetshop kaufen (Amazon, Otto etc.) oder sich für ein Video-on-Demand-Portal anmelden (Watchever, Netflix usw.). Dafür wird der Nutzer durch einen kleinen Registrierungs-Workflow geführt. Er wird nach einem Nutzernamen, einem Passwort, der E-Mail-Adresse und seiner Anschrift gefragt. Dies ist eine kleine abgeschlossene Funktionalität, die sich wunderbar für einen Microservice eignet.

Technologisch ist dieser Service wahrscheinlich simpel aufgebaut. Er besteht aus zwei bis drei HTML-Seiten oder einer AngularJS-Single-Page-App, etwas CSS, evtl. Spring Boot und einer MySQL-Datenbank. Gebaut wird das Entwicklungsprojekt mit Maven.

Während der Erfassung der Daten werden diese bei der Eingabe validiert, in das Domain-Modell überführt und in der Datenbankpersistiert. Wie wird aus dem Microservice Schritt für Schritt ein Monolith?

Einbindung neuer Funktionalität

Über den Shop oder das Video-on-Demand-Portal sollen Waren und Inhalte geliefert werden, auf die nur volljährige Personen Zugriff haben sollen. Dazu muss nun das Alter des Kunden verifiziert werden. Einer der Wege, dies zu tun, ist, das Geburtsdatum des Kunden und weitere Daten zu erfassen und einen externen Dienst

zur Altersverifikation einzubinden.

Das Daten-Modell unseres Service ist also um das Geburtsdatum zu erweitern. Spannender ist die Einbindung des externen Dienstes. Hierzu muss ein Client für eine externe API geschrieben werden, der auch mit Fehlersituationen, wie der Nichtverfügbarkeit des Anbieters, umgehen können sollte.

Sehr wahrscheinlich ist das Anstoßen der Altersverifikation ein asynchroner Prozess, sodass unser Dienst gezwungen sein könnte, eine Callback-Schnittstelle zu implementieren. Die Haltung von Prozessdaten wird nötig. Wann wurde der Altersverifikationsprozess angestoßen? Ist eine Erinnerung des Kunden per E-Mail nötig? Wurde der Verifikationsprozess erfolgreich abgeschlossen?

Was passiert hier mit dem Microservice?

1. Die Kundendaten werden um das Geburtsdatum erweitert. Das ist nicht problematisch.
2. Zu den Kundendaten kommen jetzt Prozessdaten. Achtung: Hier werden Verlaufsdaten mit den Domain-Daten gemischt.
3. Zusätzlich zu der ursprünglichen CRUD-Funktionalität des Service wird jetzt eine Art Workflow benötigt. Synchrone Verarbeitung wird mit asynchroner Verarbeitung gemischt.
4. Ein Drittssystem wird eingebunden. Der Testaufwand für den Registrierung-Microservice steigt. Ein zusätzliches System und dessen Verhalten müssen während des Tests simuliert werden.
5. Die asynchrone Kommunikation mit dem Drittssystem hat andere Skalierungsanforderungen. Während vom Registrierung-Microservice angenommen zehn Instanzen aufgrund der Last und Ausfallsicherheit benötigt werden, kann die Anbindung der Altersverifikation mit zwei Instanzen ausfallsicher und stabil betrieben werden. Hier werden also unterschiedliche Laufzeitanforderungen gemischt.

Wie an dem Beispiel zu sehen ist, kann eine an sich kleine Anforderung wie die Anbindung einer Altersverifikation massive Auswirkungen auf die Größe des Microservice haben.

Kriterien für einen neuen statt der Erweiterung eines bestehenden Service:

1. Einführung unterschiedlicher Datenmodelle und Daten (Domain vs. Prozessdaten)
2. Mischung von synchroner und asynchroner Datenverarbeitung
3. Anbindung von zusätzlichen Diensten
4. Unterschiedliche Lastszenarien für einzelne Aspekte innerhalb eines Service

Das Beispiel des Registration-Service ließe sich erweitern um den Aspekt der Verifikation der postalischen Adresse des Kunden über einen externen Anbieter. Dies ist üblich, um die physische Existenz der angegebenen Adresse sicherzustellen; oder

die manuelle Freigabe oder Überprüfung von Kunden im Falle einer Doppelregistrierung. Ebenfalls nicht selten ist die Einbindung einer Bonitätsprüfung oder Scoring des Kunden bei Registrierung.

All diese fachlichen Aspekte gehören prinzipiell zur Registrierung eines Kunden und führen jeden Entwickler und Architekten in die Versuchung, diese Anforderungen in den bestehenden Microservice zu integrieren und damit aus dem Microservice mehr als nur einen Microservice werden zu lassen.

Woran ist zu erkennen, wenn der Absprung zu einem neuen Microservice nicht geschafft wurde?

1. Der Service lässt sich nur noch als Maven-Multi-Modul-Projekt oder Gradle-Multi-Modul-Projekt sinnvoll weiter entwickeln.
2. Die Tests müssen in Testgruppen aufgeteilt und bei der Ausführung parallelisiert werden, da die Testlaufzeit über fünf Minuten liegt. (Verletzung des »Fast-Feedback«-Prinzips)
3. Die Konfiguration des Dienstes wird innerhalb der Konfigurationsdatei nach Fachlichkeit gruppiert oder die Datei wird in Einzelkonfigurationsdateien aufgeteilt, um eine bessere Übersichtlichkeit herzustellen.
4. Ein kompletter Build des Service dauert lange genug für eine Kaffeepause. Es sind keine schnellen Feedback-Zyklen mehr möglich. (Verletzung des »Fast-Feedback«-Prinzips)

Fazit

Wie das Beispiel eines Registrierung-Microservice zeigt, besteht die große Herausforderung darin, einen Microservice als Microservice zu lassen und nicht der Versuchung zu erliegen, neue Funktionalitäten aufgrund von Zeitdruck in einen bestehenden Service zu integrieren. Das gilt sogar, wenn die Funktionalitäten, wie in diesem Beispiel, eindeutig derselben fachlichen Domäne angehören.

Was kann vorbeugend getan werden, um die Erosion des Microservice zu verhindern? Prinzipiell muss es so einfach wie möglich sein, neue Services inklusive eigener Datenhaltung zu kreieren. Frameworks wie Spring Boot, Grails und Play leisten dazu einen sinnvollen Beitrag. Die Bereitstellung von Projekt-Templates wie Maven-Archetypes und die Verwendung von Container-Deployments mit Docker sind weitere Maßnahmen, um die Erstellung und Konfiguration neuer Microservices sowie deren Weg in die Produktionsumgebung so weit wie möglich zu vereinfachen. Durch Verringerung der »Kosten« zum Aufsetzen eines neuen Service sinkt die Hemmschwelle für die Einrichtung eines neuen Microservice deutlich und damit die Versuchung, neue Funktionalität in bestehende Services zu integrieren.

8.5 Microservice und Legacy-Anwendung

Die Aufteilung einer Legacy-Anwendung in eine Microservices-Architektur ist ein in der

Praxis sehr häufig anzutreffendes Szenario. Komplette Neuentwicklungen sind eher selten und Microservices versprechen vor allem Vorteile bei der langfristigen Wartung. Das ist besonders für Anwendungen interessant, die bereits kaum noch wartbar sind. Außerdem erlaubt eine Aufteilung in Microservices einen einfacheren Umgang mit Continuous Delivery: Statt einen Monolithen automatisiert zu deployen und zu testen, können kleine Microservices deployt und getestet werden. Der Aufwand dafür ist wesentlich geringer. Eine Continuous-Delivery-Pipeline für einen Microservice ist nicht besonders komplex – für einen Deployment-Monolithen hingegen kann der Aufwand sehr groß sein. Dieser Vorteil ist für viele Unternehmen ausreichend, um den Aufwand für eine Migration hin zu Microservices zu rechtfertigen.

Gegenüber komplett neuen Systemen gibt es bei der Migration vom Deployment-Monolithen hin zu Microservices einige entscheidende Unterschiede:

- Bei einem Legacy-System ist die Funktionalität fachlich klar. Das kann eine gute Basis sein, um eine saubere fachliche Architektur für Microservices zu erstellen. Gerade eine saubere fachliche Aufteilung ist für Microservices sehr wichtig.
- Aber es ist schon eine große Menge Code vorhanden. Der Code ist oft von schlechter Qualität, es gibt wenig Tests und die Deployment-Zeiten sind oft viel zu lang. Microservices sollen diese Probleme beseitigen. Dementsprechend sind die Herausforderungen in diesem Bereich oft signifikant.
- Ebenso kann es gut sein, dass die Modulgrenzen in der Legacy-Anwendung nicht der BOUNDED-CONTEXT-Idee (siehe [Abschnitt 4.3](#)) entsprechen. Dann ist eine Migration hin zu einer Microservices-Architektur eine Herausforderung, weil der fachliche Schnitt der Anwendung geändert werden muss.

Ein einfacher Ansatz ist es, den Code der Legacy-Code aufbrechen? Anwendung auf mehrere Microservices aufzuteilen. Das kann problematisch sein, wenn die Legacy-Anwendung wie so oft keine gute fachliche Aufteilung hat. Der Code lässt sich besonders einfach auf Microservices aufteilen, wenn sich die Microservices an den vorhandenen Modulen in der Legacy-Anwendung orientieren. Die haben aber eine schlechte fachliche Aufteilung, die dann in die Microservices-Architektur übernommen wird. Die Konsequenzen des schlechten Schnitts sind in einer Microservices-Architektur noch schwerwiegender: Der Schnitt beeinflusst auch die Kommunikation zwischen den Teams. Außerdem ist er in einer Microservices-Architektur nur schwer anpassbar.

Legacy-Anwendung ergänzen

Aber man kann auch ohne eine Aufteilung der Legacy-Anwendung auskommen. Ein wesentlicher Vorteil von Microservices ist, dass die Module verteilte Systeme sind. Dadurch sind die Modulgrenzen auch Grenzen von Prozessen, die über das Netz miteinander kommunizieren. Für die Aufteilung einer Legacy-Anwendung hat das Vorteile: Es ist gar nicht notwendig, die internen Strukturen der Legacy-Anwendung zu kennen oder auf dieser Basis eine Aufteilung in Microservices vorzunehmen. Stattdessen können Microservices die Legacy-Anwendungen an der Schnittstelle ergänzen oder ändern. Dazu ist es vor allem hilfreich, wenn das abzulösende System schon in einer SOA

(Abschnitt 7.2) aufgebaut ist. Wenn es einzelne Services gibt, können diese durch Microservices ergänzt werden.

Eine Inspiration zu möglichen Integrationen von Legacy-Anwendungen und Microservices bieten die Enterprise Integration Patterns [6][7]:

Enterprise Integration Patterns

- MESSAGE ROUTER beschreibt, dass bestimmte Nachrichten an einen anderen Service gehen. So kann ein Microservice einige Nachrichten auswählen, die der Microservice bearbeitet und nicht mehr die Legacy-Anwendung. Dadurch muss die Microservices-Architektur nicht sofort die gesamte Logik neu implementieren, sondern kann zunächst einige Aspekte auswählen.
- Ein besonderer Router ist der CONTENT BASED ROUTER. Er entscheidet anhand des Inhalts der Nachricht, wohin die Nachricht geschickt werden soll. So können spezielle Nachrichten an einen bestimmten Microservice geschickt werden – selbst wenn die Nachricht sich nur in einem Feld unterscheidet.
- Der MESSAGE FILTER vermeidet, dass ein Microservice uninteressante Nachrichten erhält. Dazu filtert er einfach alle Nachrichten aus, die den Microservice nicht erreichen sollen.
- Ein MESSAGE TRANSLATOR übersetzt eine Nachricht in ein anderes Format. So kann die Microservices-Architektur andere Datenformate verwenden und muss nicht unbedingt die Formate der Legacy-Anwendung nutzen.
- Durch einen CONTENT ENRICHER werden Daten in den Nachrichten ergänzt. Wenn ein Microservice zusätzlich zu den Daten der Legacy-Anwendung weitere Informationen benötigt, kann der CONTENT ENRICHEN diese Informationen einfügen, ohne dass die Legacy-Anwendung oder der Microservice davon etwas mitbekommen.
- Der CONTENT FILTER erreicht das Gegenteil: Bestimmte Daten werden aus den Nachrichten entfernt, sodass der Microservice nur die Informationen bekommt, die für ihn relevant sind.

Abb. 8–9 Legacy-Anwendung mit MESSAGE ROUTER ergänzen

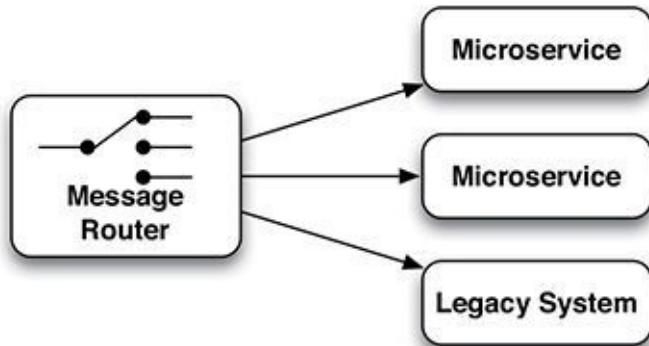


Abbildung 8–9 zeigt ein einfaches Beispiel: Ein MESSAGE ROUTER nimmt Anfragen entgegen und schickt sie an einen Microservice oder das Legacy-System. So können bestimmte Funktionalitäten in den Microservices umgesetzt werden. Diese Funktionalitäten sind immer noch im Legacy-System – werden dort aber nicht mehr genutzt. So sind die Microservices weitgehend unabhängig von den Strukturen innerhalb des Legacy-Systems. Beispielsweise können Microservices zunächst Bestellungen für

bestimmte Kunden oder bestimmte Waren bearbeiten. Dadurch müssen die Microservices nicht alle Sonderfälle implementieren.

Die Patterns können als Inspiration dienen, wie eine Legacy-Anwendung mit Microservices ergänzt werden kann. Es gibt noch zahlreiche weitere Patterns – die Auflistung ist nur ein Ausschnitt des gesamten Katalogs. Wie sonst auch können die Patterns unterschiedlich umgesetzt werden: Eigentlich fokussieren sie auf Messaging-Systeme. Es ist aber möglich, sie mit synchronen Kommunikationsmechanismen zu implementieren – wenn auch weniger elegant. So kann beispielsweise ein REST-Service eine POST-Nachricht entgegennehmen, mit weiteren Daten versehen und schließlich an einen andern Microservice weiterschicken. Das wäre dann ein CONTENT ENRICHER.

Um solche Patterns umzusetzen, muss der Sender vom Empfänger entkoppelt sein. So können zusätzliche Schritte in die Verarbeitung von Requests integriert werden, ohne dass der Sender dies bemerkt. Bei einem Messaging-Ansatz ist das ohne Weiteres möglich, da der Sender nur eine Queue kennt, in die er Nachrichten einstellt. Wer die Nachrichten abholt, ist dem Sender nicht bekannt. Bei synchroner Kommunikation per REST oder SOAP wird hingegen die Nachricht direkt an den Empfänger geschickt. Erst durch Service Discovery (siehe [Abschnitt 8.9](#)) wird der Sender vom Empfänger entkoppelt. Dann kann ein Service durch einen anderen Service ersetzt werden, ohne dass die Sender geändert werden müssen. Dann können die Patterns leichter umgesetzt werden. Wenn die Legacy-Anwendung durch einen CONTENT ENRICHER ergänzt wird, wird er statt der Legacy-Anwendung in der Service Discovery angemeldet, aber kein Sender muss modifiziert werden. Service Discovery einzuführen, kann daher ein erster Schritt hin zu einer Microservices-Architektur sein, weil so einzelne Services der Legacy-Anwendung ergänzt oder ersetzt werden können, ohne dass die Nutzer der Legacy-Anwendung modifiziert werden müssen.

Gerade bei Legacy-Anwendungen ist es wichtig, dass die Microservices nicht zu sehr von der Legacy-Anwendung abhängen. Oft ist gerade die schlechte Struktur der alten Anwendung der Grund, warum die Anwendung abgelöst werden soll. Bestimmte Abhängigkeiten sollten daher überhaupt nicht erlaubt sein. Wenn Microservices auf die Datenbank der Legacy-Anwendung direkt zugreifen, sind die Microservices von der internen Datenrepräsentation der Legacy-Anwendung abhängig. Außerdem können weder die Legacy-Anwendung noch die Microservices das Schema noch ändern, weil solche Änderungen in Microservices und Legacy-Anwendung umgesetzt werden müssen. Die gemeinsame Nutzung der Datenbank in Legacy-Anwendung und Microservices muss auf jeden Fall vermieden werden. Die Replikation der Daten der Legacy-Anwendung in ein eigenes Datenbankschema ist natürlich dennoch eine Option.

Integration begrenzen

Versteckte Abhängigkeiten

von Oliver Wehrens, E-Post Development GmbH

Am Anfang ist der Monolith. Oft ist es sinnvoll und ergibt sich natürlich, dass Software als Monolith entsteht. Der Code ist übersichtlich und die Business-Domäne ist gerade im Entstehen. Da ist es besser, wenn alles eine gemeinsame Basis hat. Es gibt eine UI, Businesslogik und eine Datenbank. Refactoring ist einfach, Deployment ist leicht und

jeder kann den gesamten Code noch verstehen.

Mit der Zeit wächst die Codemenge und es wird unübersichtlich. Nicht jeder kennt mehr alle Teile des Codes. Das gegebenenfalls erforderliche Kompilieren dauert länger und die Unit- und Integrationstests laden zu einer Kaffeepause ein. Bei einer einigermaßen stabilen Business-Domäne und sehr großer Code-Basis steht bei vielen Projekten zu diesem Zeitpunkt sicher die Option im Raum, die Funktionalität in mehrere Microservices aufzuteilen.

Je nach Phase des Unternehmens und Verständnis des Business/Product Owners werden die notwendigen Arbeiten erledigt. Sourcecode wird aufgeteilt, Continuous-Delivery-Pipelines werden eingerichtet und Server provisioniert. In diesem Schritt werden keine neuen Features entwickelt. Allein die Hoffnung, dass in Zukunft Features schneller und unabhängiger von anderen Teams geschafft werden können, rechtfertigt den nicht unerheblichen Aufwand. Als Techniker ist man sich da sehr sicher, andere Stakeholder müssen oft davon überzeugt werden.

Im Prinzip ist alles erledigt, um in einer besseren Architektur angekommen zu sein. Es existieren verschiedene Teams, die unabhängigen Sourcecode haben. Sie können ihre Software zu jedem Zeitpunkt und unabhängig von anderen Teams live bringen.

Fast.

Die Datenbank

Jeder Entwickler hat eine mehr oder minder starke Affinität zur Datenbank. In meiner Erfahrung wird sie von vielen Entwicklern als notwendiges Übel angesehen, welches etwas umständlich zu refactoren ist. Oft haben sich Tools etabliert, die es für die Entwickler übernehmen, Datenbankstrukturen zu erstellen (z. B. Liquibase oder Flyway im JVM-Bereich). Tools und Libraries (Object Relation Mapper) machen es sehr einfach, Objekte zu persistieren. Ein paar Annotationen später und die Domäne ist in der Datenbank gesichert.

All diese Werkzeuge abstrahieren die Datenbank vom typischen Entwickler, der »nur« seinen Code schreiben möchte. Das führt leider manchmal dazu, dass der Datenbank während des Entwicklungsprozesses keine zu große Beachtung geschenkt wird. So machen zum Beispiel nicht angelegte Indizes das Suchen über die Datenbank langsam. Das fällt im typischen Test, der nicht auf großen Datenmengen arbeitet, nicht auf und geht so in Produktion.

Nehmen wir den fiktiven Fall eines Schuhversandhändlers an. Dort wird ein Service gebraucht, mit dem sich die Nutzer einloggen können. Es entsteht ein User-Service mit den typischen Feldern wie ID, Vorname, Name, Adresse und Passwort. Um dem Nutzer nun Schuhe anzubieten, die ihm passen, soll er nur die Auswahl in seiner Größe angezeigt bekommen. Die Größe wird in der Willkommensmaske erfasst. Was liegt näher, als diese im bereits vorhandenen User-Service abzuspeichern. Alle sind sich sicher: Es sind nutzerbezogene Daten, und das ist die richtige Stelle.

Nun expandiert der Schuhhändler und es kommen noch andere Arten von Kleidung dazu. Kleidergröße, Hemdweite und alle anderen verwandten Daten landen nun auch im User-Service.

Es sind mehrere Teams im Unternehmen beschäftigt. Der Code wird zunehmend komplexer. Es ist der Zeitpunkt, wo der Monolith in fachliche Services aufgeteilt wird. Das Refactoring im Quellcode gelingt gut und auch die Aufteilung ist bald erledigt.

Leider stellt sich heraus, dass einfaches Ändern immer noch nicht möglich ist. Das für Schuhe verantwortliche Team möchte wegen internationaler Expansion verschiedene Währungen akzeptieren und muss die Rechnungsdatenstruktur inklusive Adressformat ändern. Während des Upgrades wird die Datenbank geblockt. Es kann keine Hemdgröße oder Lieblingsfarbe währenddessen geändert werden. Weiterhin werden die Adressdaten in verschiedenen Formularen anderer Services genutzt und können somit nicht ohne Absprache und Aufwand geändert werden. Das Feature kann also nicht zeitnah umgesetzt werden.

So gut der Code auch getrennt ist, die Teams sind versteckt gekoppelt durch die Datenbank. Spalten umbenennen in der User-Service-Datenbank ist fast unmöglich, weil niemand mehr genau weiß, wer welche Spalten benutzt. Also werden in den Teams Workarounds gemacht, sodass entweder Felder mit dem Namen 'Userattribute1' angelegt werden, die dann im Code auf die richtige Beschreibung gemappt werden, oder es werden in den Daten Trennungen geschaffen wie '#Farbe:Blau#Grösse:10'. Niemand außer dem Team weiß, was sich heute hinter 'Userattribute1' verbirgt, und es ist schwer, auf '#Farbe:#Grösse' einen Index anzulegen. Datenbankstruktur und Code werden schlechter lesbar und schwerer wartbar.

Für jeden Software-Entwickler muss es essenziell sein, sich über die Persistierung der Daten Gedanken zu machen. Das heißt: nicht nur über die Datenbankstrukturen, sondern auch, wo welche Daten gespeichert werden. Ist die Tabelle bzw. Datenbank der Platz, an dem diese Daten liegen sollten? Haben sie mit den anderen Daten fachlich (im Sinne von Business-Domäne) zu tun? Um später flexibler zu sein, lohnt es sich hier, jedes Mal genauer hinzuschauen. Typischerweise werden Datenbanken und Tabellen nicht zu oft angelegt. Sie sind aber ein Teil, der später sehr schwer zu ändern ist und über den sich sehr einfach eine versteckte Koppelung zwischen Services einschleicht. Generell muss gelten, dass Daten nur von genau einem Service mit direktem Datenbankzugriff genutzt werden können. Alle anderen Services, die die Daten nutzen wollen, dürfen nur über die öffentlichen Schnittstellen des Service zugreifen.

Wesentlicher Vorteil eines solchen Ansatzes ist, dass die Microservices weitgehend unabhängig von der Architektur der Legacy-Anwendung sind. Und die Ablösung einer Legacy-Anwendung wird meistens betrieben, weil die Architektur der Legacy-Anwendung nicht mehr tragfähig ist. Außerdem können so Systeme um Microservices ergänzt werden, die eigentlich gar nicht für eine Erweiterung gedacht sind. Beispielsweise sind Standardlösungen im Bereich CRM, E-Commerce oder ERP zwar intern erweiterbar, aber eine Erweiterung durch die externen Schnittstellen kann eine willkommene Alternative sein, weil so eine Ergänzung oft einfacher ist. Außerdem ziehen solche Systeme oft Funktionalitäten an, die gar nicht wirklich in diese Systeme gehören. Eine klare Trennung in eine andere Deployment-Einheit durch einen Microservice stellt eine dauerhafte und klare Abgrenzung sicher.

Integration über UI und Datenreplikation

Allerdings setzt dieser Ansatz nur auf der Ebene der Integration der Logik an. [Kapitel 9](#) hat bereits eine andere Ebene der Integration beschrieben, nämlich Datenreplikation. So kann ein Microservice performant auch auf umfangreiche Datenbestände einer Legacy-Anwendung zugreifen. Wichtig ist, dass die Replikation nicht auf der Basis des Datenmodells der Legacy-Anwendung geschieht. Dann ist das Datenmodell der Legacy-Anwendung nämlich praktisch nicht mehr änderbar, weil auch der Microservice es nutzt. Noch schlechter wäre eine Integration mit Nutzung derselben Datenbank. Auch auf Ebene der UI sind Integrationen möglich. Vor allem Links bei Webanwendungen sind attraktiv, weil sie nur wenige Änderungen in der Legacy-Anwendung verursachen.

Auf diese Weise können beispielsweise auch CMS, die oft sehr viele Funktionalitäten übernehmen, mit Microservices ergänzt werden. CMS (Content Management System) enthalten die Daten einer Website und verwalten den Content, sodass er durch Redakteure geändert werden kann. Die Microservices übernehmen die Behandlung bestimmter URLs. Ähnlich wie bei einem MESSAGE ROUTER kann ein HTTP-Request statt an das CMS an den Microservice geschickt werden. Oder der Microservice ändert Elemente aus dem CMS wie bei einem CONTENT ENRICHER oder modifiziert den Request wie bei einem MESSAGE TRANSLATOR. Schließlich könnten die Microservices Daten in dem CMS speichern und es so als eine Art Datenbank nutzen. Außerdem kann in das CMS JavaScript ausgeliefert werden, das die UI eines Microservice repräsentiert. Dann wird das CMS zu einem Werkzeug zur Auslieferung von Code in einem Browser.

Einige Beispiele könnten sein:

- Ein Microservice kann Content aus bestimmten Quellen importieren. Jede Quelle kann einen eigenen Microservice haben.
- Funktionalität, mit der ein Besucher zum Beispiel einem Autor folgen kann, kann in einen eigenen Microservice ausgelagert werden. Der Microservice kann entweder eine eigene URL haben und durch Links integriert werden. Oder er modifiziert die Seiten, die das CMS ausliefert.
- Während ein Autor noch im CMS bekannt ist, gibt es andere Logik, die vollständig vom CMS getrennt ist. Das können Gutscheine oder E-Commerce-Funktionalitäten sein. Auch in dem Fall kann ein Microservice das System passend ergänzen.

Gerade bei CMS-Systemen, die statisches HTML generieren, können Microservice-Ansätze für dynamischen Content nützlich sein. Das CMS tritt in den Hintergrund und ist nur noch für bestimmten Content notwendig. Es gibt ein monolithisches Deployment des CMS-Content, während die Microservices viel schneller und entkoppelt deployt werden können. Das CMS ist in diesem Zusammenhang wie eine Legacy-Anwendung.

Bewertung

Die Integrationen haben alle den Vorteil, dass die Microservices nicht an die Architektur oder die Technologie-Entscheidungen der Legacy-Anwendung

Integration ohne Übernahme der Architektur

gebunden sind. Das gibt den Microservices einen entscheidenden Vorteil gegenüber der Modifikation der Legacy-Anwendung. Aber die Migration weg von der Legacy-Anwendung mit diesem Ansatz hat auf Ebene der Architektur auch eine Herausforderung: Eigentlich müssen Microservice-Systeme einen sauberen fachlichen Schnitt haben, damit Features möglichst in einem Microservice und von einem Team implementiert werden können. Bei einer Migration mit dem hier skizzierten Ansatz lässt sich das nicht immer realisieren, weil die Migration durch die Schnittstellen der Legacy-Anwendung beeinflusst wird. Daher ist nicht immer ein so sauberer Schnitt möglich wie eigentlich wünschenswert. Außerdem werden fachliche Features noch so lange auch in der Legacy-Anwendung implementiert, bis ein großer Teil der Migration umgesetzt ist. So lange kann die Legacy-Anwendung nicht endgültig beseitigt werden. Wenn die Microservices sich auf das Umwandeln der Nachrichten beschränken, kann die Migration sehr lange dauern.

Die skizzierten Ansätze legen nahe, dass die *Kein Big Bang* vorhandene Legacy-Anwendung schrittweise um Microservices ergänzt oder einzelne Teile der Legacy-Anwendung durch Microservices abgelöst werden. Ein solcher Ansatz hat den Vorteil, dass das Risiko minimiert wird. Die Ablösung der kompletten Legacy-Anwendung in einem einzigen Schritt ist wegen der Größe der Legacy-Anwendung mit einem hohen Risiko verbunden. Schließlich müssen alle Funktionalitäten in den Microservices abgebildet werden. Dabei können sich viele Fehler einschleichen. Ebenso ist das Deployment der Microservices komplex, da sie alle auf einmal in Produktion gebracht werden müssen, um die Legacy-Anwendung abzulösen. Die schrittweise Ablösung zwängt sich bei Microservices nahezu auf, da sie unabhängig deployt werden können und die Legacy-Anwendung ergänzen. So kann die Legacy-Anwendung schrittweise durch Microservices ersetzt werden.

Legacy = Infrastruktur

Ein Teil einer Legacy-Anwendung kann auch einfach als Infrastruktur für Microservices weitergenutzt werden. Beispielsweise kann die Datenbank der Legacy-Anwendung auch für Microservices verwendet werden. Wichtig ist, dass die Schemata der Microservices untereinander und auch von der Legacy-Anwendung getrennt sind. Schließlich sollen die Microservices nicht eng miteinander gekoppelt sein.

Die Nutzung der Datenbank muss für die Microservices nicht verpflichtend werden. Microservices können durchaus andere Lösungen nutzen. Die vorhandene Datenbank ist aber in Bezug auf Betrieb oder Backup etabliert. Eine Nutzung der Datenbank kann auch für die Microservices einen Vorteil darstellen. Ähnliches gilt für andere Infrastruktur-Komponenten. So kann ein CMS (Content Management System) ebenfalls als eine gemeinsame Infrastruktur fungieren, dem aus den verschiedenen Microservices Funktionalitäten hinzugefügt werden und in das die Microservices auch Inhalte ausliefern können.

Die bisher vorgestellten Migrationsansätze fokussieren *Andere Qualitäten* darauf, die fachliche Aufteilung in Microservices zu ermöglichen, um so das System langfristig warten und weiterentwickeln zu können. Aber Microservices haben viele weitere Vorteile. Für eine Migration ist es wichtig zu verstehen, welcher Vorteil die Migration zu Microservices motiviert. Denn abhängig davon kann eine

ganz andere Strategie gewählt werden. Microservices bieten beispielsweise auch eine höhere Robustheit und Resilience, da die Kommunikation mit anderen Diensten entsprechend abgesichert wird (siehe [Abschnitt 10.5](#)). Wenn die Legacy-Anwendung aktuell in diesem Bereich ein Defizit hat oder sogar eine verteilte Architektur besteht, die diesbezüglich optimiert werden muss, können entsprechende Technologien und Architektur-Ansätze definiert werden, ohne dass die Anwendung zwingend in Microservices aufgeteilt werden muss.

Selber ausprobieren und experimentieren

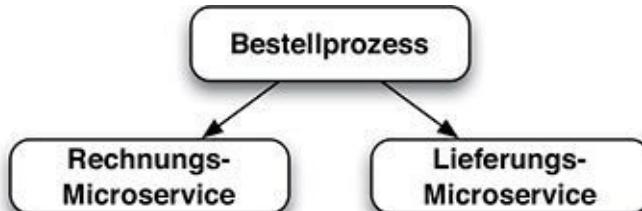
Recherchiere die restlichen Patterns of Enterprise Integration.

- Sind sie bei Microservices sinnvoll einsetzbar? In welchem Kontext?
- Können sie tatsächlich nur mit Messaging-Systemen umgesetzt werden?

8.6 Event-driven Architecture

Damit Microservices gemeinsame Logik implementieren, können sich die Microservices gegenseitig aufrufen. Beispielsweise kann am Ende des Bestellprozesses sowohl der Microservice für die Rechnung wie auch der Microservice für die Ausführung der Bestellung aufgerufen werden, damit die Rechnung erzeugt und die bestellten Waren tatsächlich geliefert werden.

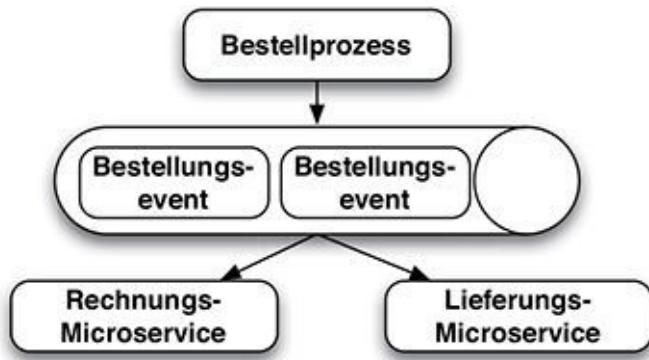
Abb. 8–10 Aufrufe zwischen Microservices



Dazu muss der Bestellprozess den Service für die Rechnung und für die Lieferung kennen. Wenn bei einer fertigen Bestellung weitere Schritte notwendig sind, muss der Bestellprozess auch die Services für diese Schritte aufrufen.

Event-driven Architecture (EDA, ereignisgetriebene Architektur) erlaubt eine andere Modellierung: Wenn die Bestellung erfolgreich beendet worden ist, schickt der Bestellprozess einen Event. Er ist ein Event Emitter. Dieser Event signalisiert allen interessierten Microservices (Event Consumer), dass es eine neue erfolgreiche Bestellung gibt. So kann nun ein Microservice eine Rechnung drucken oder ein anderer Microservice die Lieferung anstoßen.

Abb. 8–11 Event-driven Architecture



Dieses Vorgehen hat einige Vorteile:

- Wenn andere Microservices auch an Bestellungen interessiert sind, können sie sich einfach registrieren. Eine Modifikation am Bestellprozess ist nicht mehr notwendig.
- Ebenso ist es denkbar, dass auch andere Microservices identische Events auslösen – wiederum ohne Änderungen am Bestellungsprozess.
- Die Bearbeitung der Events ist zeitlich entkoppelt. Sie kann später stattfinden.

Auf architektonischer Ebene haben Event-driven Architectures den Vorteil, dass sie eine sehr lose Kopplung erlauben und damit Änderungen sehr einfach möglich sind. Die Microservices müssen sehr wenig übereinander wissen. Allerdings basiert die Kopplung darauf, dass Logik integriert wird. Dadurch kann eine Aufteilung in Microservice mit UI und Microservices mit Logik entstehen. Das ist nicht wünschenswert. Änderungen an Geschäftslogik benötigen oft Änderungen an Logik und UI. Das sind dann getrennte Microservices. Die Änderung kann nicht mehr ohne Weiteres in nur einem Microservice erfolgen und wird dann komplexer.

Technisch können solche Architekturen ohne besonders großen Aufwand mit Messaging (siehe [Abschnitt 9.4](#)) implementiert werden. Microservices in einer solchen Architektur können sehr einfach CQRS ([Abschnitt 10.2](#)) oder Event Sourcing umsetzen ([Abschnitt 10.3](#)).

8.7 Technische Architektur

Eine Aufgabe einer Architektur ist, einen Technologie-Stack zu definieren, mit dem das System gebaut werden kann. Für einzelne Microservices ist das ebenfalls eine sehr wichtige Aufgabe. Aber im Mittelpunkt dieses Kapitels steht das System aus den Microservices als Ganzes. Sicher kann für alle Microservices eine bestimmte Technologie verbindlich definiert werden. Das hat Vorteile: Dann können die Teams Wissen über die Technologie austauschen. Refactorings sind einfacher und Mitarbeiter aus einem Team können unproblematisch auch in anderen Teams aushelfen.

Aber eine Definition von Standardtechnologien ist nicht zwangsläufig: Wenn sie nicht erfolgt, wird es eine Vielzahl an unterschiedlichen Technologien und Frameworks geben. Da typischerweise aber nur jeweils ein Team mit jeder Technologie in Kontakt kommt, kann ein solcher Ansatz akzeptabel sein. Generell ist eine möglichst große Unabhängigkeit das Ziel einer Microservices-Architektur. Für den Technologie-Stack bedeutet die Unabhängigkeit, dass unterschiedliche Technologie-Stacks genutzt werden

können und die Technologie-Entscheidungen unabhängig getroffen werden, aber diese Freiheit kann auch beschnitten werden.

Auf der Ebene des Gesamtsystems gibt es dennoch einige technischen Entscheidungen zu treffen. Wichtig sind für die technische Ausrichtung der Microservices-Architektur aber andere Aspekte als der Technologie-Stack für die Implementierung:

*Technische Entscheidung im
Gesamtsystem*

- Wie im letzten Abschnitt besprochen, kann es Technologien geben, die alle Microservices nutzen können – beispielsweise Datenbanken zur Datenhaltung. Diese Regelungen müssen nicht unbedingt verpflichtend sein. Gerade für Persistenztechnologien wie Datenbanken müssen aber Backups und Disaster-Recovery-Konzepte vorhanden sein, sodass zumindest solche technische Lösungen verpflichtend sein müssen. Ähnliches gilt für andere Basissysteme wie beispielsweise CMS, die ebenfalls von allen Microservices verwendet werden können.
- Die Microservices müssen sich bezüglich Monitoring, Logging und Deployment an bestimmte Standards halten. So kann sichergestellt werden, dass die Vielzahl von Microservices noch einheitlich betrieben werden kann. Ohne solche Standards ist das bei einer größeren Anzahl Microservices kaum noch möglich.
- Weitere Aspekte umfassen Konfiguration ([Abschnitt 8.8](#)), Service Discovery ([Abschnitt 8.9](#)) und Security ([Abschnitt 8.12](#)).
- Resilience ([Abschnitt 10.5](#)) und Load Balancing ([Abschnitt 8.10](#)) sind Konzepte, die in einem Microservice umgesetzt werden müssen. Dennoch kann die übergreifende Architektur erzwingen, dass jeder Microservice in diesem Bereich Vorkehrungen hat.
- Ein weiterer Aspekt ist die Kommunikation der Microservices untereinander (siehe [Kap. 9](#)). Für das Gesamtsystem muss eine Kommunikationsinfrastruktur definiert werden, an die sich auch die Microservices halten.

Die übergreifende Architektur muss nicht unbedingt die Auswahl an Technologien einschränken: Für Logging, Monitoring und Deployment sollte eine Schnittstelle definiert sein. So kann es einen Standard geben, nach dem alle Microservices auf dieselbe Art Nachrichten loggen und an eine gemeinsame Log-Infrastruktur übergeben. Aber die Microservices müssen dazu nicht unbedingt dieselbe Technologie verwenden. Analog kann definiert werden, wie Daten an das Monitoring-System gegeben werden können und welche Daten für das Monitoring relevant sind. Ein Microservice muss die Daten an das Monitoring weiterreichen, aber es muss nicht unbedingt eine Technologie vorgeschrieben werden. Beim Deployment kann eine vollautomatische Continuous-Delivery-Pipeline gefordert werden, die auf eine bestimmte Art Software deployt oder in ein Repository ablegt. Welche konkrete Technologie genutzt wird, ist wieder eine Aufgabe für die Entwickler des jeweiligen Microservice. In der Praxis hat es Vorteile, wenn alle Microservices dieselbe Technologie nutzen. Es reduziert die Komplexität und mit der Technologie wird es auch mehr Erfahrungen geben. Aber für spezielle Anforderungen kann immer noch eine andere technische Lösung verwendet werden, wenn für einen Spezialfall die Vorteile einer solchen Lösung überwiegen. Das ist ein wesentlicher Vorteil der Technologiefreiheit bei Microservices-Architekturen.

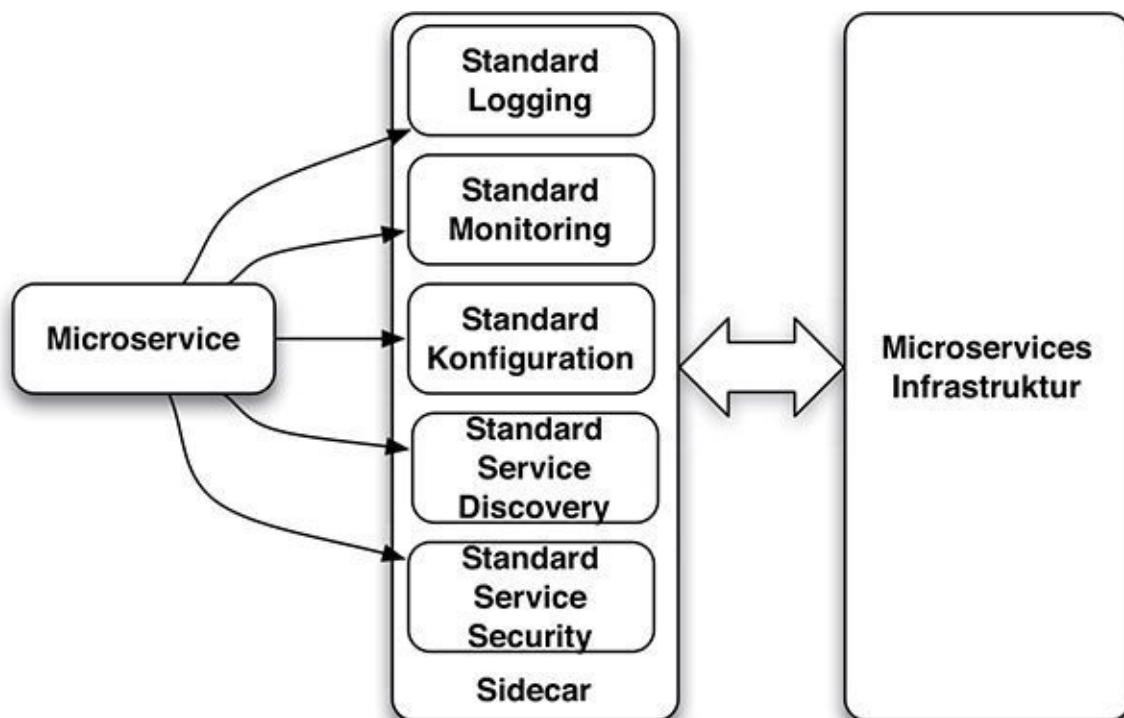
Selbst wenn bestimmte Technologien zur Umsetzung

der Anforderungen an Microservices fest definiert sind, ist es dennoch möglich, andere Technologien zu integrieren.

Sidecar

Dazu kann das Konzept eines Sidecars nützlich sein. Das ist ein Prozess, der sich mit den Standardtechnologien in die Microservices-Architektur integriert und eine Schnittstelle anbietet, mit der ein anderer Prozess diese Features nutzen kann. Dieser Prozess kann in einer ganz anderen Technologie umgesetzt sein, sodass die Technologiefreiheit erhalten bleibt. [Abbildung 8–12](#) zeigt das Konzept: Das Sidecar nutzt die Standardtechnologien und macht sie für einen Microservice in einer beliebigen Technologie verfügbar. Das Sidecar ist ein eigener Prozess und kann daher beispielsweise über REST angesprochen werden, sodass Microservices in beliebigen Technologien das Sidecar nutzen können. [Abschnitt 14.12](#) zeigt ein konkretes Beispiel für ein Sidecar.

Abb. 8–12 Sidecar macht alle Standardtechnologien über eine einfache Schnittstelle verfügbar.



Mit diesem Ansatz können auch Microservices in die Architektur integriert werden, deren technologischer Ansatz eine Nutzung der allgemeinen technischen Basis für Konfiguration, Service Discovery und Security sonst nicht erlaubt, da die Client-Komponente nicht für die genutzte Technologie verfügbar ist.

An einigen Stellen berührt die Definition des Technologie-Stacks auch andere Bereiche. Die Definition von Technologien über alle Microservices hinweg hat auch Auswirkungen auf die Organisation oder kann das Produkt einer bestimmten Organisation sein (siehe [Kap. 13](#)).

Selber ausprobieren und experimentieren

Eine Microservices-Architektur soll definiert werden.

- Welche technischen Aspekte könnte sie umfassen?
- Welche Aspekte würdest du den Teams vorschreiben? Warum?

- Welche sollten die Teams selbst entscheiden? Warum?

Letztendlich geht es darum, welche Freiheiten man den Teams lässt. Dabei gibt es sehr viele Möglichkeiten – von vollständiger Freiheit bis hin zum Vorschreiben praktisch aller Aspekte. Einige Bereiche lassen sich aber nur zentral regeln – die Kommunikationsprotokolle zum Beispiel. [Abschnitt 13.3](#) beleuchtet genauer, wer welche Entscheidungen in einem Microservices-Projekt treffen sollte.

8.8 Konfiguration und Koordination

Die Konfiguration eines Microservice-Systems ist aufwendig. Es gibt eine Vielzahl von Microservices, denen jeweils entsprechende Konfigurationsparameter mitgegeben werden müssen.

Einige Werkzeuge können die Konfigurationswerte speichern und allen Microservices zur Verfügung stellen. Letztendlich sind es Lösungen in Key/Value-Stores, die unter einem bestimmten Schlüssel einen bestimmten Wert abspeichern:

- *Zookeeper* [8] ist ein einfaches hierarchisches System, das im Cluster auf mehrere Server repliziert werden kann. Updates kommen geordnet bei den Clients an. So kann es auch in einem verteilten Umfeld beispielsweise zur Synchronisation genutzt werden. Zookeeper hat ein konsistentes Datenmodell: Alle Knoten haben jederzeit dieselben Daten. Das Projekt ist in Java implementiert und steht unter der Apache-Lizenz.
- *etcd* [9] kommt aus dem Docker/CoreOS-Umfeld. Es bietet eine HTTP-Schnittstelle mit JSON als Datenformat. etcd ist in Go implementiert und steht unter Apache-Lizenz. Ähnlich wie Zookeeper hat auch etcd ein konsistentes Datenmodell und kann für die verteilte Koordination genutzt werden. Beispielsweise kann mit diesem System in einem verteilten System ein Locking umgesetzt werden.
- *Spring Cloud Config* [10] hat ebenfalls eine REST-API. Die Konfigurationsdaten können aus einem Git-Backend kommen. So unterstützt es direkt eine Versionierung der Daten. Die Daten können auch verschlüsselt werden, um Passwörter zu schützen. Das System ist gut in das Java-Framework Spring integriert und kann in Spring-Systemen ohne zusätzlichen Aufwand genutzt werden, da Spring selber schon Konfigurationsmechanismen mitbringt. Spring Cloud Config ist in Java geschrieben und steht unter Apache-Lizenz. Für die Synchronisierung verschiedener verteilter Komponenten hat Spring Cloud Config keine Unterstützung.

Einige der Konfigurationslösungen bieten konsistente Daten. Das bedeutet, dass bei einer Anfrage alle Knoten dieselben Daten zurückgeben. Das ist eigentlich ein Vorteil. Aber laut dem CAP-Theorem kann ein Knoten bei einem Ausfall des Netzwerks nur eine inkonsistente Antwort geben – oder gar keine. Schließlich kann der Knoten ohne Netzwerk nicht wissen, ob andere Knoten bereits andere Werte bekommen haben. Wenn das System nur konsistente Antworten zulässt, kann es in dieser Situation gar keine Antwort geben. Für bestimmte Situationen ist das sehr sinnvoll: Beispielsweise soll nur ein Client eine bestimmte Code-

Konsistenz als Problem

Strecke zu einer Zeit ausführen – um beispielsweise eine Zahlung genau einmal zu veranlassen. Das dazu notwendige Locking kann das Konfigurationssystem übernehmen: In dem Konfigurationssystem gibt es eine Variable, die beim Eintritt in diese Code-Strecke gesetzt werden muss. Nur dann darf der Code ausgeführt werden. Es ist es besser, wenn das Konfigurationssystem keine Antwort zurückgibt und nicht aus Versehen zwei Clients die Code-Strecke parallel ausführen. Aber für Konfiguration sind solche strengen Anforderungen an Konsistenz oft nicht notwendig. Vielleicht ist es besser, wenn ein System einen alten Wert bekommt, als dass es gar keinen Wert bekommt. Allerdings sind bei CAP verschiedene Kompromisse möglich. etcd [10] beispielsweise gibt unter bestimmten Umständen eher eine falsche Antwort als gar keine Antwort.

Ein weiteres Problem mit der zentralisierten Ablage von Konfigurationsdaten ist, dass die Microservices nicht nur von dem Zustand des eigenen Dateisystems und den enthaltenen Dateien abhängen, sondern auch von dem Zustand des Konfigurationsservers. Daher kann ein Microservice nun nicht mehr exakt repliziert werden – dazu ist auch der Zustand des Konfigurationsservers relevant. Das erschwert das Reproduzieren von Fehlern und allgemein die Fehlersuche.

Immutable Server

Ebenso ist der Konfigurationsserver ein Widerspruch zu Immutable Server (unveränderlicher Server). Bei diesem Ansatz führt jede Änderung an der Software zu einer neuen Installation der Software. Letztendlich wird bei einem Update der alte Server beendet und ein neuer Server mit einer komplett neuen Installation der Software wird gestartet. Durch einen externen Konfigurationsserver liegt aber ein Teil der Konfiguration gar nicht auf dem Server und daher wird der Server letztendlich dann doch änderbar, indem die Konfiguration angepasst wird. Genau das sollte aber nicht passieren. Um es zu verhindern, kann statt des Konfigurationsservers eine Konfiguration im Server selber vorgenommen werden. Dann können Änderungen an der Konfiguration nur durch ein Ausrollen eines neuen Servers umgesetzt werden.

Ein ganz anderer Ansatz für die Konfiguration der einzelnen Microservices sind die Installationswerkzeuge aus [Abschnitt 12.4](#). Sie unterstützen nicht nur die Installation von Software, sondern auch die Konfiguration. Für die Konfiguration können beispielsweise Konfigurationsdateien erzeugt werden, die Microservices dann einlesen können. Der Microservice selber merkt nichts von der zentralen Konfiguration, da er nur eine Konfigurationsdatei einliest. Dennoch unterstützen diese Ansätze alle Szenarien, die in einer Microservices-Architektur typischerweise auftreten. Dieser Ansatz ermöglicht also eine zentrale Konfiguration und ist kein Widerspruch zu Immutable Server, da die Konfiguration vollständig in den Server übertragen wird.

Alternative: Installationswerkzeuge

8.9 Service Discovery

Service Discovery stellt sicher, dass Microservices sich gegenseitig finden können. Das ist eigentlich eine sehr einfache Aufgabe: Beispielsweise kann auf allen Rechnern eine Konfigurationsdatei ausgeliefert werden, in der IP-Adresse und Port des Microservice stehen. Das Ausrollen solcher Dateien ist durch typische

Konfigurationsmanagementsysteme möglich. Aber dieser Ansatz ist nicht ausreichend:

- Microservices können kommen und gehen. Das passiert nicht nur wegen des Ausfalls von Servern, sondern auch wegen neuer Deployments oder durch das Skalieren der Umgebung durch das Starten neuer Server. Service Discovery muss dynamisch sein. Eine feste Konfiguration ist nicht ausreichend.
- Durch Service Discovery sind die aufrufenden Microservices nicht mehr so eng an den aufgerufenen Microservice gebunden. Das kommt der Skalierung zugute: Ein Client ist nicht mehr an eine konkrete Instanz des Servers gebunden, sondern kann verschiedene Instanzen kontaktieren – je nach der aktuellen Last auf den einzelnen Servern.
- Wenn alle Microservices einen gemeinsamen Ansatz für Service Discovery haben, entsteht eine zentrale Registratur aller Microservices. Das kann für einen Architektur-Überblick hilfreich sein (siehe [Abschnitt 8.2](#)). Oder es können Monitoring-Informationen von allen Systemen abgeholt werden.

Service Discovery kann in Systemen entbehrlich sein, die Messaging verwenden. Messaging-Systeme entkoppeln Sender und Empfänger bereits. Beide kennen nur den gemeinsamen Kanal, über den sie kommunizieren. Sie kennen aber nicht die Identität des Kommunikationspartners. Die Flexibilität, die Service Discovery anbietet, ist dann durch die Entkoppelung über die Kanäle gegeben.

Prinzipiell ist es denkbar, Service Discovery mit Konfigurationslösungen umzusetzen (siehe [Abschnitt 8.8](#)). Schließlich soll nur die Information transportiert werden, welcher Service wo erreichbar ist. Allerdings sind Konfigurationsmechanismen dafür eigentlich die falschen Werkzeuge. Bei einer Service Discovery ist eine hohe Verfügbarkeit noch wichtiger als bei einem Konfigurationsserver. Ein Ausfall der Service Discovery kann im Extremfall dazu führen, dass Kommunikation zwischen Microservices unmöglich wird. Also ist der Trade-off zwischen Konsistenz und Verfügbarkeit anders als bei Konfigurationssystemen. Daher sollten Konfigurationssysteme für Service Discovery nur genutzt werden, wenn sie eine entsprechende Verfügbarkeit anbieten. Das kann Konsequenzen für die notwendige Architektur des Service-Discovery-Systems haben.

Es gibt viele unterschiedliche Technologien für Service Technologien Discovery:

- Ein Beispiel ist DNS (Domain Name System) [12]. Dieses Protokoll stellt sicher, dass ein Hostname wie www.ewolff.com zu einer IP-Adresse aufgelöst werden kann. DNS ist ein wesentlicher Bestandteil des Internets und hat seine Skalierbarkeit und Verfügbarkeit klar unter Beweis gestellt. DNS ist hierarchisch organisiert: Es gibt einen DNS-Server, der die .com-Domäne verwaltet. Dieser DNS-Server weiß, welcher DNS-Server die Subdomäne ewolff.com verwaltet, und der DNS-Server dieser Subdomäne kennt schließlich die IP-Adresse für www.ewolff.com. So kann ein Namensraum hierarchisch organisiert werden und verschiedene Organisationen können unterschiedliche Teile des Namensraums verwalten. Soll ein Server namens server.ewolff.com angelegt werden, kann das einfach durch eine Änderung im DNS-Server der Domäne ewolff.com erfolgen. Diese Unabhängigkeit passt gut zum Konzept

von Microservices, die Unabhängigkeit in den Mittelpunkt der Architektur stellen. Für die Ausfallsicherheit gibt es jeweils mehrere Server, die eine Domäne verwalten. Um diese Skalierbarkeit zu erreichen, unterstützt DNS Caching, sodass Anfragen nicht die komplette Auflösung des Namens über mehrere DNS-Server umsetzen müssen, sondern durch einen Cache bedient werden können. Das kommt nicht nur der Performance, sondern auch der Ausfallsicherheit zugute.

Für Service Discovery ist es nicht ausreichend, den Namen eines Servers zu einer IP-Adresse aufzulösen. Es muss für jeden Service außerdem einen Port geben. Dazu hat DNS SRV-Records. Die enthalten für einen Dienst die Information, auf welchem Rechner und Port der Service erreichbar ist. Zusätzlich können eine Priorität und ein Gewicht für einen bestimmten Server angegeben werden. Diese Werte können dazu genutzt werden, einen der Server auszuwählen und so leistungsfähigere Server zu bevorzugen. Durch diesen Ansatz bietet DNS eine Ausfallsicherheit und eine Lastverteilung auf mehrere Server an. Vorteile von DNS sind neben der Skalierbarkeit auch die Verfügbarkeit vieler verschiedener Implementierungen und die breite Unterstützung in verschiedenen Programmiersprachen.

- Eine oft genutzte Implementierung für einen DNS-Server ist BIND [13]. Es läuft auf verschiedenen Betriebssystemen (Linux, BSD, Windows, Mac OS X), ist in der Programmiersprache C geschrieben und steht unter einer Open-Source-Lizenz.
- *Eureka* [14] kommt aus dem Netflix-Stack. Es ist in Java geschrieben und steht unter der Apache-Lizenz. Die Beispielanwendung dieses Buchs nutzt Eureka für Service Discovery (siehe [Abschnitt 14.8](#)). Für jeden Service speichert Eureka unter dem Service-Namen einen Host und einen Port, unter dem der Service zur Verfügung steht. Eureka kann die Informationen über die Services auf mehrere Eureka-Server replizieren, um so die Verfügbarkeit zu erhöhen. Eureka ist ein REST-Service. Zu Eureka gehört eine Java-Bibliothek für die Clients. Durch das Sidecar-Konzept ([Abschnitt 8.7](#)) kann diese Bibliothek auch von Systemen genutzt werden, die nicht in Java geschrieben sind. Die Kommunikation mit dem Eureka-Server übernimmt der Sidecar, der dem Microservice dann Service Discovery anbietet. Auf den Clients können die Informationen vom Server in einem Cache gehalten werden, sodass Abfragen ohne Kommunikation mit dem Server möglich sind. Der Server kontaktiert regelmäßig die registrierten Services, um festzustellen, welche Services ausgefallen sind. Eureka kann als Basis für Load Balancing genutzt werden, da mehrere Instanzen für einen Service registriert werden können, auf die dann die Last verteilt werden kann. Eureka wurde ursprünglich in der Amazon-Cloud genutzt.
- *Consul* [15] ist ein Key/Value-Store und passt daher auch in den Bereich der Konfigurationsserver ([Abschnitt 8.8](#)). Es kann neben konsistenten Abfragen auch auf Verfügbarkeit optimieren [11]. Clients können sich beim Server registrieren und auf bestimmte Events reagieren. Neben einem DNS-Interface hat es auch eine HTTP/JSON-Schnittstelle. Es kann überprüfen, ob Services noch erreichbar sind, indem es Health Checks ausführt. Consul ist in Go geschrieben und steht unter der Mozilla-Open-Source-Lizenz. Consul kann außerdem Konfigurationsdateien aus Templates erstellen. So kann ein System, das Dienste in einer Konfigurationsdatei erwartet, ebenfalls mit Consul konfiguriert werden.

Jede Microservices-Architektur sollte ein Service-Discovery-System nutzen. Es bildet die Basis für die Verwaltung einer großen Anzahl von Services und für weitere Features wie Load Balancing. Bei einer kleinen Anzahl Microservices wäre es noch denkbar, ohne Service Discovery auszukommen. Aber bei einem großen System ist Service Discovery unverzichtbar. Da die Anzahl der Microservices mit der Zeit wächst, sollte man gleich von Anfang an Service Discovery in die Architektur integrieren. Außerdem nutzt praktisch jedes System mindestens die Namensauflösung von Hosts, die bereits eine einfache Service Discovery ist.

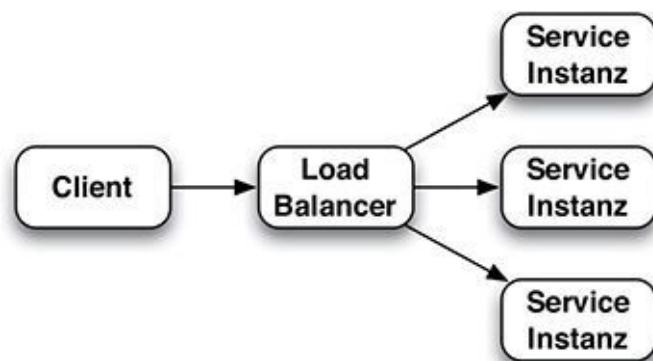
8.10 Load Balancing

Ein Vorteil von Microservices ist, dass jeder einzelne Service unabhängig skaliert werden kann. Um die Last zwischen den Instanzen aufzuteilen, können in einer Messaging-Lösung (siehe [Abschnitt 9.4](#)) einfach mehrere Instanzen registriert werden, die sich die Last teilen. Die konkrete Verteilung der einzelnen Nachrichten nimmt das Messaging-System vor. Nachrichten können entweder an einen der Empfänger (Point-to-Point) oder alle Empfänger (Publish/Subscribe) verteilt werden.

Bei REST und HTTP muss ein Load Balancer REST/HTTP verwendet werden. Die Aufgabe des Load Balancers ist, sich nach außen wie eine einzige Instanz zu verhalten, aber die Anfragen auf mehrere Instanzen zu verteilen. Außerdem kann der Load Balancer beim Deployment nützlich sein: Instanzen der neuen Version des Microservice können zunächst starten, ohne Last zu bekommen. Danach kann der Load Balancer dann so umkonfiguriert werden, dass die neuen Microservices in Betrieb genommen werden. Dabei kann die Last auch schrittweise erhöht werden. Das verringert das Risiko eines Ausfalls des Systems.

[Abbildung 8–13](#) zeigt das Prinzip eines Proxy-basierten Load Balancers: Der Client schickt seine Anfragen an einen Load Balancer, der auf einem anderen Server läuft. Dieser Load Balancer ist dafür zuständig, jeden Request an eine der bekannten Instanzen zu schicken. Dort wird er verarbeitet.

Abb. 8–13 Proxy-basierter Load Balancer



Dieser Ansatz ist für Web-Site üblich. Er ist relativ einfach umsetzbar. Der Load Balancer fragt Informationen von den Service-Instanzen ab, um die Last der Instanzen zu ermitteln. Ebenso kann der Load Balancer einen Server aus der Lastverteilung entfernen, wenn der Knoten auf Anfragen nicht mehr reagiert.

Auf der anderen Seite hat dieser Ansatz den Nachteil, dass der gesamte Verkehr für eine Art von Service über einen Load Balancer geleitet werden muss. Der Load Balancer kann dadurch ein Flaschenhals werden. Außerdem führt der Ausfall des Load Balancers zu einem Ausfall eines Microservice.

Ein zentraler Load Balancer für alle Microservices ist Zentraler Load Balancer nicht nur aus diesen Gründen nicht zu empfehlen, sondern auch wegen der Konfiguration. Die Konfiguration des Load Balancers wird sehr aufwendig, wenn nur ein Load Balancer für viele Microservices zuständig ist. Außerdem muss die Konfiguration zwischen allen Microservices koordiniert werden. Gerade beim Deployment einer neuen Version eines Microservice kann eine Änderung am Load Balancer sinnvoll sein, um so die neuen Microservices erst nach einem ausführlichen Test unter Last zu setzen. Die Koordination zwischen Microservices gerade beim Deployment sollte möglichst vermieden werden, um ein unabhängiges Deployment der Microservices zu ermöglichen. Bei einer solchen Umkonfiguration muss auch darauf geachtet werden, dass der Load Balancer eine dynamische Neukonfiguration unterstützt und beispielsweise keine Informationen über Sessions verliert, wenn der Microservice Sessions verwendet. Auch aus diesem Grund ist es empfehlenswert, Microservices zustandslos zu implementieren.

Es sollte pro Microservice einen Load Balancer geben, Load Balancer pro Microservice der die Last zwischen den verschiedenen Instanzen des Microservice verteilt. Dadurch kann jeder Microservice getrennt Last verteilen und unterschiedliche Konfigurationen pro Microservice sind einfach möglich. Ebenso ist es einfach, den Load Balancer bei der Aktivierung einer neuen Version entsprechend umzukonfigurieren. Bei einem Ausfall des Load Balancers ist dann allerdings der Microservice nicht mehr verfügbar.

Für Load Balancing in diesem Bereich gibt es Technologien unterschiedliche Ansätze:

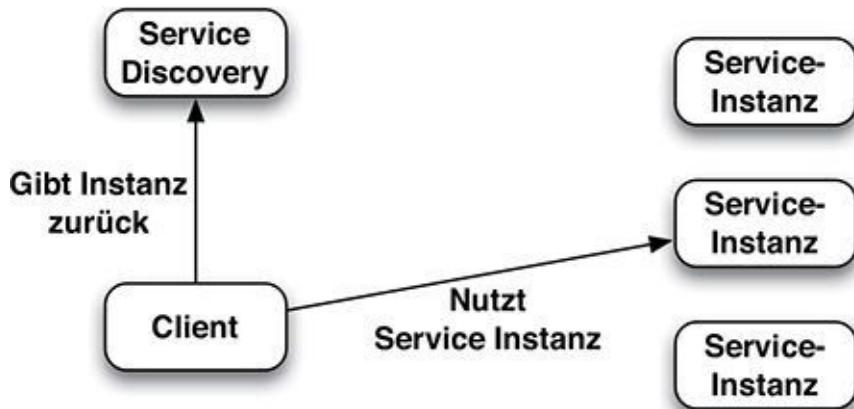
- Der Apache httpd Webserver unterstützt mit der Erweiterung mod_proxy_balancer [7] Load Balancing.
- Auch der Webserver nginx [8] kann so konfiguriert werden, dass er Load Balancing unterstützt. Einen Webserver als Load Balancer zu nutzen, hat den Vorteil, dass er auch statische Webseiten, CSS und Grafiken ausliefern kann. Außerdem wird die Anzahl der Technologien so reduziert.
- HAProxy [9] ist eine Lösung für Load Balancing und Hochverfügbarkeit. Sie unterstützt nicht HTTP, sondern alle TCP-basierten Protokolle.
- Cloud-Anbieter haben meistens auch Load Balancer im Angebot. Amazon bietet beispielsweise Elastic Load Balancing [10] an. Das kann mit Auto Scaling kombiniert werden, sodass bei höherer Last automatisch neue Instanzen gestartet werden und so die Anwendung automatisch mit der Last skaliert.

Eine weitere Möglichkeit für das Load Balancing ist Service Discovery (Abb. 8–14) (siehe Abschnitt 8.9).

Wenn die Service Discovery unterschiedliche Knoten für einen Service zurückgibt, kann so die Last auf mehrere Knoten verteilt werden. Dieses Vorgehen ermöglicht das

Umlenken auf einen anderen Knoten aber nur dann, wenn eine neue Service Discovery durchgeführt wird. Damit ist eine feingranulare Verteilung der Last schwierig. Ein neuer Knoten wird daher erst nach einiger Zeit genügend Last abbekommen. Schließlich kann der Ausfall eines Knotens nur schwer korrigiert werden, weil dazu ein neues Service Discovery notwendig wäre. Nützlich ist hierfür, dass bei DNS für eine Anfrage angegeben werden kann, wie lange die Daten gültig sind. Danach muss die Service Discovery erneut durchlaufen. Damit ist ein Load Balancing mit DNS-Lösungen und auch mit Consul einfach möglich. Leider werden diese Gültigkeitsdauern oft nicht vollständig korrekt umgesetzt.

Abb. 8–14 Load Balancing mit Service Discovery

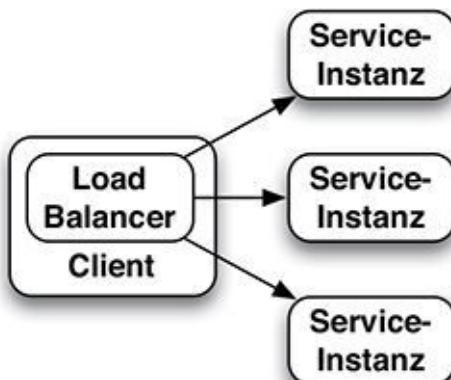


Load Balancing mit Service Discovery ist einfach, weil Service Discovery in einem Microservice-System sowieso vorhanden sein muss. Das Load Balancing führt keine zusätzlichen Software-Komponenten ein. Außerdem führt der Verzicht auf einen zentralen Load Balancer dazu, dass es keinen Flaschenhals gibt und keine zentrale Komponente, deren Ausfall erhebliche Konsequenzen hätte.

Der Client kann auch selber einen Load Balancer nutzen. Der Load Balancer kann im Code des Microservice implementiert sein oder ein Proxy-basierter Load Balancer wie nginx oder Apache httpd sein, der auf demselben Rechner läuft wie der Microservice. Dadurch gibt es keinen Flaschenhals, weil jeder Client seinen eigenen Load Balancer hat, und der Ausfall eines einzelnen Load Balancers hat kaum Konsequenzen. Allerdings müssen Änderungen an der Konfiguration an alle Load Balancer weitergegeben werden, was aufwändig ist.

Client-seitiges Load Balancing

Abb. 8–15 Client-seitiges Load Balancing



Eine Implementierung von client-seitigem Load Balancing ist Ribbon [11]. Es ist eine in Java implementierte Bibliothek, die Eureka nutzen kann, um die Service-Instanzen zu finden. Alternativ kann eine Liste von Servern an Ribbon übergeben werden. Ribbon setzt verschiedene Algorithmen für das Load Balancing um. Gerade bei der Nutzung mit Eureka muss der einzelne Load Balancer nicht mehr konfiguriert werden. Wegen des Sidecar-Konzepts kann Ribbon auch von Microservices genutzt werden, die nicht in Java implementiert sind. Das Beispielsystem nutzt Ribbon (siehe [Abschnitt 14.11](#)).

Consul bietet die Möglichkeit, für Konfigurationsdateien von Load Balancern Templates zu definieren. Dadurch kann die Konfiguration des Load Balancers mit Daten aus der Service Discovery gefüttert werden. Ein client-seitiges Load Balancing kann implementiert werden, indem für jeden Client ein Template definiert wird, in das durch Consul alle Service-Instanzen eingetragen werden. Dieser Prozess kann regelmäßig wiederholt werden. Auf diese Weise ist wieder eine zentrale Konfiguration des Systems möglich und client-seitiges Load Balancing recht einfach umsetzbar.

Es ist kaum sinnvoll, in einem Microservice-System Load Balancing und Architektur mehr als eine Art des Load Balancings zu nutzen. Daher sollte diese Entscheidung einmal für das System getroffen werden. Load Balancing und Service Discovery haben einige Berührungspunkte. Service Discovery kennt alle Service-Instanzen, Load Balancing verteilt die Last zwischen den Instanzen. Beide Technologien müssen zusammenspielen. Daher beeinflussen sich die Technologie-Entscheidungen in diesem Bereich gegenseitig.

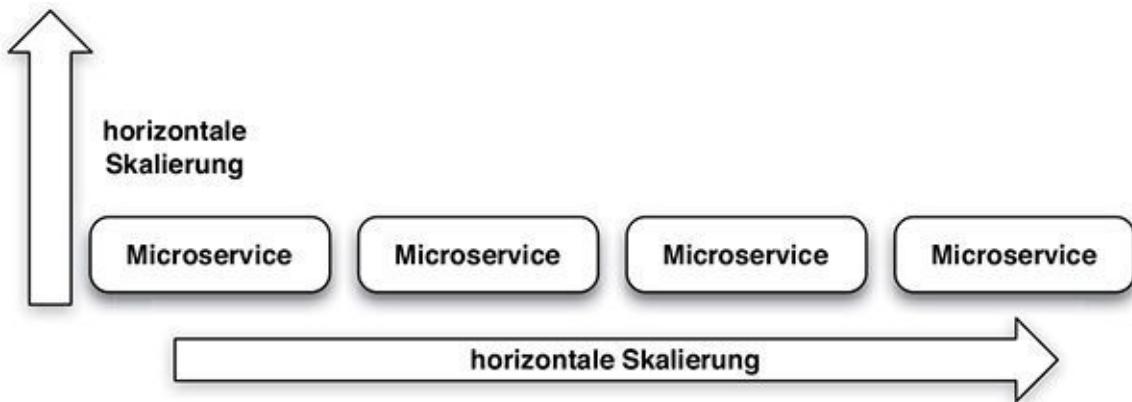
8.11 Skalierbarkeit

Um mit einer hohen Last zurechtzukommen, müssen Microservices skalieren. Skalierbarkeit bedeutet, dass ein System mehr Last bearbeiten kann, wenn es mehr Ressourcen bekommt.

Es gibt zwei verschiedene Arten der Skalierbarkeit:

- *Horizontale Skalierbarkeit* bedeutet, dass mehr Ressourcen zur Verfügung stehen, die jeweils einen Teil der Last bearbeiten. Die Anzahl der Ressourcen steigt also.
- *Vertikale Skalierbarkeit* bedeutet, dass leistungsfähigere Ressourcen genutzt werden, um mehr Last handzuhaben. Eine einzelne Ressource wird also mehr Last abarbeiten. Die Anzahl der Ressourcen bleibt konstant.

Abb. 8–16 Horizontale und vertikale Skalierung



Horizontale Skalierbarkeit ist meistens die bessere Wahl, weil die Grenze für die mögliche Anzahl Ressourcen sehr hoch ist. Außerdem ist es billiger, mehr Ressourcen zu kaufen als leistungsfähigere. Ein sehr schneller Computer ist oft teurer als viele langsamere.

Microservices nutzen meistens eine horizontale Skalierung. Dazu wird die Last mithilfe von Load Balancing über mehrere Microservice-Instanzen verteilt.

Skalierung, Microservices und Load Balancing

Die Microservices selber müssen dazu zustandslos sein. Genauer gesagt sollten sie keinen Zustand haben, der spezifisch für einen Nutzer ist. Dann kann die Last nämlich nur auf die Knoten verteilt werden, die den entsprechenden Zustand haben. Der Zustand für einen Nutzer kann in einer Datenbank gespeichert oder sonst in einem externen Speicher (z. B. In-Memory-Store) untergebracht werden, auf den alle Microservices Zugriff haben.

Skalierbarkeit bedeutet nur, dass die Last auf mehrere Knoten verteilt werden kann. Wie das System tatsächlich auf Last reagiert, ist offen. Wichtiger ist eigentlich, dass sich das System tatsächlich einer steigenden Last anpasst. Dazu ist es notwendig, dass ein Microservice abhängig von der Last neue Instanzen startet, auf die Last verteilt werden kann. So kann der Microservice auch mit hoher Last umgehen. Bei einer großen Anzahl Microservices muss dieser Prozess automatisiert sein, weil manuelle Prozesse zu aufwendig wären.

Dynamische Skalierung

Das Starten eines neuen Microservice ist an verschiedenen Stellen der Continuous-Deployment-Pipeline ([Kap. 12](#)) notwendig, um die Services zu testen. Dazu kann ein passendes Deployment-System wie Chef oder Puppet genutzt werden oder es wird einfach eine neue virtuelle Maschine oder ein neuer Docker-Container mit dem Microservice gestartet. Für eine dynamische Skalierung kann dieser Mechanismus ebenfalls genutzt werden. Er muss lediglich zusätzlich die neuen Instanzen noch in das Load Balancing eintragen. Allerdings sollte die Instanz gleich von Anfang mit der Produktionslast umgehen können: Also sollten beispielsweise die Caches schon mit Daten gefüllt sein.

Die dynamische Skalierung ist mit Service Discovery besonders einfach: Der Microservice muss sich bei der Service Discovery anmelden. Die Service Discovery kann den Load Balancer so konfigurieren, dass er Last auf die neue Instanz verteilt.

Die dynamische Skalierung muss auf Basis einer Metrik erfolgen. Wenn die Antwortzeit eines Microservice zu hoch oder die Anzahl Anfragen sehr hoch ist, müssen neue Instanzen gestartet werden. Das dynamische Skalieren kann ein Teil des Monitorings sein (siehe [Abschnitt 12.3](#)). Schließlich soll das Monitoring die Reaktion auf außergewöhnliche Metrik-Werte ermöglichen. Die meisten Monitoring-Infrastrukturen

bieten die Möglichkeit, auf Werte bei Metriken durch den Aufruf eines Skripts zu reagieren. Das Skript kann zusätzliche Instanzen des Microservice starten. Das ist mit den meisten Cloud- und Virtualisierungsumgebungen einfach möglich. Umgebungen wie die Amazon Cloud bieten entsprechende Lösungen für die automatische Skalierung an, die ähnlich funktionieren. Eine eigene Lösung ist aber nicht allzu kompliziert, da die Skripte sowieso nur alle paar Minuten laufen und daher Ausfälle für einige Zeit akzeptabel sind. Da die Skripte Teil des Monitorings sind, werden sie eine ähnliche Verfügbarkeit wie das Monitoring haben und damit ausreichend verfügbar sein.

Gerade bei Cloud-Infrastrukturen ist es wichtig, die Instanzen bei einer niedrigen Last auch wieder herunterzufahren. Denn jede laufende Instanz kostet auf einer Cloud Geld. Auch dafür können Skripte als Reaktion auf bestimmte Metrik-Werte dienen.

Microservices haben in Bezug auf die Skalierung vor allem den Vorteil, dass sie einzeln skaliert werden können. Bei einem Deployment-Monolithen kann nur der gesamte Monolith in mehr Instanzen gestartet werden. Die feingranulare Skalierung scheint zunächst kein besonders großer Vorteil zu sein, aber einen kompletten E-Commerce-Shop in vielen Instanzen laufen zu lassen, nur um die Suche schneller zu machen, führt zu hohen Aufwänden: Viel Hardware ist notwendig, eine komplexe Infrastruktur muss aufgebaut werden und es werden Systemteile vorgehalten, die gar nicht genutzt werden. Diese Systemteile verkomplizieren das Deployment und Monitoring. Die Möglichkeit zur dynamischen Skalierung hängt entscheidend von der Größe der Services ab und davon, wie schnell neue Instanzen gestartet werden können. In diesem Bereich haben Microservices klare Vorteile.

Microservices haben meistens schon ein automatisiertes Deployment, das auch sehr einfach einsetzbar ist. Ebenso ist schon ein Monitoring vorhanden. Ohne automatisiertes Deployment und Monitoring ist ein Microservice-System praktisch nicht betreibbar. Kommt noch Load Balancing hinzu, fehlt für eine automatische Skalierung nur noch ein Skript. Daher sind Microservices für die dynamische Skalierung eine sehr gute Basis.

Sharding bedeutet, dass man die verwaltete Datenmenge unterteilt und jeder Instanz die Zuständigkeit für einen Teil der Daten überträgt. Beispielsweise kann eine Instanz für die Kunden von A-E zuständig sein oder für die Kunden, bei denen die Kundenummer auf 9 endet. Sharding ist eine Spielart der horizontalen Skalierung: Es werden mehr Server genutzt. Nur sind nicht alle Server gleichberechtigt, sondern jeder ist für einen anderen Ausschnitt aus der Datenmenge zuständig. Für Microservices ist diese Skalierung recht einfach möglich, da die Fachlichkeit sowieso in mehrere Microservices aufgeteilt ist. Jeder kann dann seine Daten sharden und über Sharding horizontal skalieren. Ein Deployment-Monolith ist so kaum skalierbar, weil er alle Daten verwaltet. Wenn der Deployment-Monolith Kunden und Waren verwaltet, kann er schlecht für beide Datenarten geshardet werden. Um Sharding tatsächlich umzusetzen, muss der Load Balancer natürlich auch die Last passend auf die Shards verteilen.

Skalierbarkeit bedeutet, dass mehr Last durch mehr Ressourcen verarbeitet werden kann. Der Durchsatz steigt – also die Anzahl der verarbeiteten Anfragen. Die

Antwortzeit bleibt jedoch im besten Fall konstant – gegebenenfalls steigt sie, aber nicht so sehr, dass das System Fehler erzeugt oder für den Nutzer zu langsam ist.

Wenn schnellere Antwortzeiten gefordert sind, hilft horizontales Skalieren nicht. Einige Ansätze zur Optimierung der Antwortzeit von Microservices gibt es dennoch:

- Die Microservices können auf schnelleren Rechnern deployt werden. Das ist vertikale Skalierung. Die Microservices können dann die einzelnen Anfragen schneller abarbeiten. Wegen der Deployment-Automatisierung ist die vertikale Skalierung recht einfach umsetzbar. Der Service muss nur auf einer schnelleren Hardware deployt werden.
- Aufrufe über das Netzwerk haben eine lange Latenz. Daher kann eine mögliche Optimierung der Verzicht auf solche Aufrufe sein. Stattdessen können Caches genutzt oder die Daten repliziert werden. Caches können oft sehr einfach in die vorhandene Kommunikation eingebaut werden. Für REST reicht beispielsweise ein einfacher HTTP-Cache.
- Bei einem guten fachlichen Schnitt der Microservices sollte eine Anfrage nur in einem Microservice bearbeitet werden, sodass keine Kommunikation über das Netzwerk hinweg notwendig ist. Bei einem guten fachlichen Schnitt ist nämlich die Logik für das Bearbeiten einer Anfrage in einem Microservice implementiert, sodass eine Änderung der Logik auch nur Änderungen an einem Microservice erzwingt. Dann erreichen Microservices zumindest keine schlechtere Antwortzeit, als dies bei Deployment-Monolithen der Fall wäre.

Bei der Optimierung der Antwortzeiten bergen Microservices also die Gefahr, dass durch Kommunikation der Microservices über das Netz die Antwortzeiten eher schlecht sind. Dem kann man aber entgegensteuern.

8.12 Sicherheit

In einer Microservices-Architektur muss jeder Microservice wissen, wer den aktuellen Aufruf ausgelöst hat und das System nutzen möchte. Dazu muss eine einheitliche Sicherheitsarchitektur existieren: Schließlich können Microservices für eine Anfrage zusammenarbeiten und bei jeder Anfrage kann ein anderer Microservice zuständig sein. Daher muss die Sicherheitsstruktur auf der Ebene des Gesamtsystems definiert werden. Nur so kann sichergestellt werden, dass ein Zugriff eines Nutzers im gesamten System einheitlich bezüglich Sicherheit behandelt wird.

Sicherheit hat zwei wesentliche Aspekte: Authentifizierung und Autorisierung. Authentifizierung ist der Prozess, um die Identität eines Benutzers zu überprüfen. Autorisierung ist die Entscheidung, ob ein bestimmter Benutzer eine bestimmte Aktion ausführen darf. Die beiden Prozesse sind unabhängig voneinander: Die Überprüfung der Identität bei der Authentifizierung hat nicht direkt etwas mit der Autorisierung zu tun.

In einer Microservices-Architektur sollten die einzelnen Microservices keine Authentifizierung übernehmen. Es ist kaum sinnvoll, wenn jeder Microservice Benutzername und Passwort

Sicherheit bei Microservices

selber überprüft. Für die Authentifizierung muss ein zentraler Server genutzt werden. Bei der Autorisierung ist ein Zusammenspiel gefragt: Oft gibt es Benutzergruppen, die zentral verwaltet werden müssen. Ob eine bestimmte Benutzergruppe ein bestimmtes Feature in einem Microservice nutzen darf, sollte der Microservice entscheiden. So können Änderungen an der Autorisierung eines Microservice auf die Implementierung dieses Microservice begrenzt werden.

Eine mögliche Lösung für diese Herausforderung ist **OAuth2**. Dieses Protokoll wird auch im Internet breit genutzt. Google, Microsoft, Twitter, XING oder Yahoo bieten alle Unterstützung für dieses Protokoll.

Abb. 8–17 Das OAuth2-Protokoll

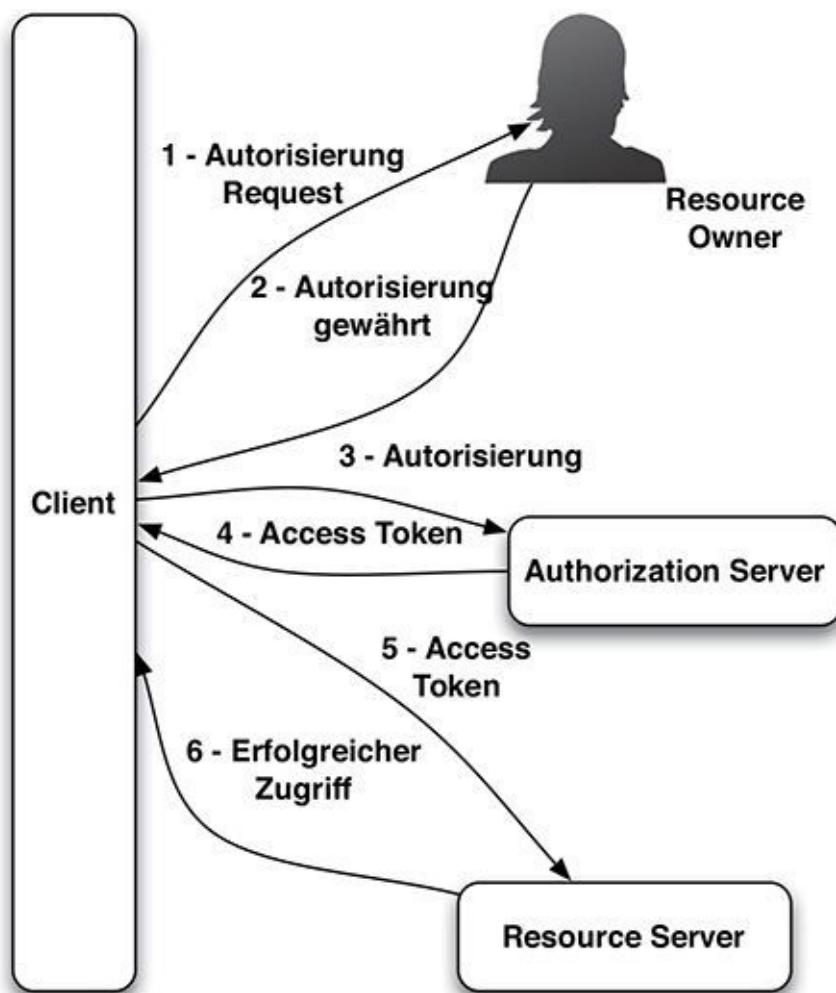


Abbildung 8–17 zeigt den Ablauf des OAuth2-Protokolls, wie es der Standard [16] definiert:

1. Der Client fragt beim Resource Owner nach, ob er eine bestimmte Aktion durchführen darf. Beispielsweise kann die Anwendung Zugriff auf das Profil oder bestimmte Daten in einem sozialen Netz anfordern, die der Resource Owner dort gespeichert hat. Der Resource Owner ist häufig der Nutzer des Systems.
2. Wenn der Resource Owner dem Client den Zugriff gewährt, bekommt der Client eine entsprechende Antwort vom Resource Owner.
3. Der Client nutzt die Antwort vom Resource Owner, um eine Anfrage beim

Authorization Server zu stellen. Im Beispiel wäre der Authorization Server in dem sozialen Netzwerk beheimatet.

4. Der Authorization Server gibt ein Access Token zurück.
5. Mit diesem Access Token kann der Client nun einen Resource Server aufrufen und dort die notwendigen Informationen bekommen. Bei dem Aufruf kann das Token beispielsweise in einem HTTP-Header untergebracht werden.
6. Der Resource Server beantwortet die Anfragen.

Die Interaktion mit dem Authorization Server kann unterschiedlich funktionieren:

- Bei dem *Passwort-Ansatz* zeigt der Client dem Nutzer in Schritt 1 ein Formular, in das der Resource Owner Benutzername und Passwort eingeben kann. Diese Informationen nutzt der Client in Schritt 3, um beim Authorization Server das Access Token mit einem HTTP POST zu bekommen. Dieser Ansatz hat den Nachteil, dass der Client Benutzername und Passwort bearbeitet. Der Client kann unsicher implementiert sein und dann sind diese Daten gefährdet.

Mögliche Abläufe des Protokolls
- Bei dem *Authorization-Ansatz* lenkt der Client den Anwender in Schritt 1 auf eine Website um, die der Authorization Server anzeigt. Dort kann der Nutzer wählen, ob der Zugriff gestattet werden soll. Ist das der Fall, bekommt der Client als Schritt 2 einen Authorization-Code über eine HTTP-URL geliefert. So kann der Authorization Server sicher sein, dass der richtige Client den Code bekommt, da der Server die URL wählt. Der Client kann dann in Schritt 3 mit einem HTTP POST das Access Token mit diesem Authorization-Code erzeugen. Dieser Ansatz wird im Wesentlichen vom Authorization Server umgesetzt und ist daher für einen Client sehr einfach zu nutzen. Der Client wäre in diesem Szenario eine Webanwendung auf dem Server: Sie bekommt vom Authorization Server den Code geschickt und nur sie kann durch den HTTP POST daraus ein Access Token machen.
- Bei *Implicit* läuft der Prozess ähnlich wie beim Authorization-Ansatz. Nach dem Redirect auf den Authorization Server in Schritt 1 bekommt der Client direkt ein Access Token über ein Redirect zurück. Dadurch kann der Browser oder eine mobile Anwendung direkt das Access Token auslesen. Schritt 3 und 4 entfallen. Das Access Token ist aber nun schlechter gegen Angriffe geschützt, da der Authorization Server es nicht an den Client schickt. Dieser Ansatz ist sinnvoll, wenn JavaScript-Code auf dem Client oder eine mobile Anwendung das Access Token nutzen soll.
- Bei *Client Credentials* nutzt der Client in Schritt 1 eine bestimmte Information, die der Client kennt, um das Access Token vom Authorization Server zu bekommen. So kann der Client ohne weitere Informationen vom Resource Owner auf Daten zugreifen. Beispielsweise könnte so eine Statistik-Software Daten der Kunden auslesen und auswerten.

Mit dem Access Token kann der Client auf Ressourcen zugreifen. Das Access Token muss geschützt sein: Wenn Unberechtigte Zugriff auf das Access Token bekommen, können sie damit alle Aktionen auslösen, die der Resource Owner auch auslösen kann. In dem Token selber können einige Informationen kodiert sein. So können neben dem Klarnamen auch

Informationen im Token enthalten sein, die dem Nutzer bestimmte Rechte zuweisen oder die Zugehörigkeit zu bestimmten Benutzergruppen.

JSON Web Token (JWT) ist ein Standard für die Informationen, die in einem Access Token enthalten sind. JSON Web Token (JWT)

JSON dient als Datenstruktur. Damit das Access Token überprüft werden kann, ist eine digitale Unterschrift mit JWS (JSON Web Signature) möglich. Ebenso kann das Access Token mit JSON Web Encryption (JWE) verschlüsselt werden. Im Access Token können Informationen über den Aussteller des Access Token, den Resource Owner, den Gültigkeitszeitraum oder die Adressaten des Access Token enthalten sein. Eigene Daten können in dem Access Token auch enthalten sein. Das Access Token ist darauf optimiert, durch eine Kodierung des JSON mit BASE64 auch als HTTP-Header genutzt zu werden. Diese sind üblicherweise größenbeschränkt.

In einer Microservices-Architektur kann der Nutzer sich zunächst durch einen der OAuth2-Ansätze authentifizieren. Dann kann er die Webseite eines Microservice oder einen Microservice per REST aufrufen. Das Access Token kann jeder Microservice bei jedem Aufruf an andere Microservices weiterreichen. Die Microservices können anhand des Access Tokens entscheiden, ob ein bestimmter Zugriff erlaubt sein soll oder nicht. Dazu kann zunächst die Gültigkeit des Tokens überprüft werden. Bei JWT muss dazu nur das Token entschlüsselt und die Unterschrift des Authorization Servers überprüft werden. Dann kann anhand der Informationen des Tokens entschieden werden, ob der Benutzer den Microservice so nutzen darf. Dazu können Informationen aus dem Token dienen. So ist es beispielsweise möglich, die Zugehörigkeit zu bestimmten Benutzergruppen direkt im Token zu speichern. OAuth2, JWT und Microservices

Wichtig ist, dass nicht im Access Token definiert ist, welcher Zugriff auf welchen Microservice möglich ist. Das Access Token wird vom Authorization Server ausgestellt. Wäre die Information über die Zugriffe im Authorization Server verfügbar, müsste jede Änderung der Zugriffsrechte im Authorization Server erfolgen – und nicht in den Microservices. Das schränkt die Änderbarkeit der Microservices ein, weil für Änderungen an den Zugriffsrechten der Authorization Server als zentrale Komponente geändert werden muss. Der Authorization Server sollte nur die Zuordnung zu Benutzergruppen verwalten und die Microservices dann auf Basis solcher Informationen aus dem Token den Zugriff erlauben oder verbieten.

Prinzipiell wären neben OAuth2 auch andere technische Ansätze denkbar, solange sie einen zentralen Server für die Autorisierung nutzen und ein Token verwenden, mit dem der Zugriff auf einzelne Microservices geregelt werden kann. Ein Beispiel ist Kerberos [18], das auf eine verhältnismäßig lange Historie zurückblicken kann. Es ist aber nicht so gut auf REST abgestimmt wie OAuth2. Eine weitere Alternative ist SAML und SAML 2.0 [23]. Es definiert ein Protokoll, das XML und HTTP nutzt, um Autorisierung und Authentifizierung durchzuführen. Technologien

Schließlich können signierte Cookies von einem selbst geschriebenen Sicherheitsservice angelegt werden. Durch eine kryptografische Signatur kann festgestellt werden, ob der Cookie tatsächlich vom System ausgestellt worden ist. Der Cookie kann

dann die Rechte oder Gruppen des Benutzers enthalten. Microservices können den Cookie überprüfen und den Zugriff gegebenenfalls einschränken. Es besteht das Risiko, dass der Cookie geklaut wird. Dazu muss der Browser aber kompromittiert oder der Cookie über eine nichtverschlüsselte Verbindung übertragen werden. Das ist oft als Risiko tragbar.

Mit einem Token-Ansatz ist es möglich, dass die Microservices sich nicht um die Autorisierung des Aufrufers kümmern müssen, aber dennoch den Zugriff auf bestimmte Benutzergruppen oder Rollen einschränken können.

Es gibt gute Gründe für die Nutzung von OAuth2:

- Es gibt zahlreiche Bibliotheken für praktisch alle gängigen Programmiersprachen, die OAuth2 oder einen OAuth2-Server implementieren [19]. Die Entscheidung für OAuth2 schränkt die Auswahl bei der Technologie für die Microservices kaum ein.
- Zwischen den Microservices muss nur noch das Access Token übertragen werden. Das kann standardisiert über einen HTTP-Header geschehen, wenn REST genutzt wird. Bei anderen Kommunikationsprotokollen können ähnliche Mechanismen verwendet werden. Auch in diesem Bereich schränkt OAuth2 die Technologie-Auswahl kaum ein.
- Durch JWT können in dem Token die Informationen hinterlegt werden, die der Authorization Server den Microservices mitteilt, damit die Microservices den Zugriff erlauben oder verbieten können. Damit ist auch in diesem Bereich das Zusammenspiel zwischen dem einzelnen Microservice und der gemeinsamen Infrastruktur einfach umsetzbar – mit Standards, die breit unterstützt werden.

Spring Cloud Security [20] bietet gerade für Java-basierte Microservices eine Basis, um OAuth2-Systeme zu implementieren.

OAuth2 löst vor allem das Problem der Authentifizierung und Autorisierung – und zwar in erster Linie für menschliche Benutzer. Es gibt weitere Maßnahmen, die ein System aus Microservices absichern können:

- Die Kommunikation zwischen den Microservices kann durch SSL/TLS gegen Abhören abgesichert werden. Alle Kommunikation wird dann verschlüsselt. Infrastrukturen wie REST oder Messaging-Systeme unterstützen meistens solche Protokolle.
- Neben der Authentifizierung von Systemen mit OAuth2 können auch Zertifikate genutzt werden, um Clients zu authentifizieren. Eine Zertifizierungsinstanz erstellt die Zertifikate. Sie können genutzt werden, um digitale Unterschriften zu überprüfen. Damit ist es möglich, einen Client anhand seiner digitalen Unterschrift zu authentifizieren. Da SSL/TLS Zertifikate unterstützt, ist zumindest auf dieser Ebene eine Nutzung von Zertifikaten und Authentifizierung mit Zertifikaten möglich.
- Ein ähnliches Konzept sind API Keys. Sie werden an externe Clients gegeben, damit diese das System nutzen können. Durch den API Key authentifizieren sich die externen Clients und können die passenden Rechte bekommen. Das kann bei OAuth2 beispielsweise mit den Client Credential umgesetzt werden.

- Firewalls können genutzt werden, um die Kommunikation zwischen den Microservices abzusichern. Normalerweise schützen Firewalls ein System gegen den unberechtigten Zugriff von außen. Eine Firewall für die Kommunikation zwischen den Microservices verhindert, dass bei einer erfolgreichen Übernahme eines Microservice die anderen Microservices gefährdet sind. So kann der Einbruch gegebenenfalls auf nur einen Microservice begrenzt werden.
- Schließlich sollte es eine Intrusion Detection geben, um unberechtigte Zugriffe auf das System zu erkennen. Dieses Thema ist eng verwandt mit Monitoring. Das Monitoring-System kann auch genutzt werden, um bei einem Einbruch den passenden Alarm auszulösen.
- Schließlich ist Datensparsamkeit ein interessantes Konzept. Es kommt eigentlich aus dem Datenschutz und besagt, dass nur die Daten gespeichert werden sollen, die unbedingt benötigt werden. Der Vorteil aus einer Sicherheitsperspektive ist, dass so größere Datensammlungen vermieden werden. Dadurch werden die Angriffsziele weniger attraktiv und außerdem sind die Konsequenzen bei einem Einbruch nicht so signifikant.

Ein Werkzeug, das viele Probleme im Bereich Sicherheit von Microservices löst, ist Hashicorp Vault [24]. Es bietet folgende Features:

- *Secrets* wie Passwörter, API Keys, Schlüssel für Verschlüsselungen oder Zertifikate können abgespeichert werden. Das kann nützlich sein, damit Benutzer ihre Secrets verwalten können. Aber auch Microservices können so mit Zertifikaten ausgestattet werden, um die Kommunikation untereinander oder mit externen Services abzusichern.
- Secrets werden über einen *Lease* an Services ausgegeben. Außerdem können sie mit einer Zugriffskontrolle versehen werden. Dadurch kann bei einem kompromittierten Service das Problem begrenzt werden. So können Secrets auch ungültig gemacht werden.
- Daten können gleich mit den Schlüsseln *verschlüsselt* oder *entschlüsselt* werden, ohne dass ein Microservice diese Schlüssel selber speichern muss.
- Zugriffe werden durch ein *Audit* nachvollziehbar. Daraus wird klar, wer welches Secret wann bekommen hat.
- Im Hintergrund kann Vault HSMs, SQL-Datenbanken oder Amazon IAM nutzen, um Secret *abzuspeichern*. Es kann auch beispielsweise neue Zugriffsschlüssel für die Amazon Cloud selbstständig generieren.

Vault löst so also die Handhabung von Schlüsseln und entlastet die Microservices von dieser Aufgabe. Mit Schlüsseln wirklich sicher umzugehen, ist eine Herausforderung. Es ist schwierig, so etwas wirklich sicher zu implementieren.

Sicherheit hat bei einer Software-Architektur ganz unterschiedliche Ausprägungen. Ansätze wie OAuth2 helfen nur bei der Vertraulichkeit. Sie verhindern, dass Daten für unautorisierte Nutzer zugreifbar sind. Aber selbst die Vertraulichkeit löst OAuth2 alleine nicht vollständig: Die

Kommunikation im Netzwerk muss ebenfalls gegen Abhören abgesichert werden – beispielsweise mit HTTPS oder anderen Arten der Verschlüsselung.

Weitere Sicherheitsaspekte sind:

- *Integrität* bedeutet, dass es keine unbemerkt Änderungen an Daten gibt. Dieses Problem muss jeder Microservice lösen. Beispielsweise können Daten signiert werden, um sicherzustellen, dass sie nicht auf irgendeine Weise manipuliert worden sind. Die konkrete Implementierung muss jeder Microservice vornehmen.
- Bei der *Verbindlichkeit* geht es darum, dass Änderungen nicht abgestritten werden können. Das kann erreicht werden, indem die Änderungen jedes Benutzers mit einem eigenen Schlüssel signiert werden. Dann ist klar, dass genau dieser Benutzer die Daten geändert hat. Die übergreifende Sicherheitsarchitektur muss den Schlüssel bereitstellen, das Signieren ist dann eine Aufgabe jedes Service.
- Die *Datensicherheit* ist gewährleistet, wenn die Daten nicht verloren gehen. Zu den Lösungen zählen Backup-Lösungen und hochverfügbare Speicherlösungen. Dieses Problem müssen die Microservices angehen, da es als Teil des Datenhaushalts in ihrer Verantwortung ist. Allerdings kann die allgemeine Infrastruktur bestimmte Datenbanken anbieten, die mit passenden Backup- und Desaster-Recovery-Mechanismen ausgestattet sind.
- Bei der *Verfügbarkeit* geht es darum, dass ein System zur Verfügung steht. Auch hier müssen die Microservices einzeln ihren Beitrag leisten. Da aber gerade bei Microservices-Architekturen mit dem Ausfall einzelner Microservices gerechnet werden muss, sind Microservice-Systeme in diesem Bereich oft gut vorbereitet. Resilience ([Abschnitt 10.5](#)) ist beispielsweise dafür nützlich.

Oft werden bei der Sicherheit diese Aspekte nicht betrachtet – aber der Ausfall eines Systems hat oft sogar schlimmere Folgen als der unberechtigte Zugriff auf Daten. Eine Gefahr sind Denial-of-Service-Attacken, bei denen Server so überlastet werden, dass sie keine sinnvolle Arbeit mehr vollbringen können. Die technischen Hürden dafür sind oft erschreckend niedrig und die Verteidigung gegen solche Attacken dafür umso schwieriger.

8.13 Dokumentation und Metadaten

Um in einer Microservices-Architektur den Überblick zu behalten, müssen bestimmte Informationen über jeden Microservice verfügbar sein. Daher muss die Microservices-Architektur definieren, wie Microservices solche Informationen zur Verfügung stellen können. Nur wenn alle Microservices diese Informationen einheitlich zur Verfügung stellen, können die Informationen einfach zusammengetragen werden. Mögliche Informationen können beispielsweise sein:

- Grundlegende Informationen wie der Name des Service und die verantwortlichen Ansprechpartner.
- Informationen über den Source-Code: Wo der Code in der Versionskontrolle zu finden ist und welche Bibliotheken er verwendet. Die verwendeten Bibliotheken können interessant sein, um die Open-Source-Lizenzen der Bibliotheken mit den

Unternehmensregeln abzugleichen oder bei einer Sicherheitslücke in einer Bibliothek die betroffenen Microservices zu identifizieren. Für solche Zwecke müssen die Informationen verfügbar sein, auch wenn die Entscheidung für die Nutzung einer bestimmten Bibliothek eigentlich nur einen Microservice beeinflusst, wobei die Entscheidung von dem zuständigen Team weitgehend unabhängig getroffen werden kann.

- Eine weitere interessante Information ist, mit welchen anderen Microservices der Microservice zusammenarbeitet. Diese Information ist für das Architekturmanagement (siehe [Abschnitt 8.2](#)) zentral.
- Ebenso können Informationen über die Konfigurationsparameter oder über Feature Toggles interessant sein. Mit Feature Toggles können Features ein- oder ausgeschaltet werden, um neue Features erst in Produktion freizuschalten, wenn sie wirklich zu Ende implementiert sind, oder durch das Deaktivieren eines Features den Ausfall eines Service zu vermeiden.

Es ist nicht sinnvoll, alle Bestandteile der Microservices zu dokumentieren oder die gesamte Dokumentation zu vereinheitlichen. Eine Vereinheitlichung ist nur für Informationen sinnvoll, die außerhalb des Teams relevant sind, das den Microservice umsetzt. Wenn ein Management des Zusammenspiels der Microservices notwendig ist oder eine Überprüfung der Lizenzen, müssen die dafür relevanten Informationen verfügbar sein, und zwar außerhalb des für den Microservice verantwortlichen Teams. Diese Fragestellungen müssen über Microservices übergreifend geklärt werden. Jedes Team kann für die eigenen Microservices weitere Dokumentation anlegen. Sie ist aber nur für das Team relevant und muss daher nicht standardisiert sein.

Ein übliches Problem bei der Dokumentation jeglicher Software ist, dass die Dokumentation veraltet und dann gegebenenfalls einen Stand dokumentiert, der nicht mehr aktuell ist. Daher sollte die Dokumentation zusammen mit dem Code versioniert werden. Außerdem sollte die Dokumentation aus Informationen generiert werden, die sowieso im System vorhanden sind. Beispielsweise kann die Liste aller verwendeten Bibliotheken aus dem Build-System entnommen werden, da bei der Kompilierung des Systems genau diese Informationen benötigt werden. Welche anderen Microservices genutzt werden, kann aus der Service Discovery entnommen oder beispielsweise für die Generierung von Firewall-Regeln genutzt werden, wenn mit einer Firewall die Kommunikation zwischen den Microservices abgesichert wird. So muss die Dokumentation nicht extra gepflegt werden, sondern sie ist ein Ergebnis der sowieso vorhandenen Informationen.

Die Dokumentation kann ein Teil der beim Build erzeugten Artefakte sein. Ebenso kann es eine Schnittstelle zur Laufzeit geben, mit der Metadaten ausgelesen werden können. Eine solche Schnittstelle kann den sonst üblichen Schnittstellen für Monitoring entsprechen und beispielsweise über HTTP JSON-Dokumente bereitstellen. So sind die Metadaten nur eine weitere Information, die Microservices zur Laufzeit bereitstellen.

In einem Service-Template kann beispielhaft gezeigt werden, wie die Dokumentation erzeugt wird. Das Service-Template kann dann die Basis für die Umsetzung von neuen

Microservices sein. Wenn das Service-Template diesen Aspekt bereits enthält, erleichtert es die Umsetzung einer standardkonformen Dokumentation. Ebenso können zumindest formale Eigenschaften der Dokumentation durch einen Test abgeprüft werden.

8.14 Fazit

Die fachliche Architektur von Microservice-Systemen ist essenziell, weil sie neben der Struktur des Systems auch die Organisation beeinflusst ([Abschnitt 8.1](#)). Leider sind gerade bei Microservices Werkzeuge für das Management der Abhängigkeiten rar gesät, sodass Teams eigene Lösungen entwickeln müssen. Oft reicht aber ein Verständnis über die Implementierung von einzelnen Geschäftsprozessen und ein Überblick über die Gesamtarchitektur ist gar nicht notwendig ([Abschnitt 8.2](#)).

Für erfolgreiche Architekturen ist eine ständige Anpassung der Architektur notwendig. Bei Deployment-Monolithen gibt es dafür zahlreiche Refactoring-Techniken. Solche Möglichkeiten gibt es bei Microservices auch – aber ohne Tool-Unterstützung und mit viel höheren Hürden ([Abschnitt 8.3](#)). Dennoch können Microservice-Systeme sinnvoll weiterentwickelt werden – beispielsweise indem man zunächst mit wenigen großen Microservices beginnt und über die Zeit sowieso immer mehr Microservices entstehen lässt ([Abschnitt 8.4](#)). Eine frühzeitige Aufteilung in viele Microservices birgt das Risiko, dass die Aufteilung falsch ist.

Ein Sonderfall ist die Migration von einer Legacy-Anwendung zu einer Microservices-Architektur ([Abschnitt 8.5](#)). In diesem Fall kann die Code-Basis der Legacy-Anwendung in Microservices aufgeteilt werden – aber das kann wegen der oft schlechten Struktur der Legacy-Anwendung zu einer schlechten Architektur führen. Alternativ kann die Legacy-Anwendung durch Microservices ergänzt werden, die schrittweise Funktionalitäten der Legacy-Anwendung ablösen.

Event-driven Architecture ([Abschnitt 8.6](#)) kann die Logik in den Microservices sehr stark voneinander entkoppeln. Dadurch ist das System einfach erweiterbar.

Zu den Aufgaben einer Architektur gehört auch die Definition der technologischen Basis ([Abschnitt 8.7](#)). Bei Microservice-Systemen geht es weniger um einen gemeinsamen Technologie-Stack für die Implementierung, sondern um die Definition der gemeinsamen Kommunikationsprotokolle oder der Schnittstellen hin zum Monitoring oder Logging. Weitere technische Funktionen für das Gesamtsystem sind Koordination und Konfiguration ([Abschnitt 8.8](#)). In diesem Bereich können Werkzeuge ausgewählt werden, die alle Microservices nutzen müssen. Oder man verzichtet auf eine zentrale Konfiguration und lässt stattdessen jeden Microservice seine eigene Konfiguration mitbringen.

Bei Service Discovery ([Abschnitt 8.9](#)) kann ebenfalls eine bestimmte Technologie ausgewählt werden. Eine Lösung für Service Discovery ist bei einem Microservice-System auf jeden Fall sinnvoll – es sei denn, es wird Messaging zur Kommunikation genutzt. Auf Basis der Service Discovery sollte Load Balancing eingeführt werden ([Abschnitt 8.10](#)), um so die Last auf die Instanzen der Microservices zu verteilen. Service Discovery kennt alle Instanzen, das Load Balancing verteilt die Last auf diese Instanzen.

Load Balancing kann über einen zentralen Load Balancer, über die Service Discovery oder über einen Load Balancer pro Client implementiert werden. Das legt die Basis für Skalierbarkeit ([Abschnitt 8.11](#)). So kann ein Microservice mehr Last abarbeiten, indem er hochskaliert.

Microservices haben eine wesentlich höhere technische Komplexität als Deployment-Monolithen. Betriebssysteme, Netzwerke, Load Balancer, Service Discovery und Kommunikationsprotokolle werden zu Teilen der Architektur. Von diesen Aspekten sind Entwickler und Architekten von Deployment-Monolithen weitgehend verschont. So müssen Architekten sich mit ganz anderen Technologien beschäftigen und auf ganz anderen Ebenen Architektur betreiben.

Im Bereich Sicherheit muss eine zentrale Komponente mindestens die Authentifizierung und Teile der Autorisierung übernehmen. Die Microservices sollten dann die Details des Zugriffs regeln ([Abschnitt 8.12](#)). Um in einem System aus vielen Microservices bestimmte Informationen zu haben, müssen Microservices eine standardisierte Dokumentation haben ([Abschnitt 8.13](#)). Die kann beispielsweise Auskunft über die verwendeten Bibliotheken geben – um sie gegen Open-Source-Lizenzregeln abzugleichen oder Sicherheitslücken zu beseitigen, wenn eine Bibliothek eine Lücke hat.

Die Architektur eines Microservice-Systems ist anders als die klassischer Anwendungen. Viele Entscheidungen werden erst in den Microservices getroffen, während Themen wie Monitoring, Logging oder Continuous Delivery übergreifend standardisiert werden.

Wesentliche Punkte

- Refactoring zwischen Microservices ist aufwendig. Daher ist eine Änderung der Architektur auf dieser Ebene schwierig. Ein besonderes Augenmerk muss auf der Weiterentwicklung der Architektur liegen.
- Wesentlicher Teil der Architektur ist die Definition übergreifender Technologien für Konfiguration und Koordination, Service Discovery, Load Balancing, Sicherheit, Dokumentation und Metadaten.

8.15 Links und Literatur

- [1] <http://dev.otto.de/2013/04/14/architekturprinzipien-2/>
- [2] <http://structure101.com/>
- [3] Martin Fowler: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999, ISBN 978-0201485677
- [4] Sam Newman: Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2015, ISBN 978-1-4919-5035-7
- [5] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 978-0-32112-521-7
- [6] Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building,

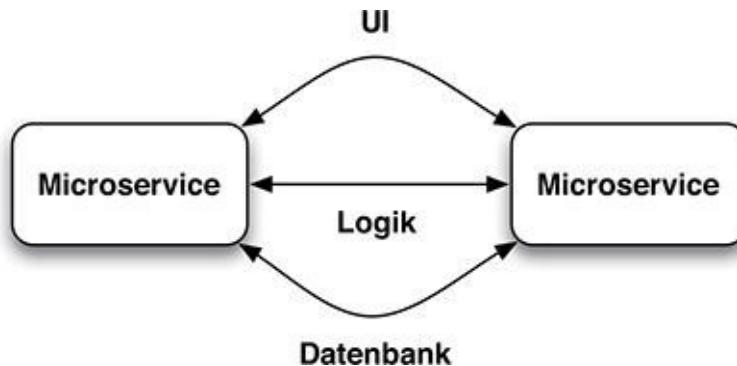
and Deploying Messaging Solutions, Addison-Wesley Longman, 2003, ISBN 978-0-32120-068-6

- [7] <http://www.eaipatterns.com/toc.html>
- [8] <https://zookeeper.apache.org/>
- [9] <https://github.com/coreos/etcd>
- [10] <http://cloud.spring.io/spring-cloud-config/>
- [11] <https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul>
- [12] <http://www.zytrax.com/books/dns/>
- [13] <https://www.isc.org/downloads/bind/>
- [14] <https://github.com/Netflix/eureka>
- [15] <http://www.consul.io>
- [16] <http://tools.ietf.org/html/rfc6749>
- [17] <https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32>
- [18] <http://tools.ietf.org/html/rfc4556>
- [19] <http://oauth.net/2/>
- [20] <http://cloud.spring.io/spring-cloud-security/>
- [21] <http://gephi.github.io/>
- [22] <http://jqassistant.org>
- [23] <https://www.oasis-open.org/committees/security/>
- [24] <https://www.vaultproject.io/>

9 Integration und Kommunikation

Microservices müssen integriert werden und kommunizieren. Das ist auf unterschiedlichen Ebenen möglich (Abb. 9–1). Jeder dieser Ansätze hat bestimmte Vor- und Nachteile. Außerdem sind unterschiedliche technische Umsetzungen der Integration auf der jeweiligen Ebene möglich.

Abb. 9–1 Integration von Microservices auf unterschiedlichen Ebenen



- Microservices enthalten eine grafische Benutzeroberfläche. Daher können Microservices auf Ebene der UI integriert werden. Diese Integration stellt [Abschnitt 9.1](#) vor.
- Auch die Logik kann in den Services integriert werden. Dazu können die Microservices REST ([Abschnitt 9.2](#)), SOAP oder RCP ([Abschnitt 9.3](#)) oder Messaging ([Abschnitt 9.4](#)) nutzen.
- Schließlich ist eine Integration auf Ebene der Daten möglich. Dazu dient Datenreplikation ([Abschnitt 9.5](#)).

Generelle Regeln für das Design von Schnittstellen gibt [Abschnitt 9.6](#).

9.1 Web und UI

Microservices sollten ihre eigene UI mitbringen. So können neue Funktionalitäten auch dann in nur einem Microservice umgesetzt werden, wenn die Änderungen auch die UI beeinflussen. Für das Gesamtsystem ist es notwendig, die UIs der Microservices miteinander zu integrieren. Dazu sind unterschiedliche Ansätze denkbar. [13] gibt einen Überblick.

Eine Single-Page-App (SPA) [1] setzt mit nur einer HTML-Seite die gesamte UI um. Die Logik ist in JavaScript implementiert, das Teile der Seite dynamisch anpasst. Die Logik kann die im Browser angezeigte URL manipulieren, sodass Bookmarks und andere typische Browser-Features genutzt werden können. SPAs entsprechen jedoch nicht dem ursprünglichen Webgedanken: SPAs drängen HTML als zentrale Technologie des Webs an den Rand. Die meiste Logik ist in JavaScript umgesetzt. Klassische Webarchitekturen setzen Logik fast ausschließlich auf dem Server um.

Mehrere Single-Page-Apps

SPAs haben vor allem Vorteile, wenn komplexe Interaktionen oder Offline-Fähigkeit notwendig sind. Ein Beispiel, das den Begriff SPA auch entscheidend geprägt hat, ist Googles GMail. Mail-Clients sind sonst oft native Anwendungen. GMail als SPA bietet fast denselben Komfort.

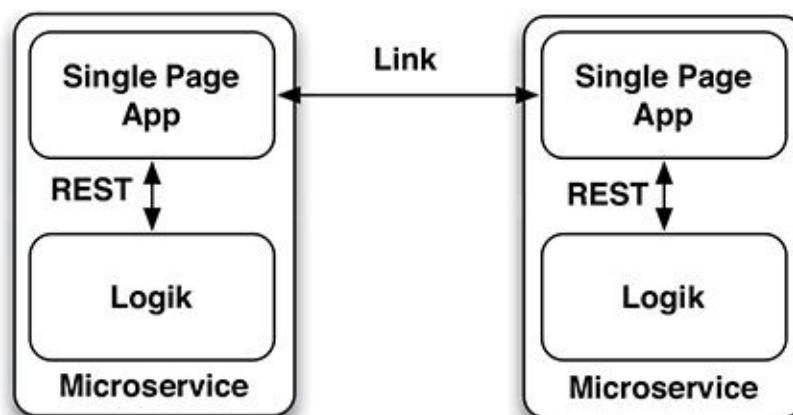
Es gibt unterschiedliche Technologien für die Implementierung von Single-Page-Apps:

- Sehr beliebt ist *AngularJS* [8]. AngularJS hat unter anderem ein bidirektionales UI-Daten-Binding: Wird einem Attribut eines gebundenen View-Modells im JavaScript-Code ein neuer Wert zugewiesen, werden die UI-Komponenten, die den Wert anzeigen, automatisch geändert. Von UI in den Code hinein funktioniert das Binding ebenfalls: Die Eingabe eines Nutzers kann AngularJS an eine JavaScript-Variablen binden. Darüber hinaus kann AngularJS HTML-Templates im Browser rendern. So kann JavaScript-Code auch komplexe DOM-Strukturen erzeugen. Dann wird die gesamte Frontend-Logik in den Browser verlegt. Hinter AngularJS steht Google, die das Framework unter die sehr liberale MIT License gestellt haben.
- *Ember.js* [9] setzt auf das Prinzip Convention over Configuration und bildet im Kern dieselben Features wie AngularJS ab. Mit dem Zusatzmodul Ember Data bietet es einen modellgetriebenen Ansatz, um auf REST-Ressourcen zuzugreifen. Ember.js ist nach der MIT License lizenziert und wird von einigen Entwicklern aus der Open-Source-Community gepflegt.
- *Ext JS* [10] bietet neben einem MVC-Ansatz auch Komponenten, die Entwickler ähnlich wie bei Rich-Client-Anwendungen zu einer UI zusammenstellen können. Ext JS ist unter der GPL v3.0 als Open Source erhältlich. Für die kommerzielle Entwicklung muss allerdings eine Lizenz vom Hersteller Sencha gekauft werden.

SPA pro Microservice

Für Microservices mit Single-Page-Apps kann jeder Microservice seine eigene SPA mitbringen (Abb. 9–2). Die SPA kann den Microservice zum Beispiel per JSON/REST ansprechen. Das ist mit JavaScript besonders einfach umsetzbar. Zwischen den SPAs kann ein Link genutzt werden.

Abb. 9–2 Microservices mit Single-Page-Apps



Die SPAs sind dadurch völlig getrennt und unabhängig. Neue Versionen eines SPA und des dazugehörigen Microservice können ohne Weiteres ausgerollt werden. Allerdings ist eine engere Integration der SPAs schwierig. Wenn der Nutzer von einer SPA auf eine

andere umschaltet, lädt der Browser eine neue Webseite und startet eine andere JavaScript-Anwendung. Dafür benötigen auch moderne Browser so viel Zeit, dass dieser Ansatz nur sinnvoll ist, wenn das Umschalten zwischen den SPAs die Ausnahme ist.

Außerdem können die SPAs uneinheitlich sein. Jede kann eine völlig eigene UI mit einem eigenen Design mitbringen. Das kann allerdings gelöst werden, indem ein Asset-Server genutzt wird. Ein solcher Server dient dazu, JavaScript-Dateien und CSS-Dateien für die Anwendungen zur Verfügung zu stellen. Wenn die SPAs der Microservices diese Ressourcen nur vom Asset-Server nutzen dürfen, kann so eine einheitliche Oberfläche erreicht werden. Um das zu erreichen, kann ein Proxy-Server Anfragen an den Asset-Server und die Microservices verteilen. So scheint es für einen Webbrower, als würden alle Ressourcen und auch die Microservices eine gemeinsame URL haben. Das Vorgehen vermeidet, dass Sicherheitsregeln die Nutzung der Inhalte unterbinden, weil sie von verschiedenen URLs kommen. Caching kann dann die Ladezeit der Anwendungen reduzieren. Wenn nur JavaScript-Bibliotheken genutzt werden dürfen, die auf dem Asset-Server abgelegt sind, kann die Auswahl der Technologien für die Microservices reduziert werden. Einheitlichkeit und die freie Technologiewahl können konkurrierende Ziele sein.

Außerdem entstehen durch die gemeinsamen Assets Code-Abhängigkeiten zwischen dem Asset-Server und allen Microservices. Eine neue Version eines Assets zieht eine Änderung aller Microservices nach sich, die dieses Asset verwenden. Schließlich müssen sie so geändert werden, dass sie die neue Version auch tatsächlich nutzen. Solche Code-Abhängigkeiten gefährden das unabhängige Deployment und sollten daher vermieden werden. Auf Ebene des Backend-Codes sind Code-Abhängigkeiten oft ein Problem (siehe [Abschnitt 8.3](#)). Eigentlich sollten auch beim Frontend solche Abhängigkeiten reduziert werden. Dann ist ein Asset-Server aber eher ein Problem als eine Lösung.

Neben einem Asset-Server können UI-Richtlinien hilfreich sein, die das Design der Anwendungen genauer beschreiben und so einen einheitlichen Ansatz auch auf anderen Ebenen ermöglichen. So kann eine einheitliche UI auch ohne gemeinsamen Asset-Server und Code-Abhängigkeiten realisiert werden.

Ebenso muss gewährleistet sein, dass die SPAs eine einheitliche Authentifizierung und Autorisierung haben, sodass sich Nutzer nicht mehrfach einloggen müssen. Dazu kann OAuth2 oder ein gemeinsamer signierter Cookie eine Lösung sein (siehe auch [Abschnitt 8.12](#)).

JavaScript kann nur auf Daten zugreifen, die unter der Domäne verfügbar sind, von der auch der JavaScript-Code stammt. Diese Same Origin Policy vermeidet, dass JavaScript-Code Daten von anderen Domänen auslesen kann. Wenn alle Microservices wegen eines Proxys nach außen unter derselben Domäne zugreifbar sind, ist das keine Einschränkung. Sonst muss die Policy deaktiviert werden, wenn die Oberfläche eines Microservice auf die Daten eines anderen Microservice zugreifen soll. Dieses Problem kann CORS (Cross Origin Resource Sharing) lösen, mit dem der Server, der die Daten ausliefert, auch JavaScript von anderen Domänen erlauben kann. Eine andere Möglichkeit ist es, alle SPA und REST-Services von außen nur unter einer Domäne anzubieten, sodass kein Zugriff über Domänen hinweg notwendig ist. Auf diese Weise kann auch ein Zugriff auf gemeinsamen JavaScript-Code auf einem Asset-Server umgesetzt werden.

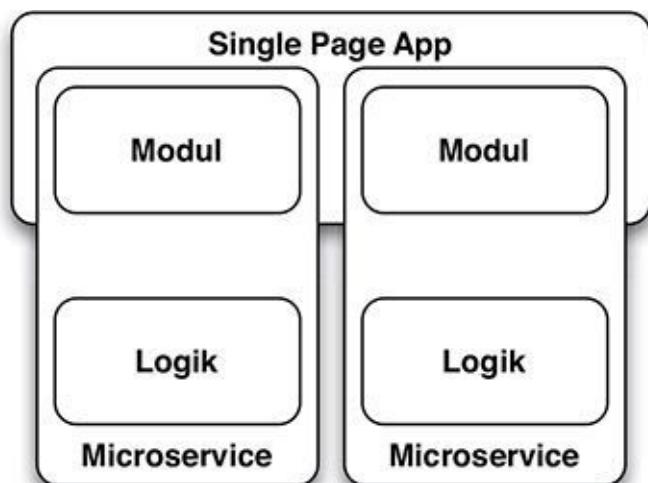
Asset-Server für Einheitlichkeit

Eine Single-Page-App für alle Microservices

Die Aufteilung in mehrere SPAs ergibt eine strikte Trennung der Frontends der Microservices. Wenn beispielsweise eine SPA für die Erfassung von Bestellungen zuständig ist und eine andere für einen grundverschiedenen Anwendungsfall wie Reports, sind die Ladezeiten bei dem Wechsel zwischen den SPAs noch akzeptabel. Vielleicht sind sogar die Benutzergruppen unterschiedlich, sodass ein Wechsel zwischen den Anwendungen nicht stattfindet.

Es gibt Fälle, in denen eine engere Integration der Oberflächen der Microservices notwendig ist. So können in einer Bestellung auch die Details zu den Waren angezeigt werden. Die Darstellung der Bestellung wird von einem Microservice verantwortet, die Darstellung der Waren von einem anderen. In dem Fall kann die SPA in Module aufgeteilt werden. Jedes Modul gehört zu einem anderen Microservice und damit zu einem anderen Team. Die Module sollten getrennt deployt werden. Sie können beispielsweise in eigenen JavaScript-Dateien auf dem Server liegen und getrennte Continuous-Delivery-Pipelines haben. Außerdem muss es passende Konventionen für die Schnittstelle geben. Beispielsweise kann nur das Schicken von Events erlaubt werden. Events entkoppeln die Module, weil die Module nur Änderungen in den Zuständen mitteilen, aber nicht, wie andere Module darauf reagieren sollen.

Abb. 9–3 Enge Integration von Microservices in einer Single-Page-App



AngularJS zum Beispiel hat ein Modul-Konzept, mit dem einzelne Teile der SPA in getrennten Einheiten implementiert werden können. Ein Microservice könnte ein AngularJS-Modul haben, mit dem die Oberfläche des Microservice angezeigt wird. Das Modul kann gegebenenfalls AngularJS-Module anderer Microservices integrieren.

Ein solches Vorgehen hat jedoch Nachteile:

- Das Deployment der SPA ist meistens nur als vollständige Anwendung möglich. Wenn ein Modul geändert wird, muss die gesamte SPA neu gebaut und ausgeliefert werden. Das muss zwischen den Microservices koordiniert werden, die Module der Anwendung bereitstellen. Auch das Deployment der Microservices auf dem Server muss mit dem Deployment der Module koordiniert werden, da die Microservices von den Modulen aufgerufen werden. Solche Koordination beim Deployment von Modulen einer Anwendung sollen Microservices vermeiden.

- Die Module können sich gegenseitig aufrufen. Abhängig davon, wie die Aufrufe implementiert sind, können Änderungen an einem Modul erzwingen, dass auch andere Module geändert werden, weil sich beispielsweise eine Schnittstelle geändert hat. Wenn die Module zu unterschiedlichen Microservices gehören, erzwingt das wiederum eine Koordination über die Microservices hinweg, was eigentlich vermieden werden soll.

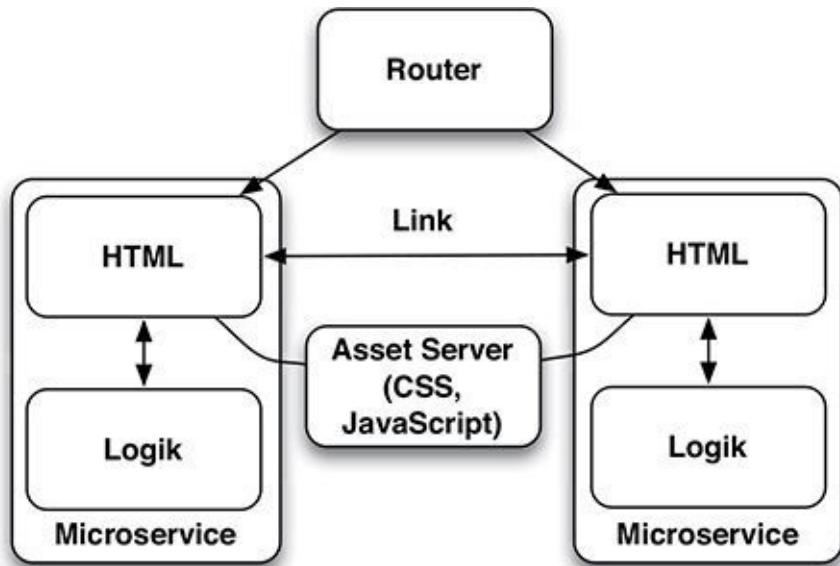
Bei SPA-Modulen ist eine wesentlich engere Abstimmung notwendig, als es bei Links zwischen den Anwendungen der Fall wäre. Dafür können dem Nutzer gleichzeitig UI-Elemente aus verschiedenen Microservices angezeigt werden. Letztendlich werden aber die Microservices auf Ebene der UI eng aneinandergekoppelt. Die Module in der SPA entsprechen den Modulkonzepten, die es in anderen Programmiersprachen auch gibt, und erzwingen ein gemeinsames Deployment. Die Microservices, die eigentlich unabhängig sein sollten, treffen sich auf der UI-Ebene in einem gemeinsamen Deployment-Artefakt. Damit macht dieser Ansatz einige wichtige Vorteile einer Microservices-Architektur – das unabhängige Deployment – zunichte.

HTML-Anwendungen

Eine andere Möglichkeit, die Benutzeroberfläche umzusetzen, sind HTML-basierte Oberflächen. Jeder Microservice hat eine oder mehrere Webseiten, die auf dem Server erzeugt werden. Die Seiten können auch JavaScript nutzen. Beim Wechsel zwischen den Seiten wird im Gegensatz zu SPAs nur eine neue HTML-Seite vom Server geladen und nicht zwangsläufig eine Anwendung geladen und gestartet.

ROCA (Resource Oriented Client Architecture) [2] ist eine Möglichkeit, in HTML-Oberflächen den Umgang mit JavaScript und dynamischen Elementen zu gestalten. ROCA positioniert sich als Alternative zu SPAs. Die Rolle von JavaScript ist in ROCA darauf beschränkt, die Nutzbarkeit der Seiten zu optimieren. JavaScript kann die Nutzung vereinfachen oder Effekte zu den HTML-Seiten hinzufügen. Aber die Anwendung muss auch ohne JavaScript nutzbar bleiben. Es geht bei ROCA nicht darum, dass Benutzer Webseiten tatsächlich ohne JavaScript verwenden. Die Anwendungen sollen nur die Architektur des Webs nutzen, die auf HTML und HTTP basiert. Gerade wenn eine Webanwendung in Microservices aufgeteilt werden soll, reduziert ROCA die Abhängigkeiten und vereinfacht eine Aufteilung. Zwischen Microservices kann die Kopplung der UI durch Links geschehen. Bei HTML-Anwendungen sind Links für die Navigation zwischen Seiten üblich und stellen eine natürliche Integration dar. Sie sind kein Fremdkörper wie bei SPAs.

Abb. 9–4 HTML-Oberflächen mit Asset-Server



Um eine Einheitlichkeit der HTML-Oberflächen zu unterstützen, können die Microservices wie auch bei SPAs einen gemeinsamen Asset-Server verwenden (Abb. 9–4). Er enthält alle CSS und JavaScript-Bibliotheken. Wenn die Teams außerdem Design-Richtlinien für die HTML-Seiten definieren und die Assets auf dem Asset-Server pflegen, wird die Oberfläche der verschiedenen Microservices weitgehend identisch sein. Allerdings ergeben sich dann wie schon dargestellt Code-Abhängigkeiten zwischen den UIs der Microservices.

Nach außen sollten die Microservices wie eine einzige Webanwendung erscheinen – idealerweise mit einer URL. Einfaches Routing

Das erleichtert auch die gemeinsame Nutzung von Assets, weil die Same Origin Policy nicht verletzt wird. Von außen müssen die Anfragen eines Nutzers aber auf den richtigen Microservice gelenkt werden. Dazu dient der Router. Er kann HTTP-Anfragen entgegennehmen und sie an einen der Microservices weiterleiten. Das kann anhand der URL geschehen. Wie einzelne URLs auf Microservices abgebildet werden, kann durch Regeln entschieden werden, die auch komplex sein können. Die Beispielanwendung verwendet für diese Aufgabe Zuul (siehe Abschnitt 14.9). Alternativen sind Reverse Proxies. Das sind Webserver wie Apache httpd oder nginx, die Anfragen an andere Server weitergeben können. Dabei kann in die Anfragen eingegriffen und beispielsweise können URLs umgeschrieben werden. Diese Mechanismen sind aber nicht so flexibel wie Zuul, das sehr einfach mit eigenem Code erweitert werden kann.

Wenn die Logik im Router sehr komplex ist, kann das zu Problemen führen. Muss diese Logik geändert werden, weil eine neue Version eines Microservice in Produktion gebracht wird, ist ein isoliertes Deployment nicht mehr einfach möglich. Das gefährdet die unabhängige Entwicklung und das unabhängige Deployment der Microservices.

In einigen Fällen ist eine engere Integration notwendig. Es kann vorkommen, dass auf einer HTML-Seite Informationen angezeigt werden, die aus unterschiedlichen Microservices stammen. Beispielsweise die Daten einer Bestellung aus einem Microservice und die Daten der enthaltenen Waren aus einem anderen Microservice. Dann reicht ein Router nicht mehr aus. Er kann nur dafür sorgen, dass ein Microservice jeweils eine vollständige HTML-Seite erzeugt. HTML mit JavaScript zusammenstellen

Eine einfache Lösung, die mit der Architektur aus [Abbildung 9–4](#) arbeitet, basiert auf Links. Der Inhalt des Links auf einen anderen Microservice kann durch AJAX geladen werden. Der Link wird dann durch das dadurch erhaltene HTML ersetzt. Im Beispiel könnte ein Link auf ein Produkt zu einer Darstellung dieses Produktes umgewandelt werden. So ist es möglich, dass die Logik für die Darstellung eines Produktes in einem Microservice implementiert ist, während das Design der Gesamtseite in einem anderen Microservice umgesetzt ist. Die Gesamtseite wäre in der Verantwortung des Bestellung-Microservice, während die Darstellung des Produktes in der Verantwortung des Produkt-Microservice wäre. Das erlaubt eine unabhängige Weiterentwicklung der beiden Microservices und der Darstellung dieser beiden Teile. Wenn die Darstellung der Ware geändert werden soll oder neue Produkte eine andere Darstellung erzwingen, ist diese Änderung im Produkt-Microservice umsetzbar. Die gesamte Logik des Bestellung-Microservice bleibt unverändert.

Ein anderes Beispiel für diesen Ansatz ist Facebooks BigPipe [14]. Es optimiert nicht nur die Ladezeit, sondern erlaubt auch das Komponieren von Webseiten aus Pagelets. Eine eigene Implementierung kann mit JavaScript bestimmte Elemente in der Seite durch anderes HTML ergänzen. Das können Links oder div-Elemente sein, wie sie sonst auch für die Strukturierung von Webseiten genutzt werden. Ein solches div-Element kann dann durch anderen HTML-Code ersetzt werden.

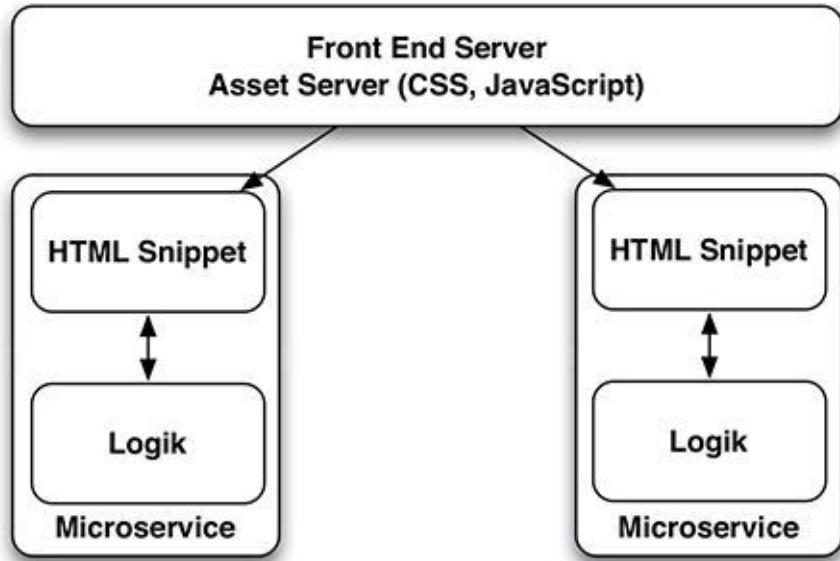
Dieser Ansatz führt jedoch zu einer recht langen Ladezeit. Er bietet sich vor allem, wenn die Web-UI sowieso viel JavaScript nutzt und nicht viel zwischen Seiten wechselt.

Frontend-Server

Eine Alternative für eine enge Integration zeigt [Abbildung 9–5](#). Ein Frontend-Server setzt die HTML-Seite aus HTML-Schnipseln zusammen, die jeweils von einem Microservice erzeugt werden. Im Frontend-Server können auch Assets wie CSS oder JavaScript-Bibliotheken abgelegt werden. Eine Möglichkeit, dieses Konzept umzusetzen, sind Edge Side Includes (ESI). ESI bieten eine recht einfache Sprache an, um HTML aus verschiedenen Quellen zu kombinieren. Damit können Caches statischen Content – beispielsweise das Skelett einer Seite – um dynamischen Content ergänzen. So können Caches bei der Auslieferung von Seiten helfen, selbst wenn sie dynamischen Content enthalten. Proxies und Caches wie Varnish [31] oder Squid [32] setzen ESI um.

Eine andere Alternative sind Server Side Includes (SSI). Sie sind ESIs sehr ähnlich, aber sind nicht in Caches, sondern in Webservern umgesetzt. Mit SSIs können Webserver in HTML-Seiten HTML-Schnipsel von anderen Servern integrieren. Die Microservices könnten jeweils Bestandteile der Seite liefern, die dann auf dem Server zusammengestellt werden. Apache httpd unterstützt beispielsweise Server Side Includes mit mod_include [28]. nginx nutzt dazu das ngx_http_ssi_module [33].

Abb. 9–5 Integration mit einem Frontend-Server



Portale sind auch angetreten, um Informationen aus verschiedenen Quellen auf einer Webseite zu konsolidieren. Die meisten Produkte nutzen Java Portlets entsprechend dem Java-Standard JSR 168 (Portlet 1.0) oder JSR 286 (Portlet 2.0). Portlets können unabhängig voneinander in Produktion gebracht werden und lösen damit eine zentrale Herausforderung im Microservice-Umfeld. In der Praxis führen diese Technologien meistens zu komplexen Lösungen. Portlets verhalten sich technisch ganz anders als normale Java-Webanwendungen, sodass die Nutzung vieler Technologien aus dem Java-Umfeld entweder erschwert oder unmöglich ist. Portlets erlauben die Komposition einer Webseite durch den Nutzer aus vorher definierten Portlets. So kann der Nutzer sich zum Beispiel seine wichtigsten Informationsquellen auf einer Webseite zusammenstellen. Das ist aber in den hier betrachteten Fällen gar nicht notwendig. Die zusätzlichen Features führen zu zusätzlicher Komplexität. Auf Portlets basierende Portal-Server sind keine besonders gute Lösung für Weboberflächen von Microservices. Außerdem schränken sie die Webtechnologien auf den Java-Bereich ein.

Mobile Clients und Rich Clients

Weboberflächen benötigen keine Installation von Software auf dem Client. Der Webbrowsert ist der universelle Client für alle Webanwendungen. Das Deployment der Weboberfläche kann mit dem Deployment des Microservice auf der Server-Seite sehr einfach koordiniert werden. Der Microservice hat einen UI-Anteil und kann per HTTP den Code der Weboberfläche ausliefern. Dadurch kann das Deployment von Client und Server recht einfach gemeinsam erfolgen.

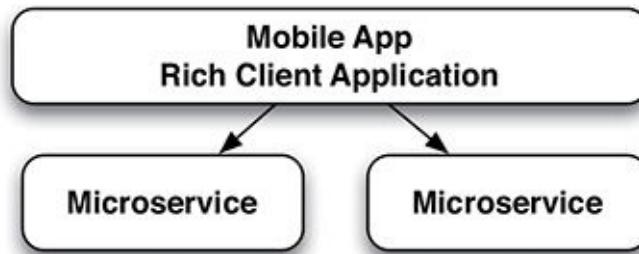
Bei mobilen Apps, Rich Clients oder Desktop-Anwendungen ist die Situation anders: Es muss eine Software auf dem Client installiert werden. Diese Client-Anwendung ist ein Deployment-Monolith, der eine Schnittstelle für alle Microservices anbieten muss. Wenn die Client-Anwendung Funktionalitäten verschiedener Microservices umfassen soll, müsste sie eigentlich modularisiert werden und die einzelnen Module müssten so wie die zugehörigen Microservices unabhängig voneinander in Produktion gebracht werden können. Das ist aber nicht möglich, denn die Client-Anwendung ist ein Deployment-Monolith. Eine SPA kann auch leicht zu einem Deployment-Monolithen werden. Manchmal hat eine SPA gerade die Aufgabe, die Entwicklung von Client und Server zu trennen. In einem Microservices-Kontext ist eine solche Nutzung von SPAs nicht

wünschenswert.

Wenn in einem Microservice ein neues Feature umgesetzt wird, das auch Änderungen in der Client-Anwendung benötigt, kann diese Änderung nicht durch eine neue Version des Microservice alleine ausgerollt werden. Es muss auch eine neue Version der Client-Anwendung ausgeliefert werden. Für jede Änderung eines kleinen Features die Client-Anwendung neu auszuliefern, ist unrealistisch. Wenn die Client-Anwendung beispielsweise in dem App Store eines mobilen Betriebssystems verfügbar sein soll, ist ein umfangreiches Review jeder Version notwendig. Wenn mehrere Änderungen zusammen ausgeliefert werden sollen, muss die Änderung koordiniert werden. Und die neue Version der Client-Anwendung muss mit den Microservices koordiniert werden, sodass die neuen Versionen der Microservices rechtzeitig bereitstehen. So ergeben sich Deployment-Abhängigkeiten zwischen den Microservices, die es eigentlich zu vermeiden gilt.

Auf organisatorischer Ebene gibt es oft ein Team, um die Client-Anwendung zu entwickeln. So wird die Aufteilung in ein eigenes Modul auch auf organisatorischer Ebene umgesetzt. Es ist gerade bei der Unterstützung verschiedener Plattformen unrealistisch, dass für jede Plattform in jedem Microservice-Team jeweils ein Entwickler sitzt. Die Entwickler werden je ein Team für jede Plattform bilden. Dieses Team muss mit allen Microservices-Teams kommunizieren, die Microservices für die mobile Application anbieten. Das kann zu sehr viel Kommunikation führen. Microservices sind eigentlich angetreten, solche Kommunikation zu vermeiden. So ergibt der Deployment-Monolith bei Client-Anwendungen auf organisatorischer Ebene eine Herausforderung.

Abb. 9–6 Mobile App/Rich Client: ein Deployment-Monolith, der mehrere Microservices integriert



Eine Lösung kann sein, neue Features zunächst für das Web zu entwickeln. Jeder Microservice kann direkt notwendige Funktionalitäten ins Web bringen. Bei einem Release der Client-Anwendung sind die Features dann auch dort verfügbar. Dann muss jeder Microservice allerdings einen bestimmten Stand von Features für die Webanwendung unterstützen und gegebenenfalls einen anderen Stand für die Client-Anwendung. Dafür kann dieser Ansatz die Webanwendung und die mobile Anwendung einheitlich halten. Es unterstützt einen Ansatz, bei dem die fachlichen Teams in den Microservices Features sowohl mobilen Nutzern und Webanwendern zur Verfügung stellen wollen. Mobile Anwendungen und Webanwendungen sind nur zwei Kanäle, um dieselben Funktionalitäten anzubieten.

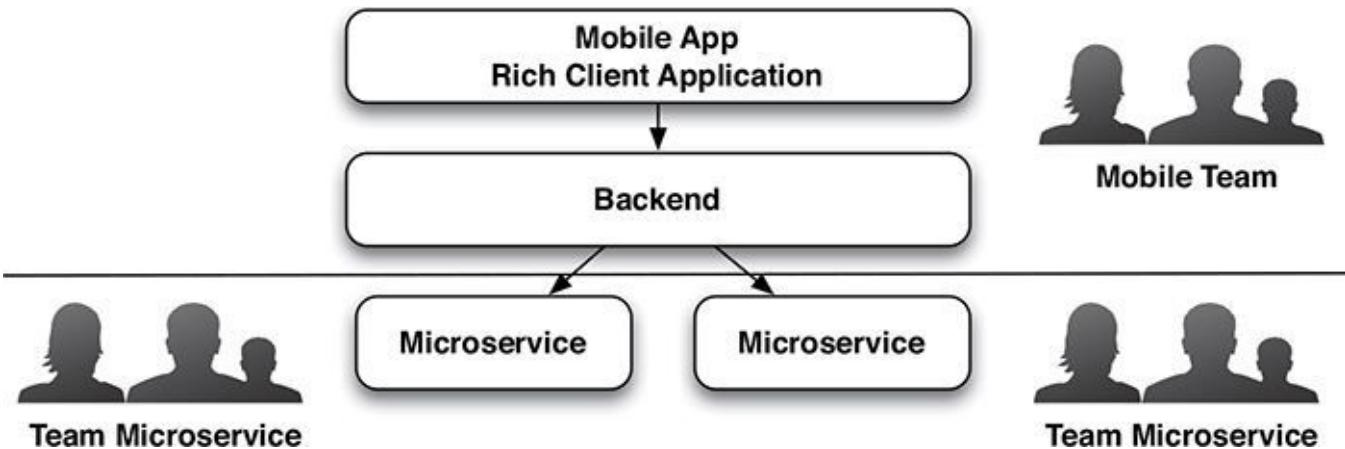
Die Anforderungen können aber auch ganz anders sein, zum Beispiel: Die mobile Anwendung ist eine

Backend für jedes Frontend

weitgehend eigenständige Anwendung, die möglichst unabhängig von den Microservices und der Weboberfläche weiterentwickelt werden soll. Oft unterscheiden sich die mobilen Anwendungsfälle so sehr von den Anwendungsfällen der Webanwendung, dass eine getrennte Entwicklung wegen der Unterschiede in den Features zwingend ist.

In solchen Fällen kann der Ansatz aus [Abbildung 9–7](#) sinnvoll sein: Das Team für die Mobile App bzw. den Rich Client hat einige Entwickler, die ein spezielles Backend implementieren. Dadurch kann die Mobile App auch im Backend unabhängig weiterentwickelt werden, weil zumindest ein Teil der Anforderungen an die Microservices durch Entwickler im eigenen Team umgesetzt werden kann. Dabei sollte nicht Fachlichkeit in dem Microservice für die Mobile App umgesetzt werden, die eigentlich in einen Backend-Microservice gehört. Aber das Backend für eine mobile Anwendung unterscheidet sich von anderen APIs. Mobile Clients haben wenig Bandbreite und eine hohe Latenz. Daher sind APIs für mobile Geräte darauf optimiert, mit möglichst wenigen Aufrufen auszukommen und nur die richtigen Daten zu übertragen. Das gilt für Rich Clients auch, aber nicht ganz im selben Maße. Die Adaption einer API auf die speziellen Anforderungen von mobilen Anwendungen kann in einem Microservice umgesetzt werden, den das Frontend-Team implementiert.

Abb. 9–7 Mobile App bzw. Rich Client mit eigenem Backend



In einer Mobile App sollte eine Benutzerinteraktion möglich schnell zu einer Reaktion der App führen. Wenn als Reaktion auf eine BenutzerInteraktion ein Aufruf eines Microservice nötig ist, kann das schon diesem Ziel widersprechen. Sind es mehrere Aufrufe, geht die Latenzzeit weiter in die Höhe. Daher sollte die API für eine Mobile App darauf optimiert sein, die notwendigen Daten mit möglichst wenigen Aufrufen zu liefern. Auch diese Optimierungen kann ein Backend für die Mobile App umsetzen.

Die Optimierungen kann das Team implementieren, das für die Mobile App zuständig ist. So können die Microservices allgemeingültige Schnittstellen anbieten, während die Teams für die Mobile Apps ihre speziellen APIs selber zusammenstellen können. Deswegen sind die Mobile-App-Teams nicht mehr so abhängig von den Teams, die für die Implementierung der Microservices zuständig sind.

Webanwendungen zu modularisieren ist einfacher als die Modularisierung von Mobile Apps, insbesondere wenn die Webanwendungen auf HTML setzen und nicht auf SPAs. Bei Mobile Apps oder Rich Client Apps ist es viel schwieriger, weil sie eine eigene Deployment-Einheit bilden und nicht so leicht aufgeteilt werden können.

Die Architektur aus [Abbildung 9–7](#) ermöglicht die Wiederverwendung von Microservices von verschiedenen Clients aus. Gleichzeitig ist es ein Einstieg in eine Schichten-Architektur. Die UI-Schicht ist von den Microservices getrennt und ein anderes Team setzt sie um. Anforderungen müssen dann in mehreren Teams umgesetzt werden. Genau das sollten Microservices vermeiden. Außerdem ergibt die Architektur die Gefahr, dass in den Services für die Client-Anwendungen Logik implementiert wird, die eigentlich in die Microservices gehört. Also hat diese Lösung nicht nur Vorteile.

Selber ausprobieren und experimentieren

Der Abschnitt hat als Alternativen für Webanwendungen ein SPA pro Microservice, ein SPA mit Modulen pro Microservice, eine HTML-Anwendung pro Microservice und einen Frontend-Server mit HTML-Snippets gezeigt. Welchen der Ansätze würdest du wählen? Warum?

Wie würdest du mit mobilen Apps umgehen? Eine Option wäre ein Team mit einem eigenen Back-end-Entwickler – oder lieber ein Team ohne Back-end-Entwickler?

9.2 REST

Microservices müssen auch untereinander Code aufrufen können, um so gemeinsam Logik umzusetzen. Das können verschiedene Technologien unterstützen.

REST (Representational State Transfer) [3] ist eine Möglichkeit, Kommunikation zwischen Microservices zu ermöglichen. REST ist der Begriff für die grundlegenden Ansätze des WWW:

- Es gibt eine Vielzahl von Ressourcen, die über URIs identifiziert werden können. URI steht für Uniform Resource Identifier und identifiziert eine Ressource global eindeutig. URLs sind praktisch dasselbe wie URIs.
- Die Ressourcen können über eine feste Menge von Methoden manipuliert werden. Bei HTTP sind das beispielsweise GET für die Abfrage der Ressource, PUT für das Ablegen einer Ressource und DELETE für das Löschen. Die Semantik der Methoden ist fest definiert.
- Für Ressourcen kann es unterschiedliche Repräsentationen geben – beispielsweise als PDF oder HTML. HTTP unterstützt die sogenannte Content Negotiation durch den Accept-Header. So kann der Client festlegen, welche Datenrepräsentationen er verarbeiten kann. Die Content Negotiation ermöglicht es beispielsweise, Ressourcen für Menschen lesbar anzuzeigen und unter derselben URL auch maschinenlesbar auszugeben. Dazu kann der Client über einen Accept-Header mitteilen, dass er für Menschen nur HTML akzeptiert, während der andere nur JSON akzeptiert.
- Beziehungen zwischen Ressourcen können durch Links ausgedrückt werden. Links können auf andere Microservices verweisen, sodass eine Integration der Logik verschiedener Microservices möglich wird.
- Der Server in einem REST-System sollte zustandslos sein. Genau deswegen

implementiert HTTP ein zustandsloses Protokoll.

Das begrenzte Vokabular ist genau das Gegenteil dessen, was objektorientierte Systeme nutzen. Objektorientierung stellt ein spezielles Vokabular mit speziellen Methoden für jede Klasse in den Mittelpunkt. Auch das REST-Vokabular kann komplexe Logik ausführen. Wenn Validierungen der Daten notwendig sind, kann das beim POST oder PUT von neuen Daten überprüft werden. Sollen komplexe Prozesse abgebildet werden, so kann ein POST den Prozess zunächst starten und dann kann regelmäßig der Zustand aktualisiert werden. Der jeweils aktuelle Stand des Prozesses kann vom Client unter der bekannten URL per GET abgeholt werden. Ebenso kann durch POST oder PUT der nächste Zustand eingenommen werden.

Eine RESTful-HTTP-Schnittstelle kann sehr einfach mit einem Cache ergänzt werden: Weil RESTful HTTP dasselbe HTTP-Protokoll wie das Web nutzt, reicht ein einfacher Web-Cache aus. Ebenso können die üblichen HTTP-Load-Balancer auch für RESTful HTTP genutzt werden. Wie mächtig diese Konzepte sind, wird schon alleine durch die Größe des Webs klar. Diese Größe ist nur wegen der Eigenschaften von HTTP möglich. Ebenso hat HTTP beispielsweise einfache und nützliche Mechanismen für Security – nicht nur Verschlüsselung über HTTPS, sondern auch Authentifizierung mit HTTP-Headern.

HATEOAS (Hypermedia as the Engine of Application State) ist ein weiterer wichtiger Bestandteil von REST. Die Beziehungen zwischen den Ressourcen werden durch Links modelliert. So muss ein Client nur einen Einstiegspunkt kennen. Von dort kann er beliebig weiternavigieren und so schrittweise alle Daten finden. Im WWW kann man beispielsweise von Google aus praktisch das gesamte Web durch Links erreichen.

REST beschreibt die Architektur des World Wide Web und damit des größten integrierten Computer-Systems. Aber REST könnte auch mit anderen Protokollen umgesetzt werden. Es ist eine Architektur, die mit unterschiedlichen Technologien umgesetzt werden kann. Die Umsetzung von REST mit HTTP heißt RESTful HTTP. Wenn RESTful-HTTP-Services statt HTML Daten als JSON oder XML austauschen, ermöglicht dieser Ansatz den Austausch von Daten und nicht nur den Zugriff auf Webseiten.

Auch Microservices profitieren von HATEOAS. HATEOAS hat keine zentrale Koordinierung, sondern nur Links. Das passt gut dazu, dass Microservices möglichst wenig zentrale Koordination verursachen sollen. Clients kennen bei REST nur Einstiegspunkte, von denen aus sie das Gesamtsystem entdecken können. In einer REST-Architektur können daher Services transparent für den Client verschoben werden. Der Client bekommt einfach neue Links. Eine zentrale Koordinierung ist dazu ebenfalls nicht notwendig. Der REST-Service muss nur andere Links zurückgeben. Im Idealfall muss der Client nur die Grundlagen von HATEOAS verstehen und kann dann durch die Links zu beliebigen Daten in den Microservice-Systemen navigieren. Die Microservice-Systeme wiederum können ihre Links ändern und so die Verteilung der Funktionalitäten auf die Microservices verändern. Auch umfangreiche Änderungen an der Architektur können transparent gehalten werden.

HATEOAS ist ein Konzept. HAL [20] ist eine Möglichkeit zur Implementierung. Es ist ein Standard, wie die Links auf andere Dokumente in einem JSON-Dokument enthalten sein sollen. Damit ist HATEOAS gerade in JSON/RESTful-HTTP-Services besonders einfach umsetzbar. Die Links sind vom eigentlichen Dokument getrennt. Damit sind Links auf Details oder auf abhängige Datensätze umsetzbar.

XML hat eine lange Historie als Datenformat. Es ist zusammen mit RESTful HTTP einfach nutzbar. Für XML gibt es verschiedene Typsysteme, die festlegen können, wann ein XML-Dokument gültig ist. Das ist sehr nützlich für die Definition einer Schnittstelle. Zu den Sprachen für die Definition der gültigen Daten zählt beispielsweise XML Schema (XSD) [17] oder RelaxNG [18]. Einige Frameworks erlauben die Generierung von Code, um XML-Daten zu verwalten, die einem solchen Schema entsprechen. XML-Dokumente können mit XLink [19] Links zu anderen Dokumenten enthalten. Das ermöglicht die Umsetzung von HATEOAS.

XML ist angetreten, um Daten und Dokumente zu übertragen. Die Informationen anzugeben, ist die Aufgabe von anderer Software. Mittlerweile hat HTML einen ähnlichen Ansatz wie XML: HTML definiert nur die Strukturen. Die Darstellung geschieht durch CSS. Für die Kommunikation zwischen Prozessen können HTML-Dokumente ausreichend sein, denn in modernen Webanwendungen enthalten die Dokumente wie XML nur noch Daten. In einer Microservices-Welt hat dieses Vorgehen den Vorteil, dass die Kommunikation zum Benutzer und zwischen den Microservices dasselbe Format verwendet. Das reduziert den Aufwand. So wird es noch einfacher, Microservices umzusetzen, die eine UI und eine Kommunikationsmöglichkeit für andere Microservices enthalten.

JSON (JavaScript Object Notation) ist eine Darstellung von Daten, die vor allem für JavaScript optimiert ist. Wie JavaScript auch, sind die Daten dynamisch typisiert. Mittlerweile gibt es aber eigentlich für alle Programmiersprachen passende JSON-Bibliotheken. Es gibt außerdem Typsysteme wie JSON Schema [16], die für JSON eine entsprechende Validierung ergänzen. Damit steht JSON Datenformaten wie XML in nichts mehr nach.

Statt textbasierten Datenrepräsentationen können auch binäre Protokolle wie Protocol Buffer [4] genutzt werden. Diese Technologie ist von Google entworfen worden, um Daten effizienter zu repräsentieren und eine höhere Performance zu erreichen. Es gibt Implementierungen für viele verschiedene Programmiersprachen, sodass Protocol Buffer ähnlich universell verwendet werden können wie JSON oder XML.

RESTful HTTP ist synchron: Typischerweise schickt ein Service einen Request los und wartet auf die Antwort, die dann ausgewertet wird, um im Programmablauf weiterzumachen. Das kann bei hohen Latenzzeiten im Netzwerk zu Schwierigkeiten führen. Es kann die Bearbeitung einer Anfrage verlängern, da auf die Antworten anderer Services gewartet werden muss. Außerdem muss nach einer bestimmten Wartezeit die Anfrage abgebrochen werden, denn nach einer bestimmten Wartezeit wird die Anfrage wahrscheinlich nie beantwortet.

Gründe können sein, dass ein Server gerade nicht zur Verfügung steht oder das Netzwerk ein Problem hat. Ein richtig behandeltes Timeout erhöht die Stabilität des Systems ([Abschnitt 10.5](#)).

Der Ausfall darf nicht dazu führen, dass weitere Services ausfallen. Daher muss durch den Timeout gewährleistet werden, dass das eigene System noch antwortet und sich der Ausfall nicht weiter fortpflanzt.

9.3 SOAP und RPC

Es ist möglich, eine Microservices-Architektur auf SOAP aufzubauen. SOAP nutzt wie REST auch HTTP, aber verwendet nur POST-Nachrichten, um Daten an einen Server zu übertragen. Letztendlich aktiviert ein SOAP-Aufruf ein bestimmtes Objekt auf dem Server und ruft auf ihm eine Methode auf. Damit ist SOAP ein RPC-Mechanismus (Remote Procedure Call), der Methoden in einem anderen Prozess aufruft.

SOAP fehlen Mechanismen wie HATEOAS, mit denen die Beziehungen zwischen den Microservices flexibel gehandhabt werden können. Die Schnittstellen müssen vollständig vom Server definiert und auf dem Client bekannt sein.

SOAP kann Nachrichten mit verschiedenen Transportmechanismen übertragen. So ist es möglich, dieselbe Nachricht mit HTTP zu empfangen und dann per JMS als Message oder SMTP/POP als E-Mail weiterzuschicken. Die auf SOAP aufsetzenden Technologien unterstützen ein Weiterleiten der Aufrufe ebenfalls. So erlaubt der Sicherheits-standard WS-Security, Teile einer Nachricht zu verschlüsseln oder zu unterschreiben. Die Teile können dann an verschiedene Services weitergeschickt werden, ohne dass sie entschlüsselt werden müssen. Der Sender kann eine Nachricht verschicken, in der einige Teile verschlüsselt sind. Die Nachricht kann dann über verschiedene Stationen verarbeitet werden. Jede Station kann als ein Teil der Nachricht verarbeitet oder an andere Empfänger verschickt werden. Schließlich kommen die verschlüsselten Teile bei den endgültigen Empfängern an – und erst dort müssen sie entschlüsselt und verarbeitet werden.

SOAP hat viele Erweiterungen für spezielle Einsatzkontexte. Die verschiedenen Erweiterungen aus dem WS-* Umfeld umfassen beispielsweise Transaktionen und die Koordination von Webservices. So kann ein komplexer Protokoll-Stack entstehen. Unter der Komplexität kann die Interoperabilität zwischen verschiedenen Services und Lösungen leiden. Einige Technologien sind für Microservices auch nicht besonders sinnvoll. So ist eine Koordination verschiedener Microservices problematisch, weil so eine Koordinationsschicht entsteht und Änderungen an einem Geschäftsprozess vermutlich die Koordination der Microservices und die Microservices selbst betreffen. Wenn die Koordinationsschicht alle Microservices umfasst, entsteht so ein Monolith, der auch bei jeder Änderung mit geändert werden muss. Das widerspricht dem Microservices-Gedanken von unabhängigem Deployment. WS-* ist eher aus dem SOA-Gedanken geboren.

Eine weitere Kommunikationsmöglichkeit ist Apache Thrift [12]. Es verwendet wie Protocol Buffer eine sehr

Flexibler Transport

Thrift

effiziente binäre Kodierung der Daten. Darüber hinaus kann Thrift Aufrufe von einem Prozess mit einer Programmiersprache über das Netzwerk an andere Prozesse weiterleiten. Die Schnittstelle wird in einer eigenen Schnittstellendefinition hinterlegt. Auf Basis dieser Definition können unterschiedliche Client- und Server-Technologien miteinander kommunizieren.

9.4 Messaging

Eine weitere Möglichkeit für die Kommunikation zwischen Microservices sind Messages und Messaging-Systeme. Wie der Name schon sagt, basieren diese Systeme auf dem Verschicken von Nachrichten. Die Nachrichten können zu einer Antwort führen, die wieder als Nachricht verschickt wird. Nachrichten können an einen oder mehrere Empfänger gehen.

Gerade bei verteilten Systemen können Messaging-Lösungen ihre Vorteile ausspielen:

- Nachrichten können auch bei Ausfall des Netzwerks übertragen werden. Das Messaging-System speichert sie zwischen und stellt sie zu, wenn das Netzwerk wieder zur Verfügung steht.
- Die Garantien können weiter erhöht werden: Das Messaging-System kann nicht nur die korrekte Übertragung der Nachrichten garantieren, sondern sogar die Verarbeitung. Wenn es bei der Verarbeitung der Nachricht ein Problem gab, kann die Nachricht erneut übertragen werden. Eine erfolgreiche Bearbeitung ist möglich, wenn der Fehler nach einiger Zeit verschwindet. Sonst wird noch einige Male versucht, die Nachricht zu bearbeiten, bis schließlich ein Fehler zurückgemeldet wird.
- In einer Messaging-Architektur werden Antworten asynchron übertragen und bearbeitet. Solche Architekturen sind gut auf hohe Latenzzeiten abgestimmt, wie sie im Netzwerk vorkommen. Das Warten auf eine Antwort ist in einer solchen Architektur der übliche Fall. Daher geht das Programmiermodell immer von einer Latenz aus.
- Ein Aufruf eines anderen Service blockiert die weitere Verarbeitung nicht. Selbst wenn die Antwort noch nicht eingetroffen ist, kann der Service weiterarbeiten und beispielsweise weitere Services aufrufen.
- Der Sender kennt den Empfänger der Nachricht nicht. Der Sender schickt die Nachricht an eine Queue oder ein Topic. Dort registriert sich der Empfänger. So sind Sender und Empfänger entkoppelt. Es kann sogar mehrere Empfänger geben, ohne dass dies dem Sender bekannt ist. Außerdem können die Nachrichten auf dem Weg modifiziert werden. Es können beispielsweise Daten ergänzt oder entfernt werden. Und es können Nachrichten auch an ganz andere Empfänger weitergeleitet werden.

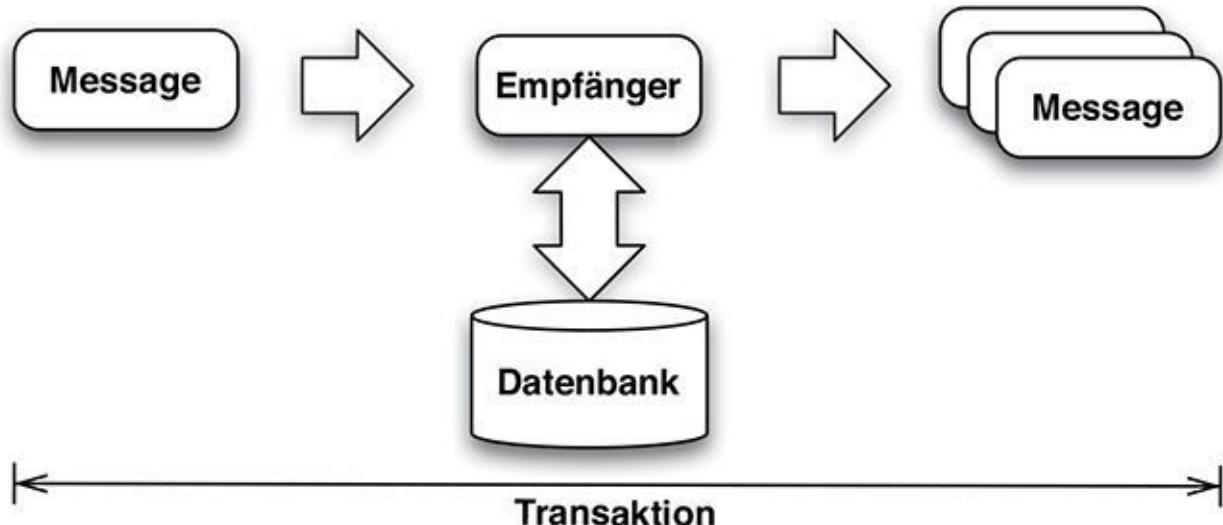
Messaging ist auch eine gute Basis für bestimmte Architekturen von Microservice-Systemen wie Event Sourcing (siehe [Abschnitt 10.3](#)) oder Even-driven Architecture ([Abschnitt 8.6](#)).

Messaging bietet eine Lösung für transaktionale Systeme mit Microservices. In einem Microservice-

Messages und Transaktionen

System sind die Garantien von Transaktionen schwierig zu gewährleisten, wenn die Microservices sich gegenseitig aufrufen. Dann müssten alle Microservices an einer Transaktion teilnehmen. Sie dürften erst dann Änderungen schreiben, wenn alle Microservices in der Transaktion fehlerfrei durchlaufen worden sind. Die Änderungen müssten dafür sehr lange zurückgehalten werden. Das ist für die Performance schlecht, weil keine andere Transaktion währenddessen die Daten ändern kann. Außerdem kann in einem Netzwerk immer ein Teilnehmer ausfallen. Dann bleibt die Transaktion lange offen oder wird vielleicht nie geschlossen. In dem Fall sind Änderungen an Daten für eine längere Zeit nicht möglich. Solche Probleme tauchen beispielsweise auf, wenn das aufrufende System abgestürzt ist.

Abb. 9–8 Transaktionen und Messaging



In einem Messaging-System kann mit Transaktionen anders umgegangen werden: Das Schicken und Empfangen von Nachrichten ist Teil einer Transaktion – genauso wie beispielsweise das Schreiben oder Lesen von der Datenbank (Abb. 9–8). Wenn beim Verarbeiten der Nachricht ein Fehler auftritt, werden alle ausgehenden Nachrichten gestrichen und die Datenbankänderungen werden rückgängig gemacht. Im Erfolgsfall finden alle diese Aktionen statt. Die Empfänger der Nachrichten können ebenfalls transaktional abgesichert sein. Dann unterliegt die Bearbeitung der ausgehenden Nachrichten denselben transaktionalen Garantien.

Entscheidend ist, dass Schicken und Empfangen von Nachrichten und die Transaktion auf der Datenbank in einer Transaktion zusammengefasst werden. Die Koordination besorgt die Infrastruktur. Es muss kein eigener Code geschrieben werden. Zur Koordination von Messaging und Datenbank kann Two Phase Commit (2PC) verwendet werden. Dieses Protokoll ist die übliche Lösung, um transaktionale Systeme wie Datenbanken und Messaging-Systeme miteinander zu koordinieren. Eine Alternative sind Produkte wie Oracle AQ oder ActiveMQ. Sie legen die Messages in einer Datenbank ab. Dann kann die Koordination zwischen Datenbank und Messaging einfach dadurch erfolgen, dass in derselben Datenbanktransaktion sowohl die Messages als auch die Änderungen an den Daten geschrieben werden. Messaging und Datenbank sind dann letztendlich dieselben Systeme.

Mit Messaging können Transaktionen umgesetzt werden, ohne dass es eine globale Koordination gibt. Jeder Microservice ist transaktional. Das transaktionale Verschicken

der Nachrichten wird durch die Messaging-Technologie gewährleistet. Wenn allerdings eine Nachricht beispielsweise wegen ungültiger Werte nicht verarbeitet werden kann, gibt es keine Möglichkeit, die bereits verarbeiteten Nachrichten zurückzurollen. Die Transaktionalität ist nicht unter allen Umständen gegeben.

Für die Umsetzung von Messaging muss eine **Messaging-Technologie** Technologie genutzt werden:

- AMQP (Advanced Message Queuing Model) [5] ist ein Standard. Er definiert ein Protokoll, mit dem Messaging-Lösungen im Netzwerk miteinander und mit Clients kommunizieren können. Eine Implementierung des Standards ist RabbitMQ [6], das in Erlang geschrieben ist und unter Mozilla-Lizenz steht. Eine andere Implementierung ist beispielsweise Apache Qpid.
- Apache Kafka [21] fokussiert auf hohen Durchsatz, Replikation und Ausfallsicherheit. Damit ist es für verteilte Systeme wie Microservices gut geeignet, weil die Ausfallsicherheit in diesem Einsatzkontext sehr hilfreich ist.
- 0MQ (auch ZeroMQ oder ZMQ genannt) [22] kommt ohne einen eigenen Server aus und ist daher sehr leichtgewichtig. Es hat einige Primitiven, die zu einem komplexeren System zusammengestellt werden können. 0MQ steht unter der LGPL-Lizenz und ist in C++ geschrieben.
- JMS (Java Messaging Service) [7] definiert eine API, mit der eine Java-Anwendung Nachrichten empfangen und schicken kann. Wie die Lösung die Messages auf dem Draht überträgt, wird im Gegensatz zu AMQP durch die Spezifikation nicht definiert. Da es ein Standard ist, setzen Java-EE-Server diese API um. Bekannte Implementierungen sind ActiveMQ [29] und HornetQ [30].
- Es ist auch möglich, ATOM Feeds [23][24] für Messaging zu nutzen. Diese Technologie wird sonst verwendet, um Blog-Inhalte zu übertragen. Clients können recht einfach neue Einträge eines Blogs abfragen. Genau kann ein Client ATOM nutzen, um neue Messages abzufragen. ATOM basiert auf HTTP und passt daher gut in eine REST-Umgebung. Allerdings hat ATOM eigentlich nur Funktionalitäten, um neue Informationen zu liefern. Komplexere Techniken wie Transaktionen unterstützt es nicht.

Für viele Messaging-Lösungen sind ein Messaging-Server und damit eine zusätzliche Infrastruktur notwendig. Die muss so betrieben werden, dass sie nicht ausfällt, weil dann die Kommunikation im gesamten Microservice-System zum Erliegen kommt. Messaging-Lösungen sind aber meistens darauf ausgelegt, hohe Verfügbarkeit beispielsweise durch Clustering zu erreichen.

Für viele Entwickler ist Messaging eher ungewohnt, da es asynchrone Kommunikation erfordert. Dadurch scheint es eher komplex zu sein. Meistens ist der Aufruf einer Methode in einem anderen Prozess einfacher zu verstehen. Mit Ansätzen wie Reactive (siehe [Abschnitt 10.6](#)) hält asynchrone Entwicklung aber Einzug in die Kommunikation in den Microservices selbst. Auch das AJAX-Modell aus der JavaScript-Entwicklung ist dem asynchronen Behandeln von Nachrichten nicht unähnlich. Mehr und mehr Entwickler sind also mit dem asynchronen Modell vertraut.

Selber ausprobieren und experimentieren

REST, SOAP/RPC und Messaging haben jeder Vor- und Nachteile. Sammle die Vor- und Nachteile und entscheide dich für eine Alternative.

In einem Microservice-System kann es mehrere Kommunikationsarten geben – aber es sollte eine vorherrschende Kommunikationsform geben. Welche würdest du wählen? Welche anderen wären außerdem noch erlaubt? In welchen Situationen?

9.5 Datenreplikation

Auf der Datenbankebene könnten Microservices sich eine Datenbank teilen und so gemeinsam auf Datenbestände zugreifen. Diese Art Integration wird schon lange praktiziert: Es ist nicht ungewöhnlich, dass eine Datenbank von mehreren Anwendungen verwendet wird. Oft überdauern Datenbanken sogar die Anwendungen, sodass nicht mehr die Anwendung mit ihren Ansprüchen im Mittelpunkt steht, sondern die Datenbank. Obwohl die Integration über eine gemeinsame Datenbank verbreitet ist, hat sie entscheidende Nachteile:

- Die Repräsentation der Daten kann nicht ohne Weiteres geändert werden. Schließlich greifen mehrere Anwendungen auf die Daten zu. Eine Änderung kann eine der Anwendungen brechen. Änderungen müssen mit allen Anwendungen koordiniert werden.
- Dadurch ist es nicht mehr möglich, die Anwendungen schnell zu ändern, wenn dazu auch Datenbankänderungen notwendig sind. Schnelle Änderbarkeit ist aber genau der Bereich, in dem Microservices Vorteile haben sollen.
- Schließlich ist es auch kaum möglich, das Schema aufzuräumen – also Spalten zu entfernen, die nicht mehr notwendig sind. Langfristig wird die Datenbank immer komplexer und schlechter wartbar werden.

Letztendlich ist die gemeinsame Nutzung einer Datenbank ein Verstoß gegen einen wichtigen Architektur-Grundsatz. Komponenten sollten ihre Datenrepräsentierung ändern können, ohne dass andere Komponenten davon betroffen sind. Das Datenbankschema ist ein Beispiel für eine interne Datenrepräsentation. Wenn mehrere Komponenten sich die Datenbank teilen, ist eine Änderung der Datenrepräsentation nicht mehr möglich. Microservices sollten daher eine streng getrennte Datenhaltung haben und sich nicht ein Datenbankschema teilen.

Allerdings kann eine Datenbankinstanz für mehrere Microservices genutzt werden, wenn die Datenbestände der Microservices vollständig getrennt sind. Beispielsweise kann jeder Microservice sein eigenes Schema in der gemeinsamen Datenbank nutzen. Dann darf es aber keine Beziehungen zwischen den Schemata geben.

Die Replikation von Daten ist für die Integration von Replikation Microservices eine mögliche Alternative. Allerdings darf die Replikation auf keinen Fall durch die Hintertür eine Abhängigkeit der Datenbankschemata einführen. Wenn die Daten einfach repliziert werden und dasselbe

Schema genutzt wird, tritt dasselbe Problem auf, wie wenn die Datenbank gemeinsam genutzt wird. Eine Schema-Änderung schlägt auf den anderen Microservice durch, sodass es eine enge Kopplung der Microservices gibt. Das gilt es zu vermeiden.

Die Daten sollten alleine schon wegen der Unabhängigkeit der Microservices in ein anderes Schema überführt werden. Auch aus fachlichen Gründen ist eine solche Transformation meistens wünschenswert.

Ein typisches Beispiel für die Nutzung von Replikation in der klassischen IT sind Data Warehouses. Sie replizieren Daten, aber speichern sie anders. Das liegt daran, dass die Zugriffe auf die Daten im Data Warehouse von ganz anderen Anforderungen getrieben sind: Es sollen sehr viele Daten analysiert werden. Die Daten werden für Lesezugriffe optimiert und oft auch zusammengefasst, weil für die Statistiken nicht jeder einzelne Datensatz relevant ist.

Wegen BOUNDED CONTEXT sind für unterschiedliche Microservices meistens sowieso unterschiedliche Repräsentationen oder Ausschnitte der Daten relevant. Aus diesem Grund wird es bei einer Replikation der Daten zwischen Microservices sowieso meistens notwendig sein, die Daten zu transformieren oder nur Ausschnitte aus den Daten zu replizieren.

Durch die Replikation entsteht eine redundante Speicherung der Daten. Das bedeutet, dass die Daten nicht sofort konsistent sind: Es dauert einige Zeit, bis Änderungen überallhin repliziert worden sind. *Problem: Redundanz und Konsistenz*

Sofortige Konsistenz kann aber verzichtbar sein. Bei einer Analyse-Aufgabe wie einem Data Warehouse kann eine Analyse ohne die Bestellungen der letzten Minuten ausreichend sein. Ebenso finden sich Fälle, in denen Konsistenz nicht so wichtig ist. Wenn eine Bestellung erst ein wenig später in dem Microservice für Lieferungen sichtbar ist, kann das akzeptabel sein, weil in der Zwischenzeit vielleicht sowieso niemand die Daten abruft.

Konsistenz ist eine Anforderung an das System. Hohe Konsistenzanforderungen erschweren Replikation. Wie konsistent die Daten genau sein müssen, wird oft beim Erheben der Anforderungen nicht ermittelt. Dann sind die Möglichkeiten zur Replikation der Daten dementsprechend begrenzt.

Auch bei Replikation muss es ein führendes System geben, das den aktuellen Datenbestand enthält. Alle anderen Replikate sollten die Daten von diesem System bekommen. So ist jederzeit klar, welcher Datenbestand wirklich aktuell ist. Es sollten keine Änderungen der Daten von verschiedenen Datenbeständen aus ausgelöst werden. Das führt sehr leicht zu Konflikten und zu einer sehr komplexen Implementierung. Wenn es nur eine Quelle für Änderungen gibt, sind solche Konflikte ausgeschlossen.

Einige Datenbanken bieten Replikation als Feature an. *Implementierung* Das ist aber nicht hilfreich für die Replikation von Daten zwischen Microservices, weil die Schemata der Microservices unterschiedlich sein sollen. Die Replikation muss selber implementiert werden. Dazu kann eine eigene Schnittstelle implementiert werden. Sie sollte performanten Zugriff auch auf eine große Menge von Daten erlauben. Um die nötige Performance zu erreichen, kann auch direkt in das

Zielschema geschrieben werden. Die Schnittstelle muss nicht unbedingt ein Protokoll wie REST nutzen, sondern kann schnellere, alternative Protokolle nutzen. Dazu kann es notwendig sein, einen anderen Kommunikationsmechanismus zu nutzen, als die Microservices sonst verwenden.

Die Replikation kann in einem Batch aktiviert werden. Batch
Dann kann der komplette Datenbestand oder zumindest die Änderungen aus einem längeren Zeitraum können übertragen werden. Bei dem ersten Replikationslauf kann die Datenmenge groß sein und die Replikation entsprechend lange dauern. Es kann sinnvoll sein, jedes Mal alle Daten zu übertragen. Dadurch können Fehler, die bei der letzten Replikation aufgetreten sind, korrigiert werden.

Eine einfache Umsetzung kann jedem Datensatz eine Version zuweisen. Auf Basis der Version können die Datensätze ausgewählt und repliziert werden, die sich geändert haben. Dieser Ansatz kann bei einem Abbruch der Replikation einfach wieder neu gestartet werden, weil der Prozess selber keinen Zustand hält.

Eine Alternative ist, die Replikation bei bestimmten Event Events zu starten. Beispielsweise können bei der Neuanlage eines Datensatzes die Daten gleich auch in die Replikate kopiert werden. Solche Ansätze sind mit Messaging ([Abschnitt 9.4](#)) besonders einfach umsetzbar.

Replikation von Daten ist gerade für den performanten Zugriff auf eine große Menge Daten eine gute Wahl. Viele Microservice-Systeme kommen ohne Replikation von Daten aus. Selbst die Systeme, die Datenreplikation nutzen, werden auch andere Integrationsmechanismen verwenden.

Selber ausprobieren und experimentieren

Würdest du in einem Microservice-System Datenreplikation nutzen? In welchen Bereichen? Wie würdest du sie implementieren?

9.6 Schnittstellen: intern und extern

Microservice-Systeme haben verschiedene Arten von Schnittstellen:

- Jeder Microservice kann eine oder mehrere Schnittstellen für andere Microservices haben. Eine Änderung an der Schnittstelle kann eine Koordination mit anderen Microservice-Teams notwendig machen.
- Die Schnittstellen zwischen Microservices, die von demselben Team entwickelt werden, sind ein Sonderfall. Teammitglieder können eng zusammenarbeiten, sodass diese Schnittstellen einfacher änderbar sind.
- Außerdem kann das Microservice-System Schnittstellen nach außen anbieten, mit denen das System auch außerhalb der Organisation der Entwickler genutzt werden kann. Im Extremfall kann das jeder Internet-Nutzer sein, wenn das System eine öffentliche Schnittstelle im Internet anbietet.

Diese Schnittstellen sind unterschiedlich einfach änderbar: Einen Kollegen im selben

Team um eine Änderung zu bitten, ist sehr einfach. Der Kollege ist vermutlich sogar im selben Raum.

Bei einer Änderung an einer Schnittstelle eines Microservice eines anderen Teams wird es schwieriger. Die Änderung muss sich gegen andere Änderungen und neue Feature durchsetzen. Wenn die Änderung mit anderen Teams abgesprochen werden muss, entstehen weitere Aufwände.

Schnittstellenänderungen zwischen Microservices können durch geeignete Tests abgesichert werden (Consumer-driven Contract Tests, [Abschnitt 11.7](#)). Sie überprüfen, ob die Schnittstelle noch den Erwartungen der Schnittstellennutzer entspricht.

Bei Schnittstellen nach außen ist die Abstimmung mit [Externe Schnittstellen](#) den Nutzern komplizierter. Es können sehr viele Nutzer sein. Bei einer öffentlichen Schnittstelle können die Nutzer sogar unbekannt sein. Techniken wie Consumer-driven Contract Tests sind aus diesen Gründen in solchen Szenarien schwer umsetzbar. Aber für Schnittstellen nach außen können Regeln definiert werden, die beispielsweise festlegen, wie lange eine bestimmte Version der Schnittstelle unterstützt wird. Auch ein größerer Fokus auf Abwärtskompatibilität kann bei öffentlichen Schnittstellen sinnvoll sein.

Für Schnittstellen nach außen kann es notwendig sein, mehrere Versionen der Schnittstelle zu unterstützen, um nicht alle Nutzer zu Änderungen zu zwingen. Zwischen den Microservices sollte es ein Ziel sein, mehrere Versionen nur zum Entkoppeln von Deployments zuzulassen. Wenn ein Microservice eine Schnittstelle ändert, sollte er auch noch die alte Schnittstelle unterstützen. Dann müssen die Microservices, die von der alten Schnittstelle abhängen, nicht sofort neu deployt werden. Aber das nächste Deployment sollte die neue Schnittstelle nutzen. Danach kann die alte Schnittstelle entfernt werden. So wird die Anzahl der zu unterstützenden Schnittstellen reduziert und damit die Komplexität des Systems.

Weil die Schnittstellen unterschiedlich einfach [Schnittstellen trennen](#) änderbar sind, sollten sie getrennt umgesetzt werden. Wenn eine Schnittstelle eines Microservice extern verwendet werden soll, kann sie anschließend nur noch geändert werden, wenn man sich mit den externen Nutzern abstimmt. Allerdings kann eine neue Schnittstelle für die interne Nutzung abgespalten werden. Die nach außen exponierte Schnittstelle wird dann der Startpunkt für eine getrennte interne Schnittstelle, die wieder einfacher änderbar ist.

Außerdem können mehrere Versionen derselben Schnittstelle intern gemeinsam umgesetzt werden. So können beispielsweise neue Parameter einer neuen Version bei Aufrufen an die alte Schnittstelle einfach mit Vorgabewerten belegt werden, sodass beide Schnittstellen intern dieselbe Implementierung nutzen.

Microservice-Systeme können Schnittstellen nach [Externe Schnittstelle umsetzen](#) außen auf unterschiedliche Arten anbieten. Neben einer Web-Schnittstelle für Nutzer kann es auch eine API geben, die von außen zugreifbar ist. Für die Web-Schnittstelle hat [Abschnitt 9.1](#) schon gezeigt, wie die Integration der Microservices so erfolgen kann, dass alle Microservices einen Teil der UI implementieren können.

Wenn das System nach außen eine REST-Schnittstelle anbietet, können mithilfe eines Routers die Anfragen von außen an einen der Microservices weitergeleitet werden. In der Beispielanwendung wird dafür der Router Zuul verwendet ([Abschnitt 14.9](#)). Zuul ist sehr flexibel und kann anhand sehr detaillierter Regeln Anfragen an unterschiedliche Microservices weiterleiten. Allerdings bietet HATEOAS auch die Freiheit, Ressourcen zu verschieben. Dann ist ein Routing entbehrlich. Die Microservices sind von außen durch URLs zugreifbar, aber sie können jederzeit verschoben werden. Schließlich werden die URLs durch HATEOAS dynamisch ermittelt.

Es wäre auch denkbar, einen Adapter für die externe Schnittstelle anzubieten, der die externen Aufrufe modifiziert, bevor sie zu den Microservices gelangen. Dann kann eine Änderung in der Logik aber nicht immer auf einen Microservice begrenzt werden, sondern auch den Adapter betreffen.

Um Änderungen an einer Schnittstelle deutlich zu *Semantic Versioning* machen, kann eine Versionsnummer genutzt werden.

Semantic Versioning [25] legt eine mögliche Bedeutung einer Versionsnummer fest. Die Versionsnummer ist aufgeteilt in MAJOR.MINOR.PATCH. Die Bestandteile haben folgende Bedeutung:

- Eine Änderung in MAJOR zeigt an, dass die neue Version Rückwärtskompatibilität bricht. Die Client müssen auf die neue Version angepasst werden.
- Die MINOR-Version wird angepasst, wenn die Schnittstelle neue Features anbietet. Die Änderungen sollten aber rückwärtskompatibel sein. Eine Änderung der Clients ist nur notwendig, wenn sie die neuen Features nutzen wollen.
- PATCH wird bei Bug Fixes erhöht. Solche Änderungen sollten vollständig rückwärtskompatibel sein und keine Anpassungen an den Clients erfordern.

Bei REST ist zu beachten, dass eine Kodierung der Version in der URL nicht sinnvoll ist. Die URL sollte eine Ressource repräsentieren – unabhängig davon, mit welcher API-Version sie angesprochen wird. Daher kann die Version zum Beispiel auch in einem Accept-Header des Requests definiert werden.

Eine weitere wichtige Grundlage für die Definition von Schnittstellen ist Postels Gesetz [6], das auch unter dem Namen Robustheitsprinzip bekannt ist. Es besagt, dass Komponenten in dem, was sie tun, streng sein sollen und liberal in dem, was sie von anderen akzeptieren. Mit anderen Worten sollte jede Komponente bei der Nutzung anderer Komponenten sich möglichst genau an die Vorgaben halten, aber selber Fehler bei der Nutzung der eigenen Schnittstelle wenn möglich kompensieren.

Postels Gesetz oder das Robustheitsprinzip

Wenn jede Komponente sich nach dem Robustheitsprinzip verhält, verbessert das die Interoperabilität: Hält sich jede Komponente tatsächlich genau an die Vorgaben, müsste die Interoperabilität schon gewährleistet sein. Sollte dennoch eine Abweichung bestehen, so wird die genutzte Komponente versuchen, ihn zu kompensieren und so die Interoperabilität zu »retten«. Dieses Konzept ist auch als Tolerant Reader [27] bekannt.

Konkret sollte ein aufgerufener Service die Aufrufe akzeptieren, wenn das irgend möglich ist. Ein Weg dazu ist es, aus einem Aufruf nur die Parameter auszulesen, die

wirklich notwendig sind. Auf keinen Fall sollte man einen Aufruf ablehnen, nur weil er formal die Schnittstellenspezifikation nicht einhält. Die eigenen Aufrufe sollten aber validiert werden. Mit einem solchen Ansatz ist es einfacher, die Kommunikation in verteilten Systemen wie Microservices reibungslos zu gewährleisten.

9.7 Fazit

Die Integration von Microservices kann auf verschiedenen Ebenen stattfinden.

Eine mögliche Ebene der Integration ist die Web-
Schnittstelle ([Abschnitt 9.1](#)):

- Jeder Microservice kann eine eigene Single-Page-App (SPA) mitbringen. Die SPAs können getrennt entwickelt werden. Der Übergang zwischen den Microservices ist aber ein Start einer komplett neuen SPA.
- Es kann für das gesamte System eine SPA geben. Jeder Microservice liefert ein Modul zur SPA. Dadurch sind die Übergänge zwischen den Microservices in der SPA sehr einfach. Aber die Microservices werden sehr eng integriert, sodass eine Koordination der Deployments notwendig sein kann.
- Jeder Microservice kann eine HTML-Anwendung mitbringen. Die Integration kann über Links stattfinden. Dieser Ansatz ist einfach umsetzbar und erlaubt eine Modularisierung der Webanwendung.
- JavaScript kann HTML nachladen. Das HTML kann von verschiedenen Microservices kommen, sodass jeder Microservice eine Darstellung seiner Daten beisteuern kann. So kann eine Bestellung die Darstellung einer Ware von einem anderen Microservice laden.
- Ein Skelett kann einzelne HTML-Schnipsel zusammenstellen. So kann eine E-Commerce-Startseite die letzten Bestellungen von einem Microservice und Empfehlungen von einem anderen Microservice darstellen. Dazu können ESI (Edge Side Includes) oder SSI (Server Side Includes) nützlich sein.

Die Integration bei einem Rich Client oder einer Mobile App ist schwierig, weil die Client-Anwendung ein Deployment-Monolith ist. Daher können Änderungen verschiedener Microservices eigentlich nur gemeinsame deployt werden. Die Teams können die Microservices ändern und dann eine bestimmte Menge dazu passender UI-Änderungen gemeinsam als neues Release der Client-Anwendung ausliefern. Es kann auch für jede Client-Anwendung ein Team geben, das neue Funktionalitäten der Microservices in die Client-Anwendung übernimmt. Organisatorisch kann es sogar im Team der Client-Anwendung Entwickler geben, die einen eigenen Service entwickeln. Dieser Service kann beispielsweise die Schnittstelle so implementieren, dass die Client-Anwendung sie performant nutzen kann.

Für die Kommunikation der Logik-Schicht ist REST eine Option ([Abschnitt 9.2](#)). REST nutzt die Mechanismen des WWW, um die Kommunikation zwischen Services zu ermöglichen. HATEOAS (Hypermedia as the Engine of Application State) bedeutet, dass die

Beziehungen zwischen Systemen durch Links abgebildet werden. Der Client kennt nur eine Einstiegs-URL. Alle anderen URLs können geändert werden, weil die Clients sie nicht direkt ansprechen, sondern durch Links von der Einstiegs-URL aus finden. HAL definiert, wie Links ausgedrückt werden können, und hilft bei der Implementierung von REST. Weitere mögliche Datenformate für REST sind XML, JSON, HTML oder Protocol Buffer.

Klassische Protokolle wie SOAP oder RPC ([Abschnitt 9.3](#)) können auch zur Kommunikation der Microservices verwendet werden. SOAP bietet Möglichkeiten, eine Nachricht an andere Microservices weiterzuleiten. Thrift hat ein effizienteres binäres Protokoll und kann ebenfalls Aufrufe zwischen Prozessen weiterleiten.

Messaging ([Abschnitt 9.4](#)) hat den Vorteil, dass es mit Netzwerkproblemen und hohen Latenzzeiten sehr gut umgehen kann. Auch Transaktionen unterstützt Messaging bestens.

Auf Ebene der Datenbank ist ein gemeinsames Schema Datenreplikation nicht zu empfehlen ([Abschnitt 9.5](#)). Dadurch wären die Microservices zu eng aneinander gekoppelt, da sie eine gemeinsame interne Datenrepräsentation hätten. Die Daten müssen in ein anderes Schema repliziert werden. Das Schema kann den Anforderungen des jeweiligen Microservice Rechnung tragen. Weil Microservices BOUNDED CONTEXTS sind, ist es unwahrscheinlich, dass die Microservices dieselben Datenmodellierungen nutzen wollen.

Schließlich sind Schnittstellen eine wichtige Grundlage Schnittstellen und Versionen der Kommunikation und Integration ([Abschnitt 9.6](#)). Die Schnittstellen sind unterschiedlich leicht änderbar: Öffentliche Schnittstellen sind praktisch gar nicht änderbar, weil zu viele Systeme von den Schnittstellen abhängen. Interne Schnittstellen sind einfacher änderbar. Öffentliche Schnittstellen sind im einfachsten Fall Routing bestimmter Funktionalitäten auf passende Microservices. Semantic Versioning hilft, Versionsnummern eine Bedeutung zu geben. Um eine möglichst große Kompatibilität zu gewährleisten, ist das Robustheitsprinzip hilfreich.

Dieser Abschnitt sollte gezeigt haben, dass Microservices nicht einfach nur Dienste sind, die RESTful HTTP nutzen. Das ist höchstens eine Option für die Kommunikation der Microservices untereinander.

Wesentliche Punkte

- Auf der UI-Ebene ist die Integration mit HTML-Oberflächen besonders einfach. SPAs, Desktop-Anwendungen oder Mobile Apps sind Deployment-Monolithen, sodass Änderungen an der Oberfläche für einen Microservice eng mit anderen Änderungen koordiniert werden müssen.
- Auf der Logik-Ebene bieten REST oder RPC-Ansätze zwar ein einfaches Programmiermodell, aber Messaging erlaubt eine losere Kopplung und kann besser mit den Herausforderungen verteilter Kommunikation über das Netz umgehen.
- Datenreplikation erlaubt den performanten Zugriff auch auf große Mengen Daten. Die Microservices dürfen auf keinen Fall dasselbe Schema für die Daten nutzen, da dann die interne Datenrepräsentation nicht mehr geändert werden kann.

9.8 Links & Literatur

- [1] http://en.wikipedia.org/wiki/Single-page_application
- [2] <http://roca-style.org/>
- [3] Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, Oliver Wolf: REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web, dpunkt.verlag, 2015, ISBN 978-3864901201
- [4] <https://developers.google.com/protocol-buffers/>
- [5] <https://www.amqp.org/>
- [6] <https://www.rabbitmq.com/>
- [7] <https://jcp.org/en/jsr/detail?id=343>
- [8] <https://angularjs.org/>
- [9] <http://emberjs.com/>
- [10] <http://www.sencha.com/products/extjs/>
- [11] <https://docs.angularjs.org/guide/module>
- [12] <https://thrift.apache.org/>
- [13] <https://www.innoq.com/blog/st/2014/11/web-based-frontend-integration/>
- [14] <https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919>
- [15] <http://swagger.io>
- [16] <http://json-schema.org/>
- [17] <http://www.w3.org/XML/Schema>
- [18] <http://relaxng.org/>
- [19] <http://www.w3.org/TR/xlink11/>
- [20] http://stateless.co/hal_specification.html
- [21] <http://kafka.apache.org/>
- [22] <http://zeromq.org/>
- [23] <http://tools.ietf.org/html/rfc4287>
- [24] <http://tools.ietf.org/html/rfc5023>
- [25] <http://semver.org/>
- [26] <http://tools.ietf.org/html/rfc793#section-2.10>
- [27] <http://martinfowler.com/bliki/TolerantReader.html>
- [28] http://httpd.apache.org/docs/2.2/mod/mod_include.html
- [29] <http://activemq.apache.org/>

- [30] <http://hornetq.jboss.org/>
- [31] <https://www.varnish-cache.org/>
- [32] <http://www.squid-cache.org/>
- [33] http://nginx.org/en/docs/http/ngx_http_ssi_module.html

10 Architektur eines Microservice

Bei der Umsetzung der Microservices sind einige Punkte zu beachten. Dieses Kapitel betrachtet zunächst die fachliche Architektur eines Microservice ([Abschnitt 10.1](#)). Für die Implementierung eines solchen kann CQRS ([Abschnitt 10.2](#)) interessant sein. Der Ansatz trennt die Änderung der Daten von den Lese-Operationen. Event Sourcing ([Abschnitt 10.3](#)) stellt Ereignisse in den Mittelpunkt der Modellierung. Der Aufbau eines Microservice kann einer hexagonalen Architektur entsprechen ([Abschnitt 10.4](#)), die Funktionalitäten in einen Logikkern und Adapter unterteilt. [Abschnitt 10.5](#) stellt Resilience und Stabilität als wesentliche Anforderungen an Microservices in den Mittelpunkt. Technische Möglichkeiten für die Implementierung von Microservices wie beispielsweise Reactive diskutiert [Abschnitt 10.6](#).

10.1 Fachliche Architektur

Die fachliche Architektur eines Microservice definiert, wie der Microservice seine Fachlichkeiten aufteilt. Das Ziel einer Microservices-Architektur ist, diese Entscheidung nicht über alle Microservices festzuschreiben. So kann der interne Aufbau der Microservices frei entschieden werden. Das erlaubt ein weitgehend unabhängiges Agieren der Teams. Sicher ist es sinnvoll, sich an etablierten Regeln zu orientieren und dadurch den Microservice einfach verstehbar, wartbar und auch ersetzbar zu halten. Aber es muss nicht zwangsläufig Vorschriften auf dieser Ebene geben.

Dieser Abschnitt zeigt, wie man Probleme bei der fachlichen Architektur in einem Mikroservice identifizieren kann. Ob tatsächlich ein Problem vorliegt und wie es zu lösen ist, muss das jeweils verantwortliche Team beantworten.

Die fachliche Architektur des Gesamtsystems beeinflusst die fachliche Architektur der einzelnen Microservices. Wie in [Abschnitt 8.1](#) dargestellt, sollten Microservices untereinander lose gekoppelt sein. Außerdem sollten die Microservices intern eine hohe Kohäsion haben. Der Microservice sollte nur eine fachliche Aufgabe erfüllen. Dementsprechend müssen die Teile des Microservice eng kollaborieren und der Microservice muss eine hohe Kohäsion haben. Ist das nicht der Fall, nimmt der Microservice wahrscheinlich mehr als eine Aufgabe wahr. Wenn die Kohäsion innerhalb des Microservice nicht hoch genug ist, kann der Microservice in mehrere Microservices aufgeteilt werden. Durch die Aufteilung bleiben die Microservices klein und damit einfach verstehbar, wartbar und ersetzbar.

Kapselung bedeutet, dass ein Teil der Architektur keine internen Informationen nach außen sichtbar macht – insbesondere keine internen Datenstrukturen. Stattdessen soll der Zugriff über eine Schnittstelle erfolgen. Dadurch bleibt die Software leicht änderbar: Interne Strukturen können geändert werden, ohne dass andere Teile des Systems beeinflusst werden. Aus diesem Grund dürfen Microservices auf keinen Fall anderen Microservices Zugriff auf ihre internen Datenstrukturen erlauben. Sonst können diese Datenstrukturen nicht mehr

modifiziert werden. Außerdem muss so jeder Microservice von einem anderen Microservice nur die Schnittstelle verstehen, was die Übersichtlichkeit und Verständlichkeit des Systems erhöht.

Domain-Driven Design ist eine Möglichkeit, einen Microservice intern zu strukturieren. Jeder Microservice kann ein DDD-Domänenmodell haben. Die dazu notwendigen Patterns aus dem Domain-Driven Design hat [Abschnitt 4.3](#) bereits erläutert. Gerade wenn Domain-Driven Design und Strategic Design die Struktur des Gesamtsystems definieren ([Abschnitt 8.1](#)), sollten die Microservices diese Ansätze auch nutzen. Strategic Design orientiert sich bei der Entwicklung des Gesamtsystems daran, welche Domänenmodelle es gibt und wie sie über die Microservices verteilt sind.

Transaktionen bündeln mehrere Aktionen so, dass sie gemeinsam ausgeführt werden oder gar nicht. Eine Transaktion kann kaum mehr als einen Microservice umfassen. Lediglich Messaging kann gegebenenfalls Transaktionen über Microservices hinweg unterstützen (siehe [Abschnitt 9.4](#)). Das fachliche Design innerhalb eines Microservice gewährleistet, dass jede Operation an der Schnittstelle einer Transaktion entspricht. So wird vermieden, dass mehrere Microservices an einer Transaktion teilnehmen müssen. Das wäre technisch nur schwer umsetzbar.

10.2 CQRS

Systeme speichern meistens einen Zustand. Operationen können Daten ändern oder lesen sie aus. Die beiden Arten von Operationen können separiert werden: Es lassen sich ändernde Operationen mit Seiteneffekten (Commands) von lesenden Operationen (Queries) unterscheiden. Eine Operation darf nicht gleichzeitig den Zustand ändern und Daten zurückgeben. Durch diese Unterscheidung ist ein System einfacher zu verstehen: Wenn eine Operation einen Wert zurückgibt, ist sie eine Query und ändert keine Werte. Daraus folgen weitere Vorteile. Beispielsweise können Queries mit einem Cache versehen werden. Würden lesende Operationen auch Werte ändern, wäre das Hinzufügen eines Caches nicht so einfach, weil Operationen mit Seiteneffekt trotz Cache immer noch ausgeführt werden müssen. Die Separierung von Queries und Commands nennt man CQS (Command Query Separation). Das Prinzip ist nicht auf Microservices begrenzt, sondern allgemein nutzbar. Beispielsweise können Klassen in einem objektorientierten System Operationen so trennen.

CQRS (Command Query Responsibility Segregation) [\[2\]](#) geht einen Schritt weiter als CQS und trennt die Verarbeitung von Queries und Commands vollständig.

Abb. 10–1 Prinzipieller Aufbau eines CQRS-Systems

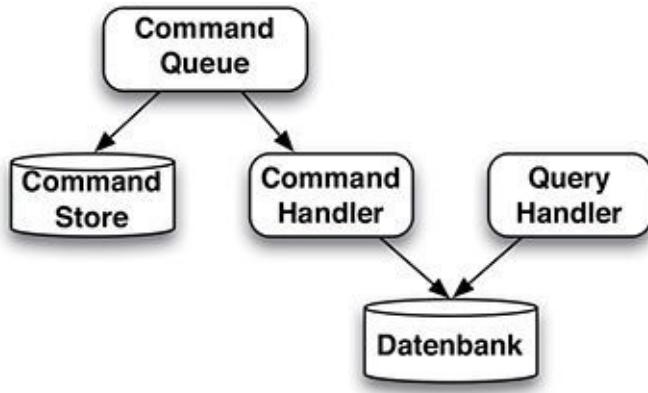


Abbildung 10–1 zeigt den Aufbau eines CQRS-Systems. Jedes Command wird im Command Store gespeichert. Zusätzlich kann es Command Handlers geben. Der Command Handler im Beispiel nutzt die Commands, um den aktuellen Zustand der Daten in einer Datenbank abzulegen. Ein Query Handler nutzt diese Datenbank, um Queries abzuarbeiten. Die Datenbank kann auf die Bedürfnisse des Query Handlers abgestimmt sein. So kann eine Datenbank zur Analyse der Bestellvorgänge ganz anders aussehen als die Datenbank, die ein Kunde zur Anzeige der eigenen Bestellvorgänge nutzt. Es können ganz unterschiedliche Technologien für die Query-Datenbank verwendet werden. So ist die Nutzung eines In-Memory-Caches möglich, der die Daten bei einem Ausfall des Servers verliert. Die Persistenz der Informationen ist durch den Command Store garantiert. Der Inhalt des Caches kann in einem Notfall durch den Command Store rekonstruiert werden.

CQRS kann mit Microservices umgesetzt werden:

Microservices und CQRS

- Die Kommunikationsinfrastruktur kann die Command Queue direkt anbieten, wenn eine Messaging-Lösung genutzt wird. Bei Ansätzen wie REST muss ein Microservice als Implementierung der Command Queue die Commands an alle interessierten Command Handler weitergeben.
- Jeder Command Handler kann ein eigener Microservice sein. Er kann auf die Commands mit seiner eigenen Logik reagieren. Dadurch kann Logik sehr einfach auf mehrere Microservices verteilt werden.
- Ebenso kann ein Query Handler ein eigener Microservice sein. Die Änderungen am Datenbestand, den der Query Handler nutzt, können durch einen Command Handler im selben Microservice geschehen. Der Command Handler kann aber auch ein getrennter Microservice sein. Dann muss der Query Handler eine geeignete Schnittstelle für den Zugriff auf die Datenbank anbieten, sodass der Command Handler den Datenbestand ändern kann.

CQRS hat einige Vorteile gerade im Zusammenspiel mit

Vorteile

Microservices:

- Das Lesen und Schreiben von Daten kann in eigene Microservices getrennt werden. So werden noch kleinere Microservices möglich. Das kann sinnvoll sein, wenn das Schreiben und Lesen so komplex ist, dass ein einziger Microservice für beides zu groß und zu schwer zu verstehen wäre.
- Ebenso kann ein anderes Modell für das Schreiben und Lesen genutzt werden. Microservices können jeweils einen eigenen BOUNDED CONTEXT darstellen und daher

unterschiedliche Datenmodellierungen nutzen. So können für einen Einkauf in einem E-Commerce-Shop sehr viele Daten geschrieben werden – aber statistische Auswertungen lesen nur wenige Daten. Technisch können die Daten durch Denormalisierung für Lese-Operationen oder mit anderen Mitteln für bestimmte Queries optimiert werden.

- Schreiben und Lesen kann unterschiedlich skaliert werden, indem unterschiedlich viele Query-Handler-Microservices und Command-Handler-Microservices gestartet werden. Das kommt der feingranularen Skalierbarkeit von Microservices entgegen.
- Durch die Command Queue ist es möglich, mit Lastspitzen beim Schreiben umzugehen. Die Queue puffert die Änderungen, die dann später abgearbeitet werden. Allerdings wird dann eine Änderung an den Daten nicht sofort bei Queries berücksichtigt.
- Es ist einfach, unterschiedliche Versionen der Command Handler parallel zu betreiben. Dadurch ist es leichter, Microservices in neuen Versionen zu deployen.

CQRS kann dazu dienen, Microservices noch kleiner zu machen, auch wenn die Operationen und Daten eigentlich sehr eng zusammenhängen. Die Entscheidung für oder gegen CQRS kann jeder Microservice einzeln treffen. Eine Schnittstelle, die Operationen zum Ändern und Auslesen von Daten anbietet, kann unterschiedlich implementiert sein. CQRS ist nur eine Option. Beide Aspekte können auch ohne CQRS in nur einem Microservice implementiert werden. Die Freiheit, verschiedene Ansätze nutzen zu können, ist einer der Hauptvorteile der Microservices-Architekturen.

CQRS führt auch zu einigen Herausforderungen:

Herausforderungen

- Transaktionen, die Lese- und Schreiboperationen umfassen, sind nur schwer umsetzbar. Die betroffenen Operationen können in unterschiedlichen Microservices umgesetzt sein. Dann ist das Zusammenfassen der Operationen zu einer Transaktion kaum möglich, weil Transaktionen über Microservices hinweg meistens unmöglich sind.
- Konsistenz der Daten über verschiedene Systeme kann nur schwer gewährleistet werden. Die Verarbeitung der Events ist asynchron, sodass unterschiedliche Knoten die Bearbeitung zu unterschiedlichen Zeitpunkten abgeschlossen haben können.
- Der Aufwand für die Entwicklung und die Infrastruktur ist höher. Es gibt mehr Systembestandteile und komplexere Kommunikationstechnologien.

Es ist nicht sinnvoll, jeden Microservice mit CQRS umzusetzen. Aber der Ansatz ist an vielen Stellen eine gute Ergänzung zu Microservices-Architekturen.

10.3 Event Sourcing

Event Sourcing [1] hat einen ähnlichen Ansatz wie CQRS. Die Events aus Event Sourcing unterscheiden sich jedoch von den Commands aus CQRS. Commands sind spezifisch: Sie definieren genau, was in einem Objekt geändert werden soll. Events enthalten Informationen über etwas, was passiert ist. Beide Ansätze können auch kombiniert werden: Ein Command kann ein Datum ändern. Daraus resultieren Events, auf die andere

Komponenten des Systems reagieren können.

Event Sourcing speichert statt des Zustandes die Ereignisse, die zum aktuellen Zustand geführt haben. Der Zustand selber wird nicht gespeichert – lässt sich aber aus den Events rekonstruieren.

Abb. 10–2 Event Sourcing im Überblick

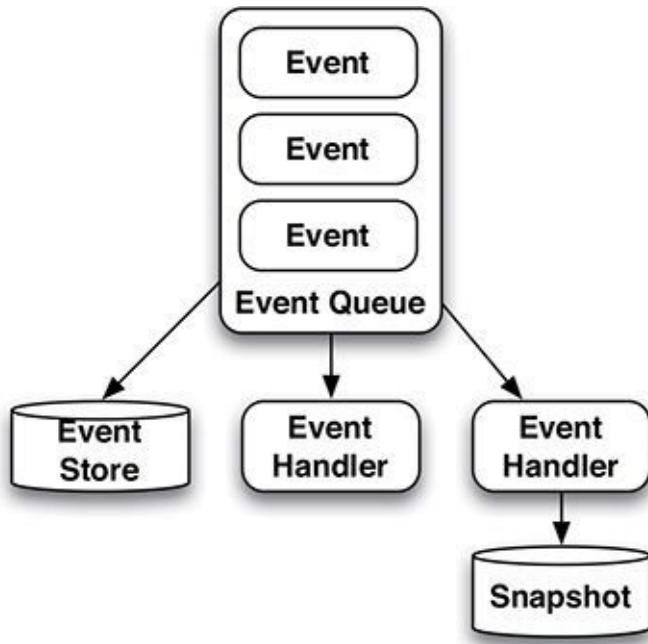


Abbildung 10–2 zeigt Event Sourcing im Überblick:

- Die *Event Queue* schickt alle Events an die verschiedenen Empfänger. Sie kann beispielsweise mit einer Messaging-Middleware implementiert sein.
- Der *Event Store* speichert alle Events. Dadurch ist es jederzeit möglich, die Abfolge der Events und die Events selber nachzuvollziehen.
- Ein *Event Handler* reagiert auf die Events. Er kann Geschäftslogik enthalten, die auf Events reagiert.
- In dem System sind nur die Events einfach nachvollziehbar – der aktuelle Zustand des Systems hingegen nicht. Daher kann es sinnvoll sein, einen *Snapshot* zu pflegen, der den aktuellen Zustand enthält. Bei jedem Event oder nach einer bestimmten Zeit werden die Daten im Snapshot entsprechend der neuen Events geändert. Der Snapshot ist optional. Es ist auch möglich, den Zustand aus den Events ad hoc zu rekonstruieren.

Events dürfen nicht im Nachhinein geändert werden. Fehlerhafte Events müssen durch neue Events korrigiert werden.

Event Sourcing beruft sich auf Domain-Driven Design (siehe [Abschnitt 4.3](#)). Daher sollten die Events im Sinne von UBIQUITOUS LANGUAGE einen Namen tragen, der auch im Geschäftskontext sinnvoll ist. In einigen Domänen ist eine Modellierung mit Events fachlich besonders sinnvoll. So können Buchungen auf einem Konto als Events aufgefasst werden. Anforderungen wie Auditing sind mit Event Sourcing viel einfacher umsetzbar: Da die Buchungen als Event modelliert sind, ist sehr einfach nachvollziehbar, welche Buchung von wem vorgenommen worden ist. Ebenso ist es relativ einfach, einen

historischen Datenbestand zu rekonstruieren. Event Sourcing kann schon aus einer fachlichen Perspektive sinnvoll sein. Generell sind Ansätze wie Event Sourcing in komplexen Domänen sinnvoll, die auch von Domain-Driven Design profitieren.

Event Sourcing hat ähnliche Vor- und Nachteile wie CQRS und beide Ansätze können leicht miteinander kombiniert werden. Besonders sinnvoll ist Event Sourcing, wenn das Gesamtsystem mit einer Event-driven Architecture arbeitet ([Abschnitt 8.6](#)). Dann schicken sich die Microservices sowieso schon Events über Zustandsänderungen und es ist sinnvoll, diesen Ansatz auch in den Microservices zu nutzen.

Selber ausprobieren und experimentieren

Wähle ein dir bekanntes Projekt.

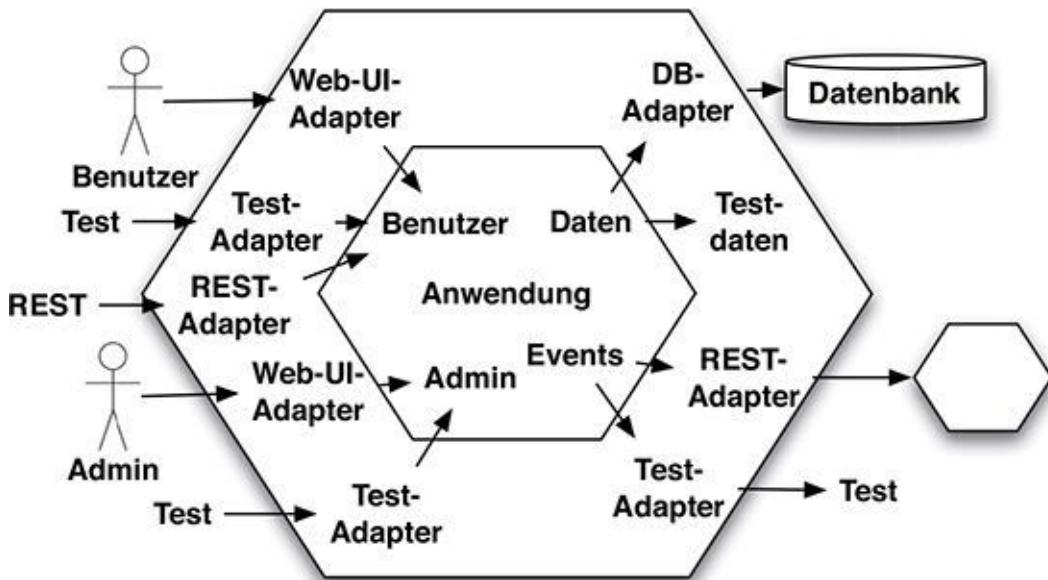
- An welchen Stellen würde Event Sourcing sinnvoll sein? Warum? Wäre Event Sourcing isoliert an einigen Stellen einsetzbar oder müsste das gesamte System auf Events umgestellt werden?
- Wo könnte CQRS hilfreich sein? Warum?
- Folgen die Schnittstellen der CQR-Regel? Dann müssten in allen Schnittstellen die lesenden und schreibenden Operationen getrennt sein.

10.4 Hexagonale Architekturen

Im Zentrum der hexagonalen Architektur [6] steht die Logik der Anwendung ([Abb. 10–3](#)). Die Logik enthält nur die Geschäftsfunktionalitäten. Sie hat verschiedene Schnittstellen, die jeweils durch eine Kante des Hexagons repräsentiert sind. Im Beispiel sind dies die Schnittstelle für die Interaktion mit Benutzern und die Schnittstelle für administrative Eingriffe in die Anwendung. Diese Schnittstellen können Nutzer mithilfe einer Web-Schnittstelle nutzen, die HTTP-Adapter umsetzen. Für Tests gibt es spezielle Adapter. Sie ermöglichen es den Tests, Nutzer zu simulieren. Schließlich gibt es einen Adapter, der die Logik auch über REST zugänglich macht. So können andere Microservices die Logik aufrufen.

Schnittstellen nehmen nicht nur Anfragen von anderen Systemen an. Auch die anderen Systeme werden durch solche Schnittstellen angesprochen: die Datenbank durch den DB-Adapter, der tatsächlich eine Datenbank nutzt. Die Alternative ist ein Adapter für Testdaten. Und schließlich kann eine andere Anwendung durch einen REST-Adapter angesprochen werden – und auch statt dieses Adapters kann ein Testsystem verwendet werden, das den genutzten Service simuliert.

Abb. 10–3 Überblick hexagonale Architektur



Ein anderer Name für hexagonale Architekturen ist »Ports and Adapters« (etwa: Schnittstellen und Adapter). Jede Facette der Anwendung wie Benutzer, Admin, Daten oder Event ist ein Port. Die Adapter setzen die Ports auf Technologien wie REST oder Weboberflächen um. Mit den Ports auf der rechten Seite des Hexagons holt die Anwendung sich Daten, während mit den Ports auf der linken Seite die eigenen Funktionalitäten und Daten für Nutzer und andere Systeme angeboten werden.

Die hexagonale Architektur teilt ein System in einen Logikkern und Adaptern auf. Nur die Adapter ermöglichen die Kommunikation nach außen.

Die hexagonale Architektur ist eine Alternative zu Hexagone oder Schichten? einer Schichten-Architektur. In einer Schichten-Architektur gibt es eine Schicht, in der die UI implementiert ist, und eine Schicht, in der die Persistenz implementiert wird. In einer hexagonalen Architektur sind das Adapter, die durch Ports mit der Logik verbunden sind. Die hexagonale Architektur stellt klar da, dass es mehr Ports als nur Persistenz und UI geben kann. Außerdem wird durch den Begriff »Adapter« verdeutlicht, dass die Logik und die Ports von den konkreten Protokollen und Implementierungen der Adapter getrennt sein sollen.

Mit hexagonalen Architekturen ist es natürlich, Logik nicht nur für andere Microservices über eine REST-Schnittstelle anzubieten, sondern auch für Benutzer über eine Web-UI. Genau diese Idee liegt auch Microservices zugrunde. Sie sollen nicht nur Logik für andere Microservices bereitstellen, sondern auch direkte Interaktion von Benutzern durch eine UI unterstützen.

Hexagonale Architekturen und Microservices

Weil für alle Ports eigene Testimplementierungen umgesetzt werden können, ist das isolierte Testen eines Microservice mit einer hexagonalen Architektur einfacher. Dazu müssen nur die Testadapter statt der eigentlichen Implementierungen verwendet werden. Gerade das unabhängige Testen einzelner Microservices ist eine wichtige Voraussetzung für die unabhängige Implementierung und das unabhängige Deployment von Microservices.

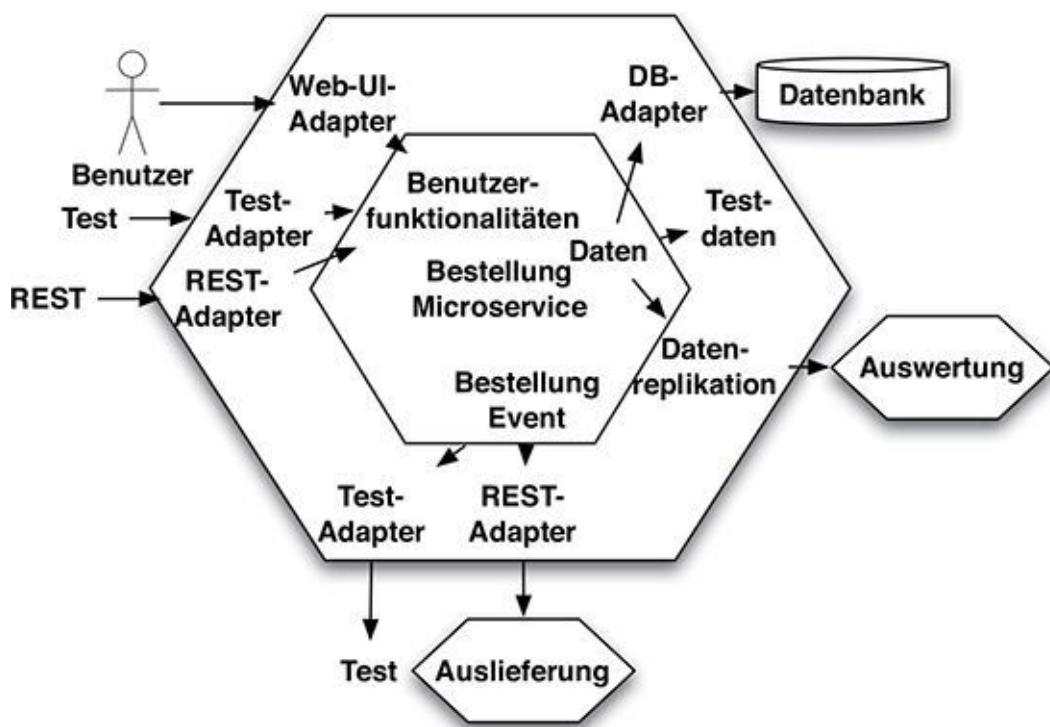
In den Adapter kann auch notwendige Logik für Resilience und Stabilität (siehe [Abschnitt 10.5](#)) oder Load Balancing ([Abschnitt 8.10](#)) implementiert werden.

Denkbar ist es ebenso, die Adapter und die eigentliche Logik in eigene Microservices zu trennen. Das erzeugt zwar mehr verteilte Kommunikation und damit einen Overhead. Auf der anderen Seite kann die Implementierung von Adapter und Kern auf verschiedene Teams aufgeteilt werden. Beispielsweise kann ein Team, das einen mobilen Client entwickelt, einen speziellen Adapter implementieren, der auf die Bandbreitenbeschränkungen mobiler Anwendungen abgestimmt ist (siehe auch [Abschnitt 9.1](#)).

Als Beispiel für eine hexagonale Architektur soll ein Ein Beispiel Microservice für Bestellungen dienen ([Abb. 10–4](#)). Der Benutzer kann über die Web-UI die Funktionalitäten des Microservice nutzen, um Bestellungen aufzugeben. Ebenso gibt es eine REST-Schnittstelle, mit der andere Microservices oder externe Clients die Funktionalitäten nutzen können. Die Web-UI, die REST-Schnittstelle und der Testadapter sind drei Adapter für die Schnittstelle Benutzerfunktionalitäten des Microservice. Die Umsetzung als eine Schnittstelle mit drei Adapters stellt in den Vordergrund, dass REST und die Web-UI nur zwei Möglichkeiten sind, dieselben Funktionalitäten zu nutzen. Außerdem werden so Microservices umgesetzt, die UI und REST integrieren. Technisch können die Adapter immer noch in eigenen Microservices umgesetzt werden.

Eine weitere Schnittstelle sind die Bestellung-Events. Sie melden dem Microservice »Auslieferung«, wenn neue Bestellungen angekommen sind, sodass die Bestellungen ausgeliefert werden können. Über diese Schnittstelle kommuniziert der Microservice »Auslieferung« auch, wenn eine Bestellung ausgeliefert worden ist oder es zu Verzögerungen kommt. Auch diese Schnittstelle kann durch einen Adapter für Tests bedient werden. Die Schnittstelle zum Microservice »Auslieferung« schreibt also nicht einfach nur Daten, sondern kann auch Änderungen an den Bestellungen vornehmen. Die Schnittstelle nutzt also andere Microservices, nimmt aber auch selber Änderungen entgegen.

Abb. 10–4 Bestellung als Microservice mit einer hexagonalen Architektur



Die hexagonale Architektur hat eine fachliche Aufteilung in eine Schnittstelle für Benutzerfunktionalitäten und eine Schnittstelle für Bestellung-Events. So stellt die Architektur den fachlichen Schnitt in den Vordergrund.

Der Zustand der Bestellungen wird in einer Datenbank gespeichert. Auch dafür gibt es eine Schnittstelle, bei der für Tests statt der Datenbank Testdaten genutzt werden können. Diese Schnittstelle entspricht der Persistenz-Schicht in einer klassischen Architektur.

Schließlich gibt es eine Schnittstelle, die über Datenreplikation die Informationen über die Bestellungen an die Auswertung übergibt. Dort können Statistiken aus den Bestellungen erstellt werden. Diese Auswertung scheint eine Persistenz-Schnittstelle zu sein, ist aber mehr: Die Daten werden nicht einfach gespeichert, sondern für Statistiken aufbereitet.

Wie das Beispiel zeigt, erzeugt eine hexagonale Architektur eine gute fachliche Aufteilung in verschiedene fachliche Schnittstellen. Jede fachliche Schnittstelle und jeder Adapter können gegebenenfalls als eigener Microservice ausgeführt werden. So kann die Anwendung in eine Vielzahl von Microservices aufgeteilt werden, wenn das notwendig ist.

Selber ausprobieren und experimentieren

Wähle ein dir bekanntes Projekt.

- Welche einzelnen Hexagone würde es geben?
- Welche Ports und Adapter würden die Hexagone haben?
- Welche Vorteile würde eine hexagonale Architektur bieten?
- Wie würde die Umsetzung aussehen?

10.5 Resilience und Stabilität

Der Ausfall eines Microservice sollte möglichst wenige Auswirkungen auf die Verfügbarkeit der anderen Microservices haben. Da ein Microservice-System ein verteiltes System ist, ist die Gefahr eines Ausfalls wesentlich höher: Netzwerke und Server sind unzuverlässig. Da Microservices auf mehrere Server verteilt sind, ist die Anzahl der Server für ein System höher und damit auch die Ausfallwahrscheinlichkeit. Wenn der Ausfall eines Microservice auch die abhängigen Microservices zum Ausfall bringt, kann ein Microservice den nächsten Microservice zum Ausfall bringen und so schrittweise das gesamte System ausfallen. Das gilt es zu vermeiden.

Aus diesem Grund muss ein Microservice gegen den Ausfall anderer Microservices abgesichert sein. Diese Eigenschaft nennt man Resilience. Die dazu notwendigen Maßnahmen müssen im Microservice ergriffen werden. Weiter gefasst ist Stabilität. Damit ist eine möglichst hohe Verfügbarkeit der Software gemeint. »Release It!« [3] führt dazu mehrere Patterns auf:

Timeouts dienen dazu, bei der Kommunikation mit Timeout

einem anderen System einen Ausfall festzustellen. Wenn nach dem Timeout keine Antwort zurückgekommen ist, gilt das System als ausgefallen. Leider haben viele APIs keine Möglichkeit, Timeouts zu definieren, und einige Timeouts sind sehr hoch. Auf Betriebssystemebene können TCP-Timeouts fünf Minuten betragen. So lange gibt der Microservice an einen Aufrufer keine Antwort zurück, weil der Service auf den anderen Microservice wartet. Damit scheint auch dieser Microservice ausgefallen zu sein. Außerdem kann die Anfrage für die Zeit einen Thread blockieren. Irgendwann sind alle Threads blockiert und der Microservice kann keine weiteren Anfragen mehr annehmen. Genau so einen Domino-Effekt gilt es zu vermeiden. Wenn die API für den Zugriff auf ein anderes System oder auf die Datenbank einen Timeout vorsieht, sollte man ihn setzen. Die Alternative ist, alle Abfragen auf externe Systeme oder Datenbanken in einem eigenen Thread stattfinden zu lassen und den Thread nach einem Timeout abzubrechen.

Ein Circuit Breaker ist eine Sicherung in einem Stromkreis. Bei einem Kurzschluss unterbricht die Sicherung die Stromversorgung, um Schlimmeres wie ein Überhitzen oder einen Brand zu verhindern. Diese Idee lässt sich auf Software übertragen: Wenn ein anderes System nicht mehr verfügbar ist oder nur noch Fehler zurückgibt, unterbindet ein Circuit Breaker den Aufruf des Systems. Aufrufe sind dann sowieso sinnlos.

Normalerweise ist der Circuit Breaker geschlossen und Aufrufe werden an das andere System weitergegeben. Wenn Fehler auftreten, wird abhängig von der Fehlerfrequenz der Circuit Breaker geöffnet. Aufrufe gehen dann nicht mehr an das andere System, sondern laufen sofort auf einen Fehler. So wird das andere System entlastet. Und es wird auch nicht auf einen Timeout gewartet. Nach einiger Zeit schließt der Circuit Breaker sich wieder. Ein Aufruf wird nun an das andere System weitergeleitet. Wenn der Fehler immer noch vorhanden ist, öffnet sich der Circuit Breaker wieder.

Der Circuit Breaker kann mit einem Timeout kombiniert werden. Ein Timeout kann den Circuit Breaker öffnen. Der Zustand des Circuit Breakers zeigt dem Betrieb, wo aktuell Probleme im System sind. Ein geöffneter Circuit Breaker bedeutet, dass ein Microservice mit einem anderen Microservice nicht mehr kommunizieren kann. Daher sollte der Zustand der Circuit Breaker für den Betrieb im Monitoring angezeigt werden.

Wenn der Circuit Breaker offen ist, muss nicht unbedingt ein Fehler erzeugt werden. Es ist genauso gut möglich, die Funktionalität einzuschränken. Nehmen wir an, dass ein Geldautomat nicht überprüfen kann, ob das Konto gedeckt ist, weil das dafür zuständige System nicht erreichbar ist. Dennoch können bis zu einem gewissen Limit Abhebungen gestattet werden. Schließlich sind Kunden sonst unzufrieden. Außerdem kann die Bank für die Abhebungen keine Gebühren verlangen. Ihr entgeht Umsatz. Ob und bis zu welchem Limit eine Abhebung ermöglicht wird, ist eine Geschäftsentcheidung. Es gilt, das Umsatzpotenzial gegen den möglichen Schaden abzuwägen. Es kann auch andere Regeln geben, die bei einem Ausfall eines anderen Systems angewendet werden. Anfragen können zum Beispiel aus einem Cache beantwortet werden. Wichtiger als die technischen Möglichkeiten sind die fachlichen Anforderungen, um das geeignete Vorgehen beim Ausfall eines Systems auszuwählen.

Ein Schott (Bulkhead) ist eine spezielle Tür in einem Schiff, die wasserdicht geschlossen werden kann. Sie

unterteilen das Schiff in mehrere Bereiche. Wenn Wasser eindringt, ist nur ein Teil des Schiffs betroffen und der Untergang des Schiffs wird verhindert.

Für Software sind ähnliche Ansätze möglich: Das Gesamtsystem muss in einzelne Bereiche unterteilt werden. Ein Ausfall oder ein Problem in einem Teil darf die anderen Teile nicht beeinflussen. Beispielsweise kann es mehrere Instanzen eines Microservice für verschiedene Clients geben. Wenn ein Client die Microservices überlastet, werden die anderen Clients nicht in Mitleidenschaft gezogen. Ähnliches gilt für Ressourcen wie Datenbankverbindungen oder Threads. Wenn unterschiedliche Teile eines Microservice unterschiedliche Pools für diese Ressourcen nutzen, kann ein Teil die anderen Teile nicht blockieren, selbst wenn er alle seine Ressourcen aufbraucht.

In Microservices-Architekturen bilden die Microservices selber getrennte Bereiche. Das ist insbesondere der Fall, wenn jeder Microservice seine eigene virtuelle Maschine mitbringt. Selbst wenn der Microservice die komplette virtuelle Maschine zum Absturz bringt oder vollständig auslastet, hat das wenige Auswirkungen auf die anderen Microservices. Sie laufen in anderen virtuellen Maschinen und sind deswegen getrennt.

Mit Steady State (etwa: fester oder sicherer Zustand) Steady State ist gemeint, dass Systeme so aufgebaut sein sollen, dass sie ewig laufen können. Das bedeutet zum Beispiel, dass Systeme nicht immer mehr Daten speichern. Sonst würde das System irgendwann alle Kapazitäten erschöpfen und dann abstürzen. Log-Dateien müssen beispielsweise irgendwann gelöscht werden. Sie sind meistens nur für eine bestimmte Zeit interessant. Ein anderer Bereich ist Caching: Wenn ein Cache immer nur wächst, wird er irgendwann allen Speicher ausfüllen. Die Werte müssen aus dem Cache auch wieder entfernt werden.

Timeouts sind nur notwendig, weil ein anderes System Fail Fast sehr lange für eine Antwort benötigt. Die Idee von Fail Fast ist, das Problem von der anderen Seite anzugehen: Jedes System soll Fehler möglichst frühzeitig und schnell anzeigen. Wenn eine Anfrage einen bestimmten Dienst benötigt und dieser Dienst gerade nicht zur Verfügung steht, kann die Anfrage gleich mit einer Fehlermeldung beantwortet werden. Gleiches gilt, wenn andere Ressourcen derzeit nicht zur Verfügung stehen. Ebenso kann die Anfrage gleich am Anfang validiert werden. Wenn sie Fehler enthält, ist es nicht sinnvoll, sie zu bearbeiten, und es kann gleich ein Fehler zurückgegeben werden. Die Vorteile von Fail Fast sind identisch mit denen, die Timeout bietet: Ein schneller Fehlschlag verbraucht weniger Ressourcen und führt so zu einem stabileren System.

Das Handshaking dient in einem Protokoll dazu, die Kommunikation einzuleiten. Dadurch erlauben Protokolle, dass ein Server bei Überlast weitere Anfragen ablehnt. Das vermeidet weitere Überlastungen, ein Absturz oder zu langsame Antworten. Protokolle wie HTTP unterstützen so etwas leider nicht. Daher muss die Anwendung die Funktionalität beispielsweise mit Health Checks nachbauen. Eine Anwendung kann so signalisieren, dass sie zwar prinzipiell erreichbar ist, aber gerade so viel Last hat, dass weitere Anfragen nicht sinnvoll sind. Protokolle, die auf Socket-Verbindungen aufbauen, können solche Ansätze selber implementieren.

Mit einem Test Harness kann eine Anwendung darauf getestet werden, wie sie sich bei bestimmten Fehlersituationen verhält. Das können Probleme auf der TCP/IP-Ebene sein oder zum Beispiel Antworten anderer Systeme mit HTTP-Headern, aber ohne HTTP-Body. So etwas sollte eigentlich nicht auftreten, da Betriebssystem oder Netzwerk-Stack sie behandeln. Dennoch können solche Fehler in der Praxis vorkommen und haben dann oft verheerende Folgen, weil die Anwendung gar nicht auf sie vorbereitet ist. Ein Test Harness kann eine Erweiterung der Tests sein, die [Abschnitt 11.8](#) diskutiert.

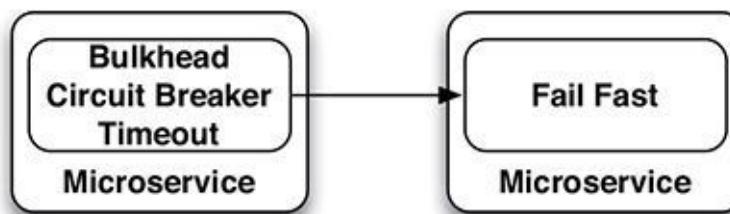
Aufrufe in einem Programm funktionieren nur auf demselben Host zur selben Zeit im selben Prozess.

Synchrone verteilte Kommunikation (z. B. REST) erlaubt die Kommunikation zwischen verschiedenen Hosts und verschiedenen Prozessen zur selben Zeit. Asynchrone Kommunikation mit Messaging-Systemen ([Abschnitt 9.4](#)) erlaubt eine Entkopplung auch über die Zeit. Ein System sollte nicht auf eine Antwort von einem asynchronen Prozess warten. Das System sollte an anderen Aufgaben weiterarbeiten, statt auf die Antwort zu warten. Fehler, bei denen die Systeme nach und nach wie Dominosteine ausfallen, sind mit asynchroner Kommunikation wesentlich weniger wahrscheinlich. Die Systeme sind dazu gezwungen, mit langen Antwortzeiten umzugehen, da asynchrone Kommunikation solche Antwortzeiten sowieso zur Folge haben kann.

Stabilitäts-Pattern wie Bulkhead beschränken Ausfälle auf eine Einheit. Microservices bieten sich als Einheit an.

Sie laufen auf getrennten virtuellen Maschinen und sind dadurch für die meisten Belange schon isoliert. Dadurch ergibt sich das Bulkhead-Pattern in einer Microservices-Architektur ganz natürlich. [Abbildung 10–5](#) zeigt einen Überblick: Ein Microservice kann durch Bulkhead, Circuit Breaker und Timeout die Nutzung anderer Microservices absichern. Der benutzte Microservice kann zusätzlich Fail Fast umsetzen. Die Absicherung durch die Patterns kann in den Teilen eines Microservice umgesetzt werden, die für die Kommunikation mit anderen Microservices zuständig sind. Dadurch ist dieser Aspekt in einem Bereich des Codes implementiert und nicht über den gesamten Code verteilt.

Abb. 10–5 Stabilität bei Microservices



Technisch können die Patterns unterschiedlich umgesetzt werden. Bei Microservices gibt es folgende Optionen:

- Timeouts sind einfach implementierbar: Für den Zugriff auf das andere System wird ein eigener Thread gestartet, der nach dem Timeout abgebrochen wird.
- Circuit Breaker sind auf den ersten Blick nicht besonders komplex und könnten selbst entwickelt werden. Aber die Implementierung muss auch bei höherer Last funktionieren und eine Schnittstelle für den Betrieb zum Monitoring anbieten. Das ist

dann schon nicht mehr so einfach. Daher ist eine eigene Implementierung nicht sehr sinnvoll.

- Bulkheads bringen Microservices sowieso mit, weil ein Problem in vielen Fällen schon auf nur einen Microservice begrenzt ist. Ein Speicherleck wird beispielsweise nur einen Microservice zum Absturz bringen.
- Steady State, Fail Fast, Handshaking und Test Harness muss jeder Microservice implementieren.
- Entkoppelung durch Middleware ist eine Option für die gemeinsame Kommunikation der Microservices.

Das Reactive Manifesto [4] führt Resilience als Resilience und Reactive wesentliche Eigenschaft einer Reactive-Anwendung auf.

In einer Anwendung kann Resilience umgesetzt werden, indem Aufrufe asynchron ausgeführt werden. Jeder Teil der Anwendung, der Nachrichten verarbeitet (»Aktor«), muss überwacht werden. Wenn ein Aktor nicht mehr reagiert, kann er neu gestartet werden. So können Fehler beseitigt werden, sodass die Anwendungen widerstandsfähig sind.

Hystrix [5] implementiert Timeout und Circuit Breaker.

Hystrix

Dazu müssen die Entwickler Aufrufe in Commands kapseln oder es können Java-Annotationen verwendet werden. Die Aufrufe finden in eigenen Thread Pools statt. Es können mehrere Thread Pools angelegt werden. Wenn es pro aufgerufenem Microservice einen Thread Pool gibt, können die Aufrufe der Microservices so voneinander getrennt werden, dass ein Problem mit einem Microservice die Nutzung der anderen Microservices nicht beeinflusst. Das entspricht der Bulkhead-Idee. Hystrix ist eine Java-Bibliothek, die unter der Apache-Lizenz steht und aus dem Netflix-Stack stammt. Die Beispieldaten nutzt Hystrix mit Spring Cloud (siehe [Abschnitt 14.10](#)). Mit einem Sidecar kann Hystrix auch für Anwendungen genutzt werden, die nicht in Java geschrieben sind (siehe [Abschnitt 8.7](#)). Hystrix stellt Informationen über den Zustand der Thread Pools und der Circuit Breaker für das Monitoring und den Betrieb zur Verfügung. Diese Informationen können in einem eigenen Monitoring-Werkzeug angezeigt werden – dem Hystrix-Dashboard. Intern nutzt Hystrix die Reactive Extensions für Java (RxJava). Hystrix ist die Bibliothek mit der größten Verbreitung im Bereich Resilience.

Selber ausprobieren und experimentieren

Dieses Kapitel hat acht Patterns für Stabilität vorgestellt. Priorisiere die Patterns.

- Welche Eigenschaften sind unverzichtbar? Welche sind wichtig? Welche unwichtig?
- Wie kann überprüft werden, ob die Microservices tatsächlich die Patterns umsetzen?

10.6 Technische Architektur

Die technische Architektur eines Microservice kann flexibel gehandhabt werden. Frameworks oder Programmiersprachen müssen nicht einheitlich für alle Microservices sein. Daher können einzelne Microservices durchaus unterschiedliche Plattformen verwenden. Aber bestimmte technische Infrastrukturen sind für Microservices besser geeignet als andere.

Process Engines, die sonst zur Orchestrierung von Services in einer SOA dienen ([Abschnitt 7.1](#)), können in einem Microservice genutzt werden, um einen Geschäftsprozess zu modellieren. Wichtig ist, dass ein Microservice nur einen fachlichen Bereich umsetzt – beispielsweise einen BOUNDED CONTEXT. Ein Microservice sollte nicht zur reinen Integration oder Orchestrierung verschiedener anderer Microservices ohne eigene Logik werden. Sonst muss für eine Änderung nicht nur dieser Microservice, sondern es müssen auch die integrierten Microservices geändert werden. Ziel einer Microservices-Architektur ist aber gerade, die Änderungen möglichst auf einen Microservice zu begrenzen. Wenn es gilt, mehrere Geschäftsprozesse zu implementieren, sollten dazu verschiedene Microservices genutzt werden, die jeweils einen Geschäftsprozess mit den abhängigen Services implementieren. Natürlich wird sich nicht immer vermeiden lassen, dass andere Microservices integriert werden müssen, um einen Geschäftsprozess zu implementieren. Aber ein Microservice, der nur eine Integration darstellt, ist nicht sinnvoll.

Es ist von großem Vorteil, wenn die Microservices zustandslos sind. Genau genommen sollen die Microservices in der Logik-Schicht keinen Zustand speichern. Zustand in der Datenbank oder auf dem Client ist akzeptabel. Durch diesen Ansatz spielt der Ausfall einer Instanz keine besondere Rolle. Die Instanz kann einfach durch eine neue Instanz ersetzt werden. Ebenso kann die Last zwischen mehreren Instanzen verteilt werden – ohne Rücksicht darauf, welche Instanz die vorherigen Anfragen des Nutzers bearbeitet hat. Und schließlich ist ein Deployment einer neuen Version einfacher, weil die alte Version einfach gestoppt und ersetzt werden kann, ohne dass dabei Zustand migriert werden muss.

Die Umsetzung von Microservices mit Reactive-Technologien [[4](#)] kann besonders sinnvoll sein. Diese Ansätze sind vergleichbar mit Erlang (siehe [Abschnitt 15.7](#)): Anwendungen bestehen aus Aktoren. In Erlang heißen sie Prozesse. Arbeit in jedem Aktor ist sequenziell, aber unterschiedliche Aktoren können parallel an unterschiedlichen Nachrichten arbeiten. Das ermöglicht eine parallele Abarbeitung von Aufgaben. Aktoren können Nachrichten an andere Aktoren schicken, die in den Mailboxen der Aktoren landen. I/O-Operationen sind in Reactive-Anwendungen nicht blockierend: Es wird eine Anfrage nach Daten geschickt. Wenn die Daten da sind, wird der Aktor aufgerufen und kann die Daten bearbeiten. In der Zwischenzeit kann er andere Requests bearbeiten.

Wesentliche Eigenschaften sind laut dem Reactive Manifesto:

- **Responsive:**

Das System soll möglichst schnell auf Anfragen reagieren. Das hat unter anderem Vorteile für Fail Fast und damit für die Stabilität (siehe [Abschnitt 10.5](#)). Reactive kann beispielsweise ab einem bestimmten Füllgrad der Mailbox die Annahme

weiterer Nachrichten verweigern. Dann wird der Sender entsprechend langsamer und das System insgesamt nicht überlastet. Andere Anfragen können noch bearbeitet werden. Auch der Verzicht auf blockierende I/O-Operationen kommt dem Ziel Responsive zugute.

- Auf *Resilience* und den Zusammenhang mit Reactive-Anwendungen ist [Abschnitt 10.5](#) bereits eingegangen.
- *Elastic* bedeutet, dass zur Laufzeit neue Systeme gestartet werden können, die sich die Last teilen. Dazu muss das System skalierbar sein und gleichzeitig außerdem zur Laufzeit so geändert werden können, dass die Last auf die verschiedenen Knoten verteilt wird.
- *Message Driven* heißt, dass die einzelnen Bestandteile durch Messaging miteinander kommunizieren. Wie in [Abschnitt 9.4](#) dargestellt, passt diese Kommunikation gut zu Microservices. Reactive-Anwendungen nutzen ganz ähnliche Ansätze auch in der Anwendung selbst.

Reactive kann Microservices besonders einfach umsetzen, da die Ideen aus dem Reactive-Bereich sehr gut zu Microservices passen. Aber es ist auch mit klassischen Technologien möglich, genauso gute Ergebnisse zu erzielen.

Technologien aus dem Bereich Reactive sind beispielsweise:

- Die Programmiersprache *Scala* [13] mit dem Reactive Framework *Akka* [14] und dem darauf basierenden Web-Framework *Play* [14]. Die Frameworks sind auch mit Java nutzbar.
- Es gibt Reactive-Erweiterungen für praktisch alle populären Sprachen [16]. Dazu zählen *RxJava* [17] für Java oder *RxJS* [18] für JavaScript.
- Ähnliche Ansätze unterstützt auch *Vert.x* [19] (siehe auch [Abschnitt 15.6](#)). Dieses Framework basiert zwar auf der JVM, unterstützt aber viele verschiedene Sprachen wie Java, Groovy, Scala, JavaScript, Clojure, Ruby oder Python.

Reactive ist nur eine Möglichkeit, ein System mit [Microservices ohne Reactive?](#) Microservices zu implementieren. Das klassische Programmiermodell mit blockierendem I/O, ohne Akteuren und mit synchronen Aufrufen ist für diese Art von Systemen ebenfalls geeignet. Wie schon erläutert, kann Resilience durch eigene Libraries umgesetzt werden. Elastic ist möglich, indem neue Instanzen der Microservices beispielsweise als virtuelle Maschinen gestartet werden. Und klassische Anwendungen können auch über Messages miteinander kommunizieren. Reactive-Anwendungen haben für Responsive Vorteile. Allerdings muss dazu gewährleistet sein, dass Operationen tatsächlich nicht blockieren. Für I/O-Operationen können die Reactive-Lösungen das meistens garantieren. Aber eine komplexe Berechnung kann das System so blockieren, sodass keine Nachrichten mehr verarbeitet werden und das gesamte System blockiert ist. Ein Microservice muss nicht mit Reactive-Technologien umgesetzt werden – aber es ist sicher eine interessante Alternative.

Selber ausprobieren und experimentieren

Informiere dich näher zu Reactive und Microservices.

- Wie werden die Vorteile genau umgesetzt?
- Gibt es eine Reactive Extension für deine bevorzugte Programmiersprache?
- Welche Features bietet sie?
- Wie hilft das bei der Umsetzung von Microservices?

10.7 Fazit

Die fachliche Architektur eines Microservice muss das Team verantworten, das den Microservice implementiert. Es sollte an möglichst wenige Vorgaben gebunden sein, um so die Unabhängigkeit der Teams zu gewährleisten.

Niedrige Kohäsion kann ein Hinweis auf ein Problem im fachlichen Schnitt des Microservice sein. Domain-Driven Design (DDD) ist eine interessante Möglichkeit, einen Microservice zu strukturieren. Ebenso können Transaktionen Hinweise auf eine sinnvolle fachliche Aufteilung sein: Eine Operation eines Microservice sollte eine Transaktion sein ([Abschnitt 10.1](#)).

CQS (Command Query Separation) unterteilt Operationen eines Microservice oder einer Klasse in lesende Operationen (Queries) und ändernde Operationen (Commands). CQRS (Command Query Responsibility Segregation) ([Abschnitt 10.2](#)) separiert Datenänderung mit Commands von Query Handlern, die Anfragen bearbeiten können. So entstehen Microservices oder Klassen, die nur lesende oder nur schreibende Zugriffe umsetzen. Event Sourcing ([Abschnitt 10.3](#)) speichert Events und stellt so nicht den Zustand in den Vordergrund, sondern die Historie aller Ereignisse. Diese Ansätze sind für den Aufbau von Microservices hilfreich, weil so kleinere Microservices erstellt werden können, die nur lesen oder nur schreiben. Das ermöglicht eine unabhängige Skalierung und Optimierungen für die beiden Operationsarten.

Die hexagonale Architektur ([Abschnitt 10.4](#)) stellt in den Mittelpunkt eines Microservice einen Kern, der über Adapter beispielsweise durch eine UI oder eine API aufgerufen werden kann. Ebenso können Adapter die Nutzung anderer Microservices oder von Datenbanken ermöglichen. Für Microservices ergibt sich dadurch eine Architektur, die UI und REST-Schnittstelle in einem Microservice unterstützt.

Für Resilience und Stabilität hat [Abschnitt 10.5](#) einige Patterns vorgestellt. Wichtig sind vor allem Circuit Breaker, Timeout und Bulkhead. Eine populäre Implementierung ist Hystrix, das nicht nur für Java-Anwendungen genutzt werden kann.

[Abschnitt 10.6](#) hat bestimmte technische Optionen für Microservices dargestellt: Die Nutzung von Process Engines ist beispielsweise eine Option für einen Microservice. Zustandslosigkeit ist vorteilhaft. Und schließlich sind Reactive-Ansätze eine gute Basis für die Implementierung von Microservices.

Damit hat das Kapitel wesentliche Faktoren für die Implementierung einzelner Microservices näher erläutert.

Wesentliche Punkte

- Microservices in einem Microservice-System können unterschiedliche fachliche Architekturen haben.
- Microservices können intern mit Event Sourcing, CQRS oder hexagonaler Architektur umgesetzt werden.
- Technische Eigenschaften wie Stabilität kann nur jeder Microservice einzeln umsetzen.

10.8 Links & Literatur

- [1] <http://de.slideshare.net/mploed/event-sourcing-fr-reaktive-anwendungen>
- [2] <https://speakerdeck.com/owolf/cqrs-for-great-good-2>
- [3] Michael T. Nygard: Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8
- [4] <http://www.reactivemanifesto.org/>
- [5] <https://github.com/Netflix/Hystrix/>
- [6] <http://alistair.cockburn.us/Hexagonal+architecture>
- [7] http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html
- [8] http://nginx.org/en/docs/http/load_balancing.html
- [9] <http://www.haproxy.org/>
- [10] <http://aws.amazon.com/de/elasticloadbalancing/>
- [11] <https://github.com/Netflix/ribbon>
- [12] <https://github.com/hashicorp/consul-template>
- [13] <http://www.scala-lang.org/>
- [14] <http://akka.io/>
- [15] <https://www.playframework.com/>
- [16] <http://reactivex.io/>
- [17] <https://github.com/ReactiveX/RxJava>
- [18] <https://github.com/Reactive-Extensions/RxJS>
- [19] <http://vertx.io/>

11 Testen von Microservices und Microservice-Systemen

Die Aufteilung eines Systems in Microservices hat Auswirkungen auf das Testen. [Abschnitt 11.1](#) erläutert die Motivation für Software-Tests. [Abschnitt 11.2](#) diskutiert grundlegende Ansätze für Tests nicht nur von Microservices. [Abschnitt 11.3](#) zeigt dann, wieso das Testen von Microservices spezielle Herausforderungen bereithält, die es in anderen Systemen so nicht gibt. Ein Beispiel: In einem Microservice-System muss das gesamte System getestet werden ([Abschnitt 11.4](#)), das aus allen Microservices besteht. Das ist aufwendig, weil es sehr viele Microservices geben kann. [Abschnitt 11.5](#) behandelt den Sonderfall, dass es eine Legacy-Anwendung gibt, die durch Microservices abgelöst werden soll. Dann muss die Integration von Microservices und Legacy-Anwendung getestet werden. Das Testen der Microservices alleine reicht nicht. Eine Möglichkeit, die Schnittstellen zwischen Microservices abzusichern, sind Consumer-Driven Contract Tests ([Abschnitt 11.7](#)). Sie reduzieren den Aufwand für die Tests des Gesamtsystems. Natürlich müssen auch die einzelnen Microservices getestet werden. Dann stellt sich aber die Frage, wie die Microservices einzeln ohne andere Microservices überhaupt ausgeführt werden können ([Abschnitt 11.6](#)). Microservices ergeben Technologiefreiheit, aber es muss dennoch bestimmte Standards geben. Daher können die Tests technische Standards umfassen ([Abschnitt 11.8](#)), die in der Architektur definiert worden sind.

11.1 Warum testen?

Testen von Software ist ein wesentlicher Bestandteil jedes Software-Entwicklungsprojekts. Die Frage nach dem Sinn der Tests wird dennoch selten gestellt. Letztendlich sind Tests Risikomanagement. Sie sollen das Risiko verringern, dass Fehler in Produktion auftauchen und die Nutzer die Fehler bemerken – oder sonstige Schäden entstehen.

Aus dieser Bewertung ergeben sich Konsequenzen:

- Jeder Test muss danach bewertet werden, welches konkrete Risiko er minimiert. Er ist nur sinnvoll, wenn er letztendlich konkrete Fehlerszenarien vermeidet, die sonst in Produktion auftreten würden.
- Tests sind nur eine Möglichkeit, mit diesen Risiken umzugehen. Konsequenzen eines Fehlers in Produktion können auf andere Art minimiert werden. Ein wichtiger Punkt ist, wie lange es dauert, bis ein Fehler in Produktion behoben ist. Je länger der Fehler in Produktion ist, desto schwerwiegender sind üblicherweise die Konsequenzen. Es gibt also einen Zusammenhang zwischen Tests und Deployment-Strategien.
- Ebenso wichtig ist, wie lange es dauert, bis ein Fehler in Produktion festgestellt wird. Diese Zeitspanne hängt davon ab, wie gut Monitoring und Logging funktionieren.

Letztendlich können durch viele Maßnahmen Fehler in Produktion angegangen werden.

Nur auf Tests zu setzen reicht nicht, um Kunden qualitativ hochwertige Software anzubieten.

Tests können mehr, als nur Risiken zu minimieren.

Tests minimieren Aufwand.

Tests können helfen, Aufwände zu minimieren oder zu vermeiden. Ein Fehler in Produktion kann großen Aufwand erzeugen. Der Fehler kann den Kundendienst beeinflussen und dort zu Aufwänden führen. Die Identifikation und Behebung von Fehlern in Produktion ist meist aufwendiger als im Test. Der Zugriff auf die Systeme ist meistens eingeschränkt. Und die Entwickler haben mittlerweile sicher andere Features implementiert, sodass sie sich in den fehlerhaften Code erst wieder einarbeiten müssen.

Auch der Ansatz für die Tests kann helfen, Aufwände zu vermeiden oder zu reduzieren. Tests zu automatisieren ist nur auf den ersten Blick aufwendig. Wenn Tests so weit definiert sind, dass die Ergebnisse reproduzierbar sind, dann ist der Schritt hin zu einer vollständigen Formalisierung und Automatisierung nicht groß. Dann sind die Kosten für die Ausführung der Tests vernachlässigbar. So kann öfter getestet werden und das kommt der Qualität zugute.

Ein Test definiert, was der Code tun soll. Dadurch stellt er eine Art Dokumentation dar. Unit-Tests definieren, wie der Produktiv-Code genutzt werden soll, und zeigt auch, wie er sich in Ausnahme- und Grenzfällen verhalten soll. Akzeptanztests spiegeln die Anforderungen der Kunden wider. Der Vorteil von Tests gegenüber Dokumentation ist, dass sie ausgeführt werden. So ist sichergestellt, dass die Tests tatsächlich das aktuelle Verhalten darstellen und nicht einen veralteten Stand oder einen Stand, der erst in Zukunft erreicht werden soll.

Testgetriebene Entwicklung macht sich zunutze, dass Tests Anforderungen sind: Bei diesem Vorgehen schreiben Entwickler zunächst Tests und dann die Implementierung. So ist sichergestellt, dass der gesamte Code mit Tests abgesichert ist. Außerdem werden Tests dann nicht durch Wissen über den Code beeinflusst, da der Code noch gar nicht existiert. Werden Tests erst im Nachhinein implementiert, können Entwickler durch das Wissen über die Implementierung bestimmte Probleme ausblenden. Bei testgetriebener Entwicklung ist das unwahrscheinlich. Tests werden so zu einer wichtigen Basis des Entwicklungsprozesses. Sie treiben die Entwicklung: Vor jeder Änderung muss es einen Test geben, der nicht funktioniert. Erst dann darf der Code angepasst werden, bis der Test erfolgreich ist. Das gilt sowohl auf der Ebene einzelner Klassen, die durch vorherige Unit-Tests abgesichert werden, wie auch auf der Ebene von Anforderungen, die durch Akzeptanztests abgesichert werden.

11.2 Wie testen?

Es gibt verschiedene Arten von Tests, die unterschiedliche Risiken handhaben:

Unit-Tests testen Bestandteile (Units) des Systems. Sie minimieren das Risiko, dass die Bestandteile Fehler enthalten. Unit-Tests testen möglichst kleine Einheiten – einzelne Methoden oder

Funktionen. Alle Abhängigkeiten müssen dazu ersetzt werden, weil sonst nicht die einzelne Unit, sondern auch die abhängigen Units getestet werden. Um die Abhängigkeiten zu ersetzen, gibt es zwei Möglichkeiten:

- *Mocks* simulieren einen bestimmten Aufruf mit einem bestimmten Ergebnis. Nach dem Aufruf kann der Test überprüfen, ob tatsächlich die erwarteten Aufrufe stattgefunden haben. Ein Test kann beispielsweise einen Mock definieren, der für eine bestimmte Kundennummer einen festen Kunden zurückgeben wird. Nach dem Test kann überprüft werden, ob der Kunde tatsächlich vom Code ausgelesen worden ist. Bei einem anderen Testfall kann der Mock einen Fehler auslösen. Dadurch können Unit-Tests Fehlersituationen simulieren, die sonst nur schwer reproduzierbar wären.
- *Stubs* hingegen simulieren den gesamten Microservice – aber mit einer begrenzten Funktionalität. Beispielsweise kann der Stub einen konstanten Wert zurückgeben. Dadurch kann ein Test ohne den eigentlichen abhängigen Microservice getestet werden. Beispielsweise kann ein Stub implementiert werden, der für bestimmte Kundennummern Testkunden zurückgibt – jeweils mit bestimmten Eigenschaften.

Unit-Tests liegen in der Verantwortung der Entwickler. Es gibt Unit-Test-Frameworks für alle gängigen Programmiersprachen. Die Tests nutzen Wissen über den Aufbau der Units. Beispielsweise ersetzen sie die Abhängigkeiten durch Mocks oder Stubs. Außerdem kann das Wissen genutzt werden, um alle Pfade bei Fallunterscheidungen in den Tests zu durchlaufen. Die Tests sind die White-Box-Tests, weil sie Wissen über den Aufbau der Units ausnutzen. Eigentlich müsste man von einer durchsichtigen Box sprechen, aber der Begriff »White Box« hat sich eingebürgert.

Der Vorteil der Unit-Tests ist die Geschwindigkeit: Die Unit-Tests auch eines komplexen Projekts können innerhalb weniger Minuten durchlaufen. So kann buchstäblich jede Code-Änderung durch Unit-Tests abgesichert werden.

Integrationstests testen die Bestandteile im **Integrationstests** Zusammenspiel. Damit sollen sie das Risiko verringern, dass in der Integration der Bestandteile Fehler enthalten sind. Sie nutzen keine Stubs oder Mocks. Die Bestandteile können als Anwendung über die UI oder mit speziellen Test-Frameworks getestet werden. Integrationstests überprüfen mindestens, ob die einzelnen Teile überhaupt miteinander kommunizieren können. Sie können die Logik beispielsweise anhand von Geschäftsprozessen testen.

Wenn sie Geschäftsprozesse testen, sind die Integrationstests den Akzeptanztests ähnlich, die Anforderungen des Kunden testen. Diesen Bereich decken Werkzeuge für BDD (Behavior Driven Design) und ATDD (Acceptance Test Driven Design) ab. Die Werkzeuge ermöglichen einen testgetriebenen Ansatz, bei dem zunächst die Tests und dann die Implementierung geschrieben wird.

Integrationstests nutzen keine Informationen über das zu testende System. Sie sind Blackbox-Tests, weil sie kein Wissen über die Interna des Systems ausnutzen.

UI-Tests testen die Anwendung durch die Oberfläche. **UI-Tests** Sie müssen eigentlich nur testen, ob die Benutzeroberfläche korrekt funktioniert. Es gibt zahlreiche Frameworks und Werkzeuge

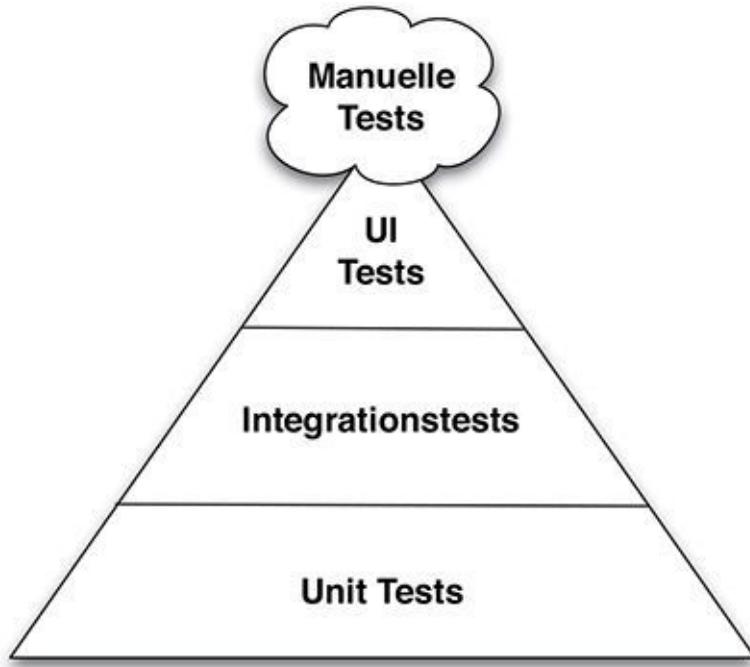
für Tests der Oberfläche. Dazu gehören Werkzeuge für Web-UIs, aber auch für Desktop- oder Mobile-Anwendungen. Die Tests sind Blackbox-Tests. Da sie die Oberfläche testen, sind die Tests fragil: Änderungen an der Oberfläche können zu Problemen führen, selbst wenn die Logik unverändert bleibt. Außerdem benötigen die Tests meistens ein vollständiges System, sodass sie langsam sind.

Schließlich kann es manuelle Tests geben. Sie können entweder das Risiko für Fehler in neuen Features minimieren oder aber bestimmte Aspekte wie Sicherheit, Performance oder fachliche Features untersuchen, bei denen es bisher Qualitätsprobleme gab. Sie sollten explorativ sein: Sie untersuchen Probleme in bestimmten Bereichen der Anwendungen. Tests, ob ein Fehler wieder auftaucht, sollten auf keinen Fall manuell sein, weil automatisierte Tests solche Fehler einfacher, kostengünstiger und reproduzierbar finden können. Manuelle Tests sind auf explorative Tests begrenzt.

Last-Tests untersuchen das Verhalten der Anwendung unter Last. Performance-Tests hingegen testen die Geschwindigkeit und Kapazitätstests, wie viele Kunden oder Anfragen das System bearbeiten kann. Diese Tests untersuchen alle die Leistungsfähigkeit der Anwendung. Dazu nutzen sie ähnliche Werkzeuge, die Antwortzeiten messen und Last generieren. Außerdem kann bei solchen Tests auch der Ressourcenverbrauch beobachtet oder überprüft werden, ob bei einer bestimmten Last Fehler auftreten. Tests, die überprüfen, ob ein System auch langfristig eine hohe Last vertragen kann, heißen auch Endurance-Tests.

Die Verteilung von Tests veranschaulicht die Testpyramide ([Abb. 11–1](#)): Die breite Basis der Pyramide zeigt, dass es viele Unit-Tests gibt. Sie können schnell ausgeführt und die meisten Fehler können auf dieser Ebene gefunden werden. Integrationstests gibt es weniger, weil sie aufwendiger sind, länger laufen und bei der Integration der Bestandteile nicht so viele Fehler auftreten können. Die reine Logik kann durch Unit-Tests abgesichert werden. UI-Tests müssen nur noch die korrekte Funktion der grafischen Benutzeroberfläche prüfen. Sie sind noch aufwendiger, weil die Automatisierung der UI kompliziert und eine vollständige Umgebung notwendig ist. Manuelle Tests sind nur vereinzelt notwendig.

Abb. 11–1 Testpyramide: das Optimum



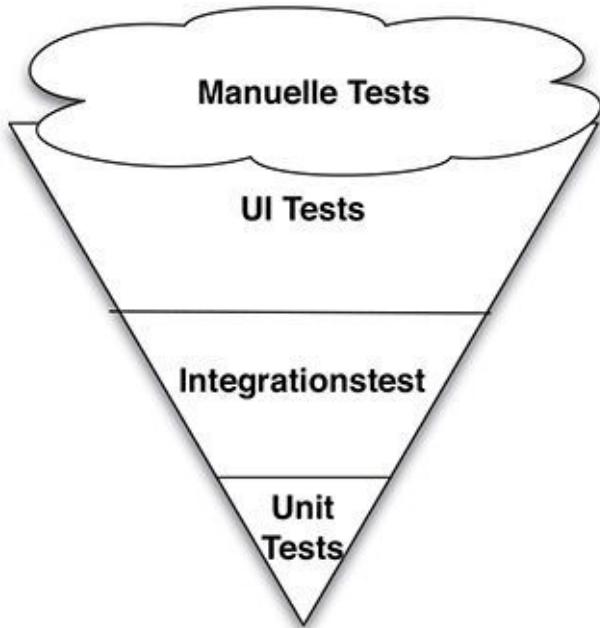
Testgetriebene Entwicklung führt fast zwangsläufig zu einer Testpyramide: Für jede Anforderung wird ein Integrationstest geschrieben und für jede Änderung an einer Klasse ein Unit-Test. Dadurch entstehen automatisch viele Integrationstests und noch mehr Unit-Tests.

Die Testpyramide erreicht eine hohe Qualität mit geringem Aufwand. Die Tests sind soweit es geht automatisiert. Jedes Risiko wird mit einem möglichst einfachen Test angegangen: Logik testen einfache und schnelle Unit-Tests. Aufwendigere Tests sind auf die Bereiche reduziert, die anders nicht getestet werden können.

Viele Projekte sind von dem Optimum der Testpyramide weit entfernt. Leider sind Tests in der Realität oft eher eine Eiswaffel ([Abb. 11–2](#)). Dann gibt es die folgenden Herausforderungen:

- Es gibt ausführliche manuelle Tests, weil solche Tests sehr einfach sind. Außerdem haben viele Tester nicht ausreichend Erfahrung mit Testautomatisierung. Insbesondere wenn die Tester keinen wartbaren Testcode schreiben können, ist es kaum möglich, Tests zu automatisieren.
- Tests über die Oberfläche sind die einfachste Form der Automatisierung, weil sie den manuellen Tests weitgehend gleichen. Wenn es automatisierte Tests gibt, dann meistens UI-Tests. Leider sind automatisierte UI-Tests fragil: Änderungen an der grafischen Benutzeroberfläche führen oft schon zu Problemen. Da die Tests das gesamte System testen, sind sie langsam. Wenn sie parallelisiert werden, kommt es oft zu Fehlschlägen, weil das System unter zu hoher Last ist.

Abb. 11–2 Test-Eiswaffel: Oft die traurige Realität



- Es gibt deutlich weniger Integrationstests. Solche Tests setzen ein umfangreiches Wissen über das System und Automatisierungstechniken voraus, was Tester oft nicht haben.
- Unit-Tests kann es durchaus mehr geben, als in der Grafik dargestellt. Oft ist aber die Qualität ebenfalls schlecht, weil die Entwickler nicht genügend Praxis mit dem Schreiben von Unit-Tests haben.

Auch werden oft unnötig komplexe Tests für bestimmte Fehlerquellen genutzt. UI-Tests oder manuelle Tests werden genutzt, um Logik zu testen. Dazu sind aber Unit-Tests ausreichend, die wesentlich schneller wären. Es gilt beim Testen, diese Probleme und die Eiswaffel zu vermeiden und stattdessen zu einer Testpyramide zu kommen.

Außerdem muss das Testkonzept auf die Risiken der Software abgestimmt sein, um an den richtigen Stellen Tests zu haben. Zum Beispiel sollte ein Projekt, das vor allem an der Performance gemessen wird, in diesem Bereich automatisierte Tests haben. Funktionale Tests sind dann vielleicht nicht ganz so wichtig.

Selber ausprobieren und experimentieren

An welchen Stellen entspricht das Vorgehen in Deinem aktuellen Projekt nicht der Testpyramide, sondern der Test-Eiswaffel?

- Wo werden manuelle Tests genutzt? Sind zumindest die wichtigsten Tests automatisiert?
- Wie ist das Verhältnis von UI zu Integrations- und Unit-Tests?
- Wie ist die Qualität der verschiedenen Tests?
- Wird testgetriebene Entwicklung genutzt? Für einzelne Klassen oder auch für Anforderungen?

Die Continuous-Delivery-Pipeline (Abb. 5–2, Abschnitt 5.1) definiert die Abfolge der verschiedenen Tests. Daher

Continuous-Delivery-Pipeline

ist sie für das Testen der Microservices interessant und nicht so sehr für das Deployment. Die Unit-Tests aus der Testpyramide werden in der Commit-Phase ausgeführt. UI-Tests können Teil der Akzeptanztests sein oder auch in der Commit-Phase ausgeführt werden. Die Kapazitätstests testen verschiedene Elemente im Zusammenspiel und können daher als Integrationstest aus der Testpyramide aufgefasst werden. Die explorativen Tests sind die manuellen Tests aus der Testpyramide.

Die Automatisierung der Tests ist bei Microservices noch wichtiger als ohnehin schon. Das Ziel von Microservices-Architekturen ist ein unabhängiges und häufiges Deployment von Software. Das lässt sich nur umsetzen, wenn die Qualität der Microservices durch Tests abgesichert ist, sonst ist ein Deployment zu risikoreich.

11.3 Risiken beim Deployment minimieren

Ein wichtiger Vorteil von Microservices ist das schnelle Deployment wegen der geringen Größe der deploybaren Einheiten. Außerdem vermeidet Resilience, dass der Ausfall eines Microservice zum Ausfall weiterer Microservices oder des gesamten Systems führt. So sinkt das Risiko, falls ein Fehler trotz der Tests in Produktion kommt.

Aber es gibt noch weitere Gründe, warum Microservices dieses Risiko minimieren:

- Das Beheben eines Fehlers geht schneller, weil nur ein Microservice neu deployt werden muss. Das ist wesentlich schneller und einfacher als das Deployment eines Deployment-Monolithen.
- Ansätze wie Blue/Green-Deployment oder Canary Releasing ([Abschnitt 12.4](#)) reduzieren das Risiko eines Deployments weiter. Ein fehlerhaftes Release eines Microservice kann mit wenig Aufwand und schnell wieder aus der Produktion genommen werden. Diese Verfahren sind mit Microservices einfacher umsetzbar, weil es weniger aufwendig ist, die benötigten Umgebungen für einen Microservice vorzuhalten als für einen ganzen Deployment-Monolithen.
- Der Service kann blind in Produktion mitlaufen. Er bekommt dann zwar dieselben Nachrichten wie die Version in Produktion, aber alle Änderungen an Daten, die der neue Service auslösen würde, werden nicht an den Daten vorgenommen, sondern nur mit den Änderungen des Service aus Produktion verglichen. Das kann beispielsweise durch Manipulationen am Datenbanktreiber oder der Datenbank selber erfolgen. Der Service kann auch eine Kopie der Datenbank nutzen. Der Microservice ändert in dieser Phase jedenfalls nie die produktiven Daten. Und auch Nachrichten, die der Microservice nach außen schickt, können mit den Nachrichten anderer Microservices verglichen werden, statt sie tatsächlich an die Empfänger zu schicken. Der Microservice läuft so schon mit allen Sonderfällen der Produktivdaten, die auch der beste Test nicht alle abdecken kann. Ebenso kann ein solches Vorgehen bessere Informationen über die Performance geben, wobei das Schreiben der Daten nicht erfolgt und so die Performance-Daten nicht ganz zuverlässig sind. Solche Verfahren sind mit Deployment-Monolithen kaum umsetzbar. Schließlich kann kaum der gesamte Deployment-Monolith blind laufen. Dazu wären viele Ressourcen notwendig und eine sehr komplexe Konfiguration, weil der Deployment-Monolith an

so vielen Stellen Änderungen an Daten vornehmen kann. Mit Microservices ist dieser Ansatz immer noch aufwendig, da er umfangreiche Unterstützung in der Software und im Deployment benötigt. Es muss eigener Code geschrieben werden, um die neue und die alte Version aufzurufen und die Änderungen und ausgehenden Nachrichten beider Versionen zu vergleichen. Aber immerhin ist der Ansatz überhaupt realisierbar.

- Schließlich kann der Service mit Monitoring genau unter die Lupe genommen werden, um Probleme möglichst schnell zu erkennen und zu beheben. Das verkürzt die Zeit, bis ein Problem auffällt und behoben werden kann. Das Monitoring nimmt bis zu einem gewissen Maße die Funktion von Akzeptanzkriterien der Tests ein. Code, der in einem Lasttest durchfällt, sollte auch beim Monitoring in Produktion ein Alarm erzeugen. Eine enge Abstimmung zwischen Monitoring und Tests ist daher sinnvoll.

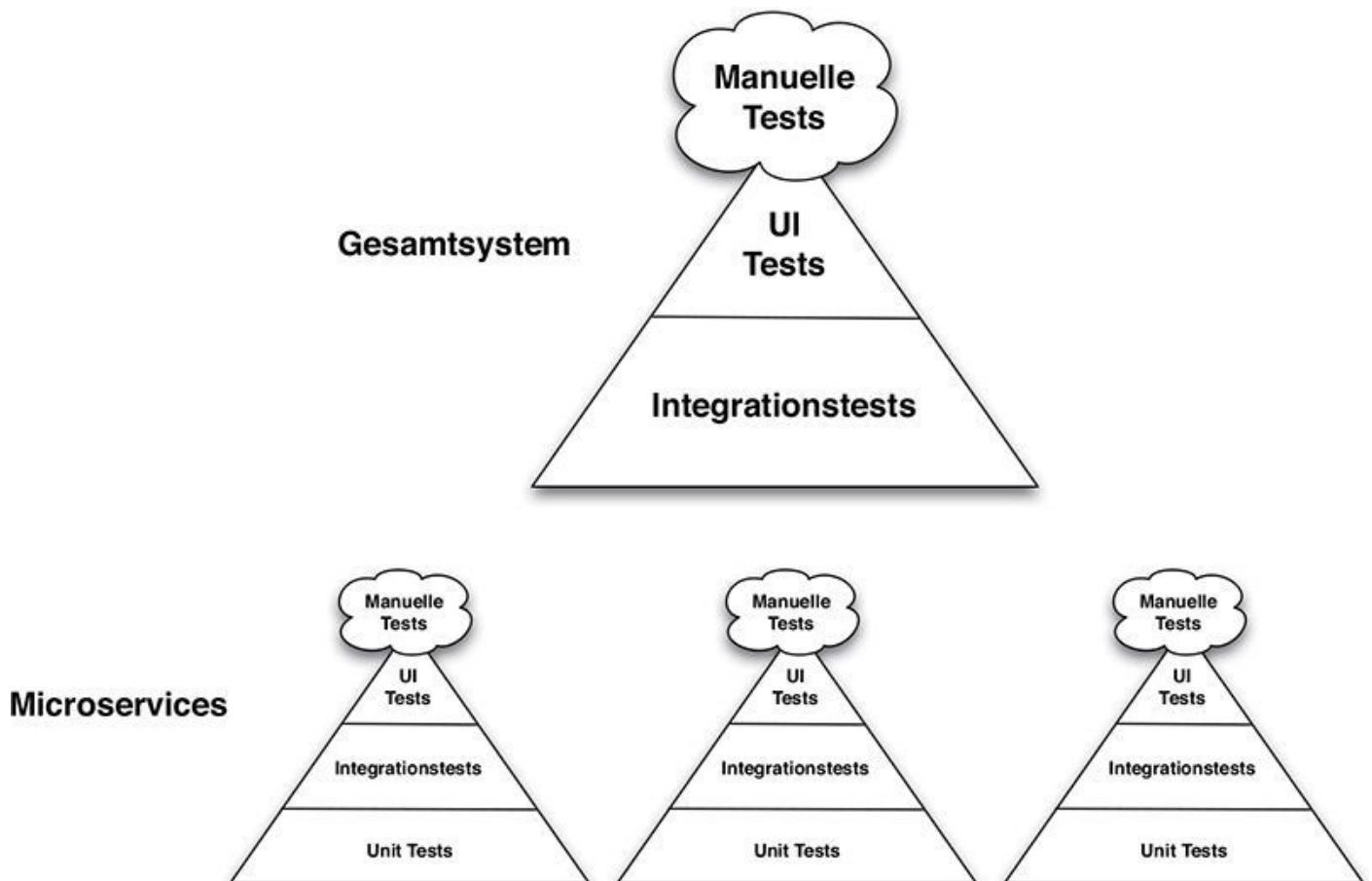
Letztendlich ist die Idee dieser Ansätze, das Risiko bei der Produktivstellung eines Microservice zu reduzieren, statt die Risiken durch Tests abzudecken. Wenn die neue Version des Microservice keine Daten ändern kann, ist ein Deployment praktisch risikolos. Das ist bei Deployment-Monolithen kaum möglich, weil der Deployment-Prozess eines Monolithen viel aufwendiger ist, länger dauert und mehr Ressourcen benötigt. Daher kann das Deployment nicht schnell durchgeführt werden. So kann das Deployment bei Fehlern nicht einfach wieder rückgängig gemacht werden.

Das Vorgehen ist auch interessant, weil einige Risiken sich durch Tests kaum beseitigen lassen. So können Last- und Performance-Tests zwar ein Indikator für das Verhalten der Anwendung in Produktion sein. Aber völlig zuverlässig sind diese Tests nicht, weil die Datenmengen in Produktion anders sind, das Benutzerverhalten sich unterscheidet und die Hardware anders dimensioniert ist. Alle diese Aspekte in einer Teststellung abzudecken, ist unrealistisch. Ebenso kann es fachliche Fehler geben, die nur mit Datenkonstellationen aus der Produktion auftreten. Sie lassen sich durch Tests kaum nachstellen. Monitoring und schnelles Deployment kann in Microservices-Umgebungen durchaus eine Alternative zu Tests sein. Wichtig ist es abzuwegen, welches Risiko mit welcher Maßnahme reduziert werden kann.

11.4 Tests des Gesamtsystems

Bei Microservices gibt es neben den Tests der einzelnen Microservices auch einen Test des Gesamtsystems. Es gibt die Testpyramide mehrfach: einmal für jeden einzelnen Microservice und dann noch für das Gesamtsystem. Für das Gesamtsystem gibt es Integrationstests der Microservices, UI-Tests der gesamten Anwendung und manuelle Tests. Unit-Tests auf dieser Ebene sind die Tests der Microservices, da sie die Units des Gesamtsystems sind. Diese Tests bestehen aus einer vollständigen Testpyramide der einzelnen Microservices.

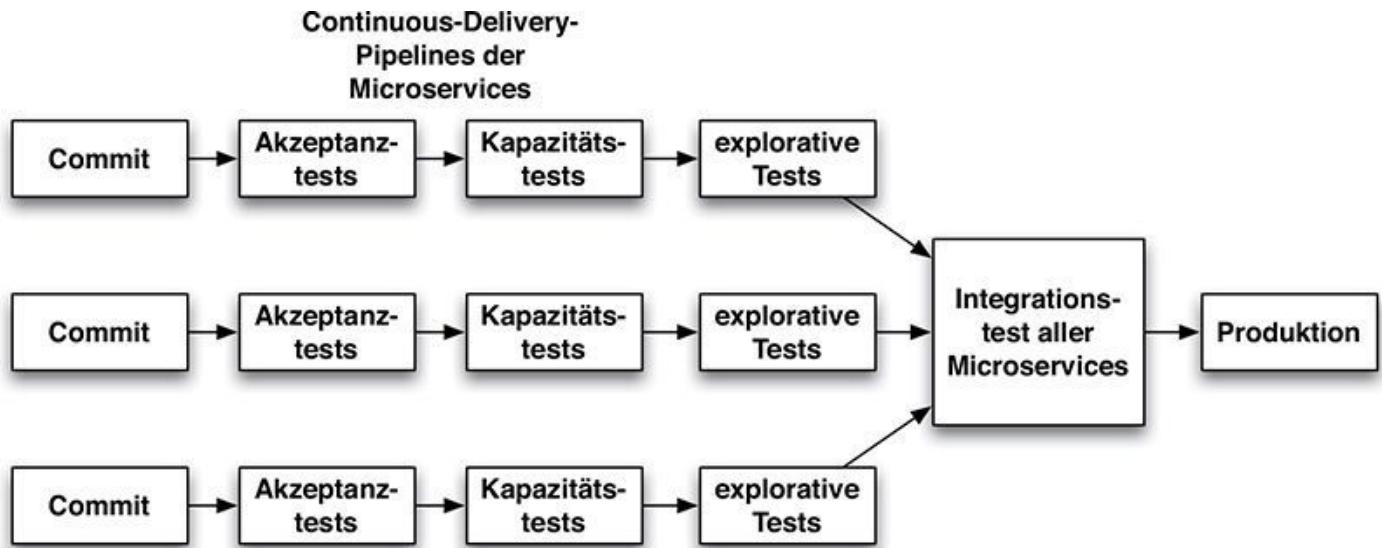
Abb. 11–3 Testpyramide für Microservices



Die Tests des Gesamtsystems sind dafür zuständig, Probleme im Zusammenspiel der Microservices zu identifizieren. Microservices sind verteilte Systeme. Aufrufe können die Zusammenarbeit mehrerer Microservices benötigen, um ein Ergebnis an den Nutzer zurückzugeben. Das ist für das Testen eine Herausforderung: Verteilte Systeme haben viel mehr Fehlerquellen. Tests des Gesamtsystems müssen diesen Risiken begegnen. Beim Testen von Microservices wird aber ein anderer Ansatz gewählt: Die einzelnen Microservices sollten dank Resilience auch bei Problemen mit anderen Microservices funktionieren. Funktionale Tests können mit Stubs oder Mocks der anderen Microservices erfolgen. So können Microservices getestet werden, ohne dass dazu ein komplexes verteiltes System aufgebaut und auf alle Fehlerszenarien überprüft werden muss.

Dennoch sollte jeder Microservice vor dem Deployment in Produktion in der Integration mit den anderen Microservices getestet werden. Das erzwingt Änderungen an der Continuous-Delivery-Pipeline, wie sie in [Abschnitt 5.1](#) beschrieben worden ist: Jeder Microservice sollte am Ende der Deployment-Pipeline zusammen mit den anderen Microservices getestet werden. Jeder Microservice sollte diese Stufe einzeln durchlaufen. Wenn neue Versionen mehrerer Microservices gemeinsam in dieser Stufe getestet werden, ist nicht mehr klar, welcher Microservice für einen Fehlschlag der Tests verantwortlich ist. Es ist denkbar, mehrere Microservices gleichzeitig in diese Phase zu schicken, wenn bei einem Fehler klar ist, welcher Microservice ihn verursacht. In der Praxis sind solche Optimierungen aber kaum zu beherrschen.

Abb. 11–4 Integrationstests am Ende der Deployment-Pipelines



Es ergibt sich das Bild aus [Abbildung 11–4](#). Die Continuous-Delivery-Pipelines der Microservices laufen am Ende in einen gemeinsamen Integrationstest, in den jeder Microservice getrennt eintreten muss. Wenn ein Microservice gerade durch den Integrationstest läuft, müssen die anderen Microservices warten, bis der Integrationstest beendet ist. Um sicherzustellen, dass tatsächlich nur ein Service zur Zeit durch die Integrationstests läuft, können die Tests auf einer eigenen Umgebung ausgeführt werden. Dann darf nur ein Microservice in dieser Umgebung zu einem Zeitpunkt in einer neuen Version ausgeliefert werden. Die Umgebung erzwingt eine serielle Abarbeitung der Integrationstests der Microservices.

Ein solcher Synchronisationspunkt verlangsamt das Deployment der Microservices und so den gesamten Prozess. Wenn der Integrationstest beispielsweise eine Stunde dauert, kann an einem Arbeitstag mit acht Stunden nur acht Mal ein Microservice deployt werden. Wenn es in dem Projekt acht Teams gibt, kann jedes Team genau einmal pro Tag deployen. Für schnelle Fehlerbehebung in Produktion ist das ungenügend. Außerdem wird so ein wesentliches Ziel von Microservices nicht erreicht: Jeder Microservice soll unabhängig deployt werden können. Das kann er zwar noch – aber das Deployment dauert zu lange und durch die Integrationstests haben die Microservices Abhängigkeiten untereinander – zwar nicht auf Code-Ebene, aber in den Deployment-Pipelines. Ebenso ist es nicht ausgewogen, wenn die Continuous Delivery ohne den letzten Integrationstest zum Beispiel nur eine Stunde dauert, aber dennoch nur ein Release in Produktion pro Tag möglich ist.

Dieses Problem kann die Testpyramide lösen. Sie verschiebt den Fokus von Integrationstests des Gesamtsystems zu Integrationstests der einzelnen Microservices und Unit-Tests. Wenn es wenige Integrationstests des Gesamtsystems gibt, dauern die Integrationstests auch nicht so lange, weniger Synchronisation ist notwendig und das Deployment in Produktion ist schneller. Die Integrationstests sollen nur das Zusammenspiel der Microservices testen. Es ist ausreichend, wenn die Microservices sich wechselseitig erreichen können. Alle anderen Risiken können schon vorher abgedeckt werden. Mit Consumer-Driven Contract Tests ([Abschnitt 11.7](#)) können sogar Fehler bei der Kommunikation zwischen den Microservices ausgeschlossen werden, ohne dass die Microservices zusammen getestet werden. Alle diese Maßnahmen tragen dazu bei, die

Integrationstests des Gesamtsystems vermeiden

Anzahl der Integrationstests und damit deren Dauer zu reduzieren. Es wird nicht weniger getestet – sondern nur an anderen Stellen: in den Tests der einzelnen Microservices und in den Unit-Tests.

Die Tests für das Gesamtsystem können von allen Teams gemeinsam weiterentwickelt werden. Sie fallen damit in die teamübergreifende Makro-Architektur, weil sie das Gesamtsystem betreffen und aus dem Grund auch nicht in der Zuständigkeit eines einzelnen Teams stehen können (siehe [Abschnitt 13.3](#)).

Das Gesamtsystem kann auch manuell getestet werden. Es ist aber unrealistisch, dass jede neue Version eines Microservice erst nach einem manuellen Test im Zusammenspiel mit den anderen Microservices in Produktion geht. Die Verzögerungen sind einfach zu groß. Die manuellen Tests des Gesamtsystems können beispielsweise Features betrachten, die in Produktion noch gar nicht aktiviert sind. Oder es können bestimmte Aspekte wie Sicherheit getestet werden, wenn in diesen Bereichen zuvor Probleme festgestellt worden sind.

11.5 Legacy-Anwendungen mit Microservices testen

Microservices werden oft genutzt, um Legacy-Anwendungen abzulösen. Die Legacy-Anwendungen sind meistens Deployment-Monolithen. Daher testet die Continuous-Delivery-Pipeline der Legacy-Anwendung viele Funktionalitäten, die auf Microservices verteilt werden müssen. Wegen der vielen Funktionalitäten dauert ein Durchlauf durch die verschiedenen Teststufen der Continuous-Delivery-Pipeline bei einem Deployment-Monolithen sehr lange. Dementsprechend ist das Deployment in Produktion ebenfalls komplex und dauert lange. Unter diesen Bedingungen ist es unrealistisch, dass jede Änderung an der Legacy-Anwendung in Produktion geht. Oft gibt es Deployments am Ende eines Sprints von 14 Tagen oder gar ein Release pro Quartal. Nächtliche Tests fühlen dem aktuellen Stand des Systems auf den Zahn. Tests können aus der Continuous-Delivery-Pipeline in die nächtlichen Tests verschoben werden. Die Continuous-Delivery-Pipeline ist dann schneller, aber bestimmte Fehler werden erst im nächtlichen Test erkannt. Dann ist die Frage, welche der Änderungen des letzten Tages für den Fehler verantwortlich ist.

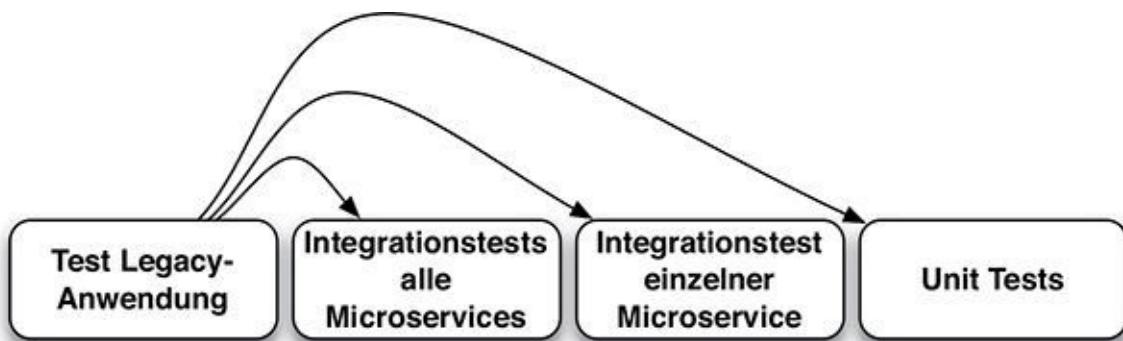
Bei der Migration von einer Legacy-Anwendung hin zu Microservices sind Tests besonders wichtig. Werden einfach nur die Tests der Legacy-Anwendung weiter genutzt, testen sie einige Funktionalitäten, die mittlerweile in den Microservices beheimatet sind. Dann müssten diese Tests bei jedem Release jedes Microservice ausgeführt werden – was viel zu lange dauert. Die Tests müssen verlagert werden. Sie können zu Integrationstests für die Microservices werden ([Abb. 11–5](#)). Aber die Integrationstests der Microservices sollen schnell durchlaufen. Es ist nicht notwendig, in dieser Phase Tests für Funktionalitäten zu nutzen, die in einem Microservice liegen. Dann müssen die Tests der Legacy-Anwendung zu Integrationstests der einzelnen Microservices werden oder sogar zu Unit-Tests. Sie sind dann viel schneller. Und sie laufen als Test für einen Microservice, sodass sie nicht die gemeinsamen Tests der Microservices verlangsamen.

Tests der Legacy-Anwendung verlagern

Nicht nur die Legacy-Anwendung muss migriert werden, sondern auch die Tests. Sonst sind schnelle Deployments trotz Migration der Legacy-Anwendung nicht möglich.

Die Tests für die Funktionalitäten, die in die Microservices übertragen worden sind, können aus den Tests der Legacy-Anwendung entfernt werden. So wird das Deployment der Legacy-Anwendung schrittweise schneller und einfacher. Dadurch werden auch Änderungen an der Legacy-Anwendung zunehmend einfacher.

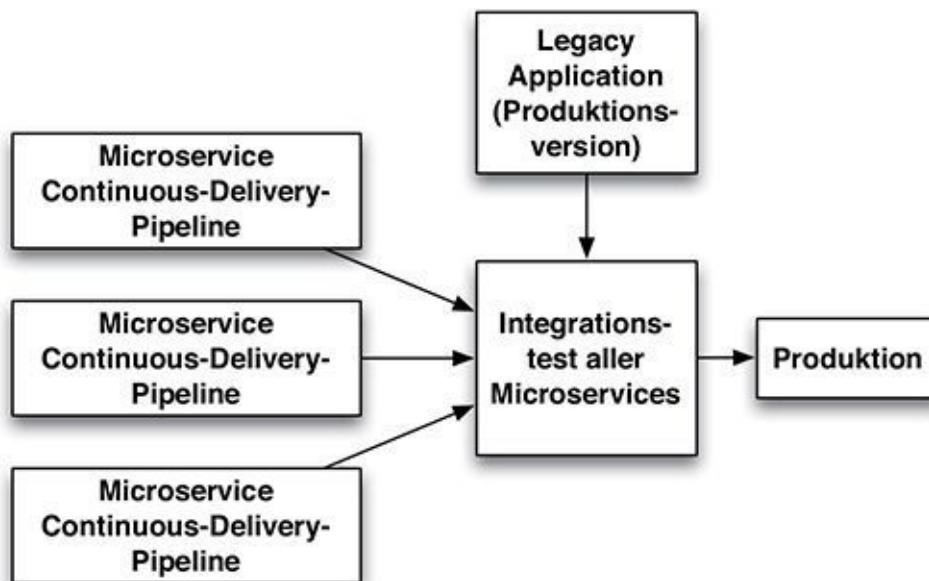
Abb. 11–5 Verlagern der Tests



Die Legacy-Anwendung muss auch zusammen mit den Microservices getestet werden. Die Microservices müssen mit der Version der Legacy-Anwendung aus der Produktion getestet werden. Das sichert ab, dass die Microservices auch in Produktion mit der Legacy-Anwendung funktionieren. Dazu kann in die Integrationstests der Microservices die Version der Legacy-Anwendung aus der Produktion integriert werden. Es ist die Verantwortung jedes Microservice, mit dieser Version fehlerfrei durch den Test zu kommen (Abb. 11–6).

Integrationstest Legacy-Anwendung und Microservices

Abb. 11–6 Legacy-Anwendung in der Continuous-Delivery-Pipeline der Microservices

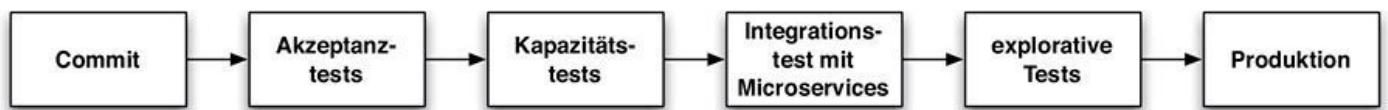


Wenn die Deployment-Zyklen der Legacy-Anwendung Tage oder Woche betragen, ist parallel eine Version der Legacy-Anwendung in Entwicklung. Die Microservices müssen auch mit dieser Version getestet werden. Das stellt sicher, dass beim Release der neuen Legacy-Anwendung nicht plötzlich Fehler auftreten. Die aktuell in Entwicklung befindliche Version der Legacy-Anwendung testet als Teil der eigenen Deployment-Pipeline gegen die aktuellen Microservices (Abb. 11–7). Dazu muss die Version der

Microservices genutzt werden, die in Produktion ist.

Die Versionen der Microservices ändern sich wesentlich häufiger als die Legacy-Anwendung. Eine neue Version eines Microservice kann die Continuous-Delivery-Pipeline der Legacy-Anwendung brechen. Das Team der Legacy-Anwendung kann diese Probleme nicht beheben, da es den Code der Microservices nicht kennt. Der Microservice ist allerdings gegebenenfalls schon mit dieser Version in Produktion. Eine neue Version des Microservice muss dann ausgeliefert werden, um den Fehler zu beheben – obwohl die Continuous-Delivery-Pipeline des Microservice erfolgreich durchgelaufen ist.

Abb. 11–7 Microservices in der Continuous-Delivery-Pipeline der Legacy-Anwendung



Die Alternative wäre, die Microservices auch durch einen Integrationstest mit der Entwicklungsversion der Legacy-Anwendung zu schicken. Das verlängert jedoch den übergreifenden Integrationstest der Microservices und macht dadurch die Entwicklung der Microservices komplexer.

Das Problem kann durch Consumer-Driven Contract Tests ([Abschnitt 11.7](#)) angegangen werden. Die Erwartungen der Legacy-Anwendung an die Microservices und der Microservices an die Legacy-Anwendung können mit Consumer-Driven Contract Tests definiert werden, sodass die Integrationstests auf ein Mindestmaß reduziert werden können.

Ebenso kann die Legacy-Anwendung mit einem Stub der Microservices getestet werden. Diese Tests sind keine Integrationstests, weil sie nur die Legacy-Anwendung testen. So wird die Anzahl der übergreifenden Integrationstests reduziert. Dieses Konzept zeigt [Abschnitt 11.6](#) am Beispiel der Tests von Microservices. Das bedeutet aber, dass die Tests der Legacy-Anwendung angepasst werden müssen.

11.6 Tests einzelner Microservices

Tests der einzelnen Microservices sind eine Aufgabe des jeweiligen Teams, das für einen Microservice zuständig ist. Das Team muss als Teil der eigenen Continuous-Delivery-Pipeline die verschiedenen Tests wie Unit-Tests, Last Tests und Akzeptanztest umsetzen – wie das auch für andere Systeme, die keine Microservices sind, der Fall wäre.

Allerdings benötigen Microservices für einige Funktionalitäten Zugriff auf andere Microservices. Für die Tests stellt das eine Herausforderung dar: Es ist nicht sinnvoll, für jeden Test jedes Microservice eine komplette Umgebung vorzuhalten. Zum einen würde das zu viele Ressourcen verbrauchen. Außerdem ist es schwierig, alle diese Umgebungen mit der aktuellen Software zu versorgen. Technisch können leichtgewichtige Virtualisierungsansätze wie Docker zumindest den Ressourcenaufwand reduzieren. Aber bei 50 oder 100 Microservices wird auch dieser Ansatz nicht mehr ausreichend sein.

Eine Lösung ist eine Referenzumgebung, in der die Microservices in der aktuellen Version vorgehalten

Referenzumgebung

werden. Die Tests der verschiedenen Microservices können die restlichen Microservices aus dieser Umgebung nutzen. Allerdings kann es zu Fehlern kommen, wenn mehrere Teams unterschiedliche Microservices parallel mithilfe der Microservices aus der Referenzumgebung testen. Die Tests können sich gegenseitig beeinflussen und so Fehler erzeugen. Außerdem muss die Referenzumgebung verfügbar sein. Wenn durch einen Test ein Teil der Umgebung ausfällt, sind im Extremfall für alle Teams Tests unmöglich. Die Microservices müssen in der aktuellen Version in der Referenzumgebung vorgehalten werden. Das erzeugt weitere Aufwände. Eine Referenzumgebung ist keine gute Lösung für das isolierte Testen von Microservices.

Eine andere Möglichkeit ist die Simulation des *Stubs* benutzten Microservice. Für die Simulation von Bestandteilen eines Systems für Tests gibt es zwei unterschiedliche Möglichkeiten, wie [Abschnitt 11.2](#): dargestellt hat, nämlich Stubs und Mocks. Stubs sind die bessere Wahl für das Ersetzen von Microservices. Sie können verschiedene Testszenarien unterstützen. So kann das Implementieren eines einzigen Stubs die Entwicklung aller abhängigen Microservices unterstützen.

Werden Stubs verwendet, muss ein Team zu einem Microservice auch einen Stub liefern. So ist gewährleistet, dass die Microservices und die Stubs sich tatsächlich weitgehend identisch verhalten. Wenn Consumer-driven Contract Tests die Stubs (siehe [Abschnitt 11.7](#)) auch testen, sichert das die korrekte Simulation der Microservices durch die Stubs ab.

Die Stubs sollten mit einer einheitlichen Technologie implementiert werden. Alle Teams, die einen Microservice nutzen, müssen auch den Stub für Tests nutzen. Eine einheitliche Technologie macht die Handhabung der Stubs einfacher. Sonst muss ein Team, das mehrere Microservices nutzt, für Tests eine Vielzahl von Technologien beherrschen.

Stubs könnten mit denselben Technologien wie die eigentlichen Microservices implementiert werden. Allerdings sollen die Stubs weniger Ressourcen verbrauchen als die Microservices. Daher ist es besser, wenn Stubs einen einfacheren Technologie-Stack nutzen. Das Beispiel in [Abschnitt 14.13](#) nutzt für die Stubs dieselbe Technologie wie die eigentlichen Microservices. Aber die Stubs liefern nur konstante Werte und laufen im selben Prozess wie die Microservices, die den Stub nutzen. Dadurch verbrauchen die Stubs weniger Ressourcen.

Es gibt Technologien, die auf die Implementierung von Stubs spezialisiert sind. Werkzeuge für Consumer-driven Contract Tests können oft auch Stubs erzeugen (siehe [Abschnitt 11.7](#)).

- *mountebank* [5] ist in JavaScript mit Node.js geschrieben. Es kann Stubs für TCP, HTTP, HTTPS und SMTP bereitstellen. Neue Stubs können zur Laufzeit erzeugt werden. Die Definition des Stubs erfolgt in einer JSON-Datei. Sie definiert, unter welchen Bedingungen welche Antworten von dem Stub zurückgeliefert werden sollen. Eine Erweiterung mit JavaScript ist ebenfalls möglich. mountebank kann auch als Proxy fungieren. Dann schickt es Anfragen an einen Service weiter – oder es wird nur die erste Anfrage weitergeschickt und die Antwort wird aufgezeichnet. Alle

folgenden Anfragen beantwortet mountebank dann mit der aufgezeichneten Antwort. Neben Stubs unterstützt mountebank auf Mocks.

- *WireMock* [6] ist in Java geschrieben und steht unter der Apache 2-Lizenz. Das Framework erlaubt es sehr einfach, für bestimmte Anfragen bestimmte Daten zurückzuliefern. Das Verhalten legt Java-Code fest. WireMock unterstützt HTTP und HTTPS. Der Stub kann in einem eigenen Prozess laufen, in einem Servlet-Container oder direkt im JUnit-Test.
- *Moco* [7] ist ebenfalls in Java geschrieben und steht unter der MIT-Lizenz. Das Verhalten des Stubs kann mit Java-Code oder einer JSON-Datei ausgedrückt werden. Es unterstützt HTTP, HTTPS und einfache Socket-Protokolle. Die Stubs können in einem Java-Programm gestartet werden oder in einem eigenständigen Server.
- *stubby4j* [8] ist in Java geschrieben und steht unter der MIT-Lizenz. Es nutzt zur Definition des Verhaltens eine YAML-Datei. Neben HTTP wird HTTPS als Protokoll unterstützt. Die Definition der Daten erfolgt in YAML oder JSON. Es kann auch eine Interaktion mit einem Server aufgenommen oder das Verhalten des Stubs mit Java programmiert werden. Aus dem Request können Informationen in den Response kopiert werden.

Selber ausprobieren und experimentieren

Nutze das Beispiel aus [Kapitel 14](#) und ergänze Stubs mit einem Stub-Framework deiner Wahl. Die Beispielanwendung hat die Konfiguration application-test.properties. Dort ist definiert, welcher Stub für die Tests genutzt wird.

11.7 Consumer-Driven Contract Test

Jede Schnittstelle einer Komponente ist letztendlich ein Vertrag: Der Aufrufer erwartet, dass bestimmte Seiteneffekte ausgelöst werden oder Werte zurückkommen, wenn er die Schnittstelle nutzt. Dieser Vertrag ist üblicherweise nicht formal definiert. Wenn ein Microservice gegen die Erwartungen verstößt, äußert sich das in einem Fehler, der entweder in Produktion oder im Integrationstest auffällt. Wenn es gelingt, diesen Vertrag explizit zu machen und einzeln zu testen, können die Integrationstests vom Testen des Vertrags entlastet werden, ohne ein höheres Risiko für Fehler in Produktion einzugehen. Außerdem würden die Microservices einfacher änderbar, weil einfach zu überprüfen wäre, welche Änderungen zu Problemen bei der Nutzung durch andere Microservices führen.

Oft werden Änderungen an Systembestandteilen nicht vorgenommen, weil die genaue Nutzung des Systembestandteils unklar ist. Es besteht das Risiko eines Fehlers im Zusammenspiel mit anderen Microservices und es wird befürchtet, dass der Fehler zu spät auffällt. Wenn es klar ist, wie ein Microservice genutzt wird, sind Änderungen viel leichter durchzuführen und abzusichern.

Zu dem Vertrag eines Microservice gehören [1]:

Bestandteile des Vertrags

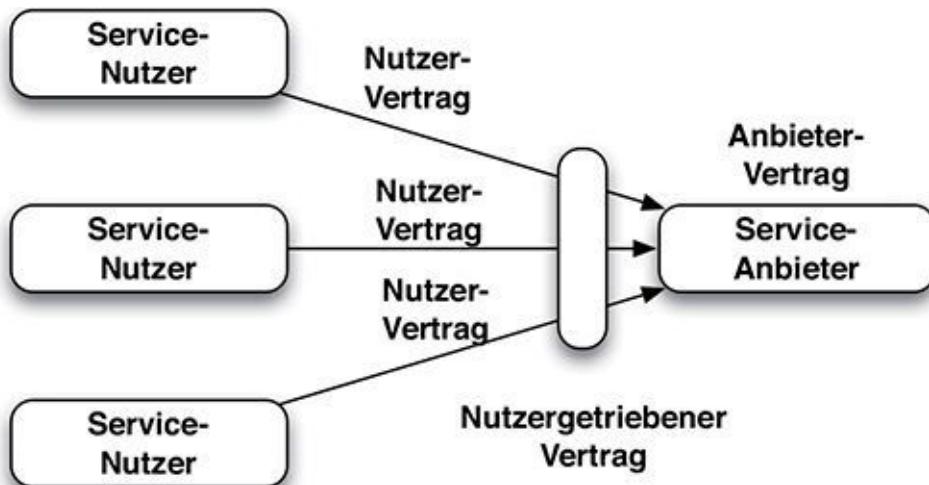
- Die *Datenformate* definieren, in welchem Format die Informationen von den anderen Microservices erwartet und wie sie an einen Microservice übergeben werden.

- Die *Schnittstelle* legt fest, welche Operationen zur Verfügung stehen.
- *Abläufe* oder *Protokolle* definieren, welche Operationen in welcher Abfolge mit welchen Ergebnissen durchgeführt werden können.
- Schließlich gibt es *Meta-Informationen* bei den Aufrufen, die beispielsweise eine Benutzeroauthentifizierung umfassen können.
- Und es kann bestimmte *nichtfunktionale Aspekte* geben wie die Latenzzeit oder einen bestimmten Durchsatz.

Zwischen dem Konsumenten und dem Anbieter eines Verträge Service gibt es verschiedene Verträge:

- Der *Provider Contract* (Anbieter-Vertrag) umfasst alles, was der Service-Anbieter zur Verfügung stellt. Pro Service-Anbieter gibt es einen solchen Vertrag. Er definiert den gesamten Service vollständig. Er kann sich beispielsweise mit der Version des Service (siehe [Abschnitt 9.6](#)) ändern.
- Der *Consumer Contract* (Nutzer-Vertrag) umfasst die Funktionalitäten, die ein Service-Nutzer tatsächlich verwendet. Pro Service gibt es mehrere solche Verträge – mit jedem Nutzer mindestens einen. Er umfasst nur die Teile des Service, die der Nutzer auch tatsächlich verwendet. Er kann sich durch Modifikationen am Service-Nutzer ändern.
- Der *Consumer-driven Contract, CDC* (nutzergetriebene Vertrag) umfasst alle Nutzer-Verträge. Daher enthält er alle Funktionalitäten, die irgendein Service-Nutzer verwendet. Es gibt nur einen solchen Vertrag pro Service. Da er von den Nutzer-Verträgen abhängt, kann er sich ändern, wenn neue Aufrufe vom Service-Nutzer an den Service-Anbieter gehen oder es neue Anforderungen an die Aufrufe gibt.

Abb. 11–8 Unterschiedliche Verträge zwischen Nutzer und Anbieter



Durch den Consumer-driven Contract wird offensichtlich, welche Bestandteile des Provider Contracts tatsächlich genutzt werden. Dadurch ist dann auch klar, wo der Microservice seine Schnittstelle ändern kann und welche Bestandteile des Microservice nicht genutzt werden.

Idealerweise wird aus dem Consumer-driven Contract Implementierung ein Consumer-driven Contract Test, den der Service-

Anbieter ausführen kann. Es muss für die Service-Nutzer möglich sein, diese Tests zu ändern. Sie können entweder zusammen mit dem Service-Anbieter in der Versionskontrolle abgelegt werden. Dann müssen die Service-Nutzer Zugriff auf die Versionskontrolle des Service-Anbieters bekommen. Gegebenenfalls können die Tests auch in der Versionskontrolle der Service-Nutzer abgelegt werden. Dann muss der Service-Anbieter die Tests aus der Versionskontrolle holen und mit jeder Version der Software ausführen. Dann ist allerdings keine Versionierung der Tests zusammen mit der getesteten Software möglich, da Test und getestete Software in zwei getrennten Projekten in der Versionskontrolle abgelegt sind.

Die Gesamtheit aller Tests stellt den Consumer-driven Contract dar. Die Bestandteile jedes Teams entsprechen jeweils den Consumer Contracts. Die Consumer-driven Contract Tests können als Teil der Tests des Microservice ausgeführt werden. Sind sie erfolgreich, so sollten alle Nutzer mit dem Microservice erfolgreich zusammenarbeiten können. Durch den Test wird verhindert, dass erst beim Integrationstest Fehler auffallen. Außerdem wird die Änderung der Microservices einfacher, weil Anforderungen an die Schnittstellen bekannt sind und ohne besonderen Aufwand getestet werden können. Das Risiko bei einer Änderung, die Auswirkungen auf die Schnittstelle hat, ist dadurch wesentlich geringer, weil Probleme vor den Integrationstests oder der Produktion auffallen.

Um die Consumer-driven Contract Tests zu schreiben, Werkzeuge muss eine Technologie definiert werden. Die Technologie sollte einheitlich für alle Projekte sein, denn ein Microservice kann mehrere andere Microservices nutzen. Dann muss ein Team für verschiedene andere Microservices Tests schreiben. Das ist einfacher, wenn es eine einheitliche Technologie gibt. Sonst müssen die Teams viele unterschiedliche Technologien kennen. Die Technologie für die Tests kann durchaus eine andere sein als die Technologie, die für die Implementierung genutzt wird.

- Ein *beliebiges Testframework* ist eine Möglichkeit, die Consumer-driven Contract Tests zu implementieren. Für Lasttests können weitere Werkzeuge definiert sein. Neben den fachlichen Anforderungen können auch Anforderungen an das Lastverhalten existieren. Allerdings muss klar definiert sein, wie dem Test der Microservice zur Verfügung gestellt wird. Beispielsweise kann er unter einem bestimmten Port auf der Testmaschine verfügbar sein. So kann der Test über die Schnittstelle des Microservice erfolgen, über die auch der Zugriff anderer Microservices stattfindet.
- In der Beispielanwendung ([Abschnitt 14.13](#)) werden einfache *JUnit*-Tests genutzt, um den Microservice zu testen und dabei festzustellen, ob die benötigten Funktionalitäten unterstützt werden. Wenn inkompatible Änderungen an den Datenformaten vorgenommen werden oder die Schnittstelle sich anderweitig ändert, schlagen die Tests fehl.
- Es gibt Werkzeuge, die speziell für die Implementierung von Consumer-driven Contract Tests gedacht sind. Ein Beispiel ist *Pacto* [2]. Es ist in Ruby geschrieben und steht unter der MIT-Lizenz. Pacto unterstützt REST/HTTP und ergänzt solche Schnittstellen um einen Vertrag. Pacto kann in einen Testaufbau integriert werden. Dann gleicht Pacto die Header gegen erwartete Werte ab und die JSON-Datenstrukturen im Body gegen JSON-Schemata ab. Diese Informationen stellen den

Vertrag dar. Der Vertrag kann auch aus einer aufgenommenen Interaktion zwischen einem Client und einem Server generiert werden. Mit dem Vertrag kann Pacto die Anfragen und Antworten eines Systems validieren. Ebenso kann Pacto mit diesen Informationen einen einfachen Stub bauen. Pacto kann zusammen mit RSpec auch Systeme testen, die in anderen Sprachen geschrieben worden sind. Ohne RSpec bietet Pacto die Möglichkeit, einen Server laufen lassen. Es ist so möglich, Pacto auch außerhalb eines Ruby-Systems zu nutzen.

- *Pact* [3] ist ebenfalls in Ruby geschrieben und steht auch unter der MIT-Lizenz. Der Service-Nutzer kann mit Pact einen Stub für den Service schreiben und dann die Interaktion mit dem Stub aufnehmen. Daraus entsteht eine Pact-Datei, die den Vertrag festhält. Sie kann auch genutzt werden, um zu testen, ob der eigentliche Service den Vertrag korrekt umsetzt. Pact ist vor allem für Ruby nützlich, aber pact-jvm [4] unterstützt einen ähnlichen Ansatz für verschiedene JVM-Sprachen wie Scala, Java, Groovy oder Clojure.

Selber ausprobieren und experimentieren

Nutze das Beispiel aus [Kapitel 14](#) und ergänze Consumer-driven Contracts mit einem Framework deiner Wahl. Die Beispielanwendung hat die Konfiguration `application-test.properties`. Dort ist definiert, welcher Stub für die Tests genutzt wird. Überprüfe auch die Contracts in der Produktionsumgebung.

11.8 Technische Standards testen

Microservices müssen bestimmten technischen Anforderungen genügen. So sollten die Microservices auch beim Ausfall anderer Microservices noch funktionieren und sich in der Service Discovery registrieren. Diese Eigenschaften können ebenfalls durch Tests überprüft werden. Das hat einige Vorteile:

- Die Vorschriften sind durch die Tests eindeutig definiert. Es gibt keine Diskussionen, wie die Vorschriften genau gemeint sind.
- Sie können automatisch getestet werden. Dadurch ist jederzeit klar, ob ein Microservice die Regeln erfüllt oder nicht.
- Neue Teams können neue Komponenten darauf überprüfen, ob die Regeln eingehalten werden.
- Wenn ein Microservice nicht den sonst üblichen Technologie-Stack nutzt, kann trotzdem abgesichert werden, dass die Microservices sich technisch korrekt verhalten.

Zu den möglichen Tests zählen beispielsweise:

- Die Microservices müssen sich für die Service Discovery registrieren ([Abschnitt 8.9](#)). Der Test kann überprüfen, ob die Komponente sich beim Hochfahren in der Service Registry registriert.
- Außerdem müssen die gemeinsamen Mechanismen für die Konfiguration und

Koordination genutzt werden ([Abschnitt 8.8](#)). Der Test kann überprüfen, ob bestimmte Werte aus der zentralen Konfiguration ausgelesen werden. Dazu kann eine eigene Testschnittstelle implementiert werden.

- Eine gemeinsame Sicherheitsinfrastruktur kann getestet werden, indem das Nutzen des Microservice mit einem bestimmten Token getestet wird ([Abschnitt 8.12](#)).
- Bei Dokumentation und Metadaten ([Abschnitt 8.13](#)) kann darauf getestet werden, ob der Test auf dem definierten Weg auf die Dokumentation zugreifen kann.
- Für Monitoring ([Abschnitt 12.3](#)) und Logging ([Abschnitt 12.2](#)) kann überprüft werden, ob der Microservice beim Starten die entsprechenden Schnittstellen bedient und Werte bzw. Log-Einträge liefert.
- Beim Deployment ([Abschnitt 12.4](#)) ist es ausreichend, den Microservice auf einem Server zu deployen und zu starten. Wenn dabei der definierte Standard genutzt wird, ist dieser Aspekt ebenfalls richtig umgesetzt.
- Als Test für die Steuerung ([Abschnitt 12.5](#)) kann der Microservice einfach neu gestartet werden.
- Für Resilience ([Abschnitt 10.5](#)) kann im einfachsten Fall getestet werden, ob der Microservice auch ohne abhängige Microservices zumindest hochfährt und im Monitoring Fehler zeigt. Die korrekte Funktion des Microservice bei Verfügbarkeit der anderen Microservices wird durch die funktionalen Tests abgesichert. Ein solcher Fall, bei dem der Microservice keinen anderen Service erreichen kann, fehlt hingegen in normalen Tests.

Im einfachsten Fall kann der technische Test den Microservice deployen und starten. Das testet Deployment und Steuerung. Es müssen keine abhängigen Microservices vorhanden sein. Das Starten des Microservice sollte auch ohne abhängige Microservices wegen Resilience möglich sein. Dann können Logging und Monitoring untersucht werden, die auch in dieser Situation funktionieren und Fehler enthalten sollten. Schließlich kann die Integration in die gemeinsamen technischen Dienste wie Service Discovery, Konfiguration und Koordination oder Sicherheit überprüft werden.

Ein solcher Test ist nicht schwer zu schreiben und kann viele Diskussionen über die genaue Auslegung der Vorschriften überflüssig machen. Daher ist dieser Tests sehr hilfreich. Außerdem testet er Szenarien, die automatisierte Tests üblicherweise nicht abdecken – beispielsweise den Ausfall abhängiger Systeme.

Dieser Test gibt nicht unbedingt eine vollständige Sicherheit, dass der Microservice alle Regeln einhält. Aber er kann zumindest überprüfen, ob grundlegende Mechanismen funktionieren.

Technische Standards können einfach mit Skripten getestet werden. Die Skripte sollten den Microservice auf die definierte Art und Weise auf einer virtuellen Maschine installieren und starten. Dann kann das Verhalten zum Beispiel bezüglich Logging und Monitoring überprüft werden. Da die technischen Standards projektspezifisch sind, ist ein einheitlicher Ansatz kaum möglich. Gegebenenfalls kann ein Werkzeug wie Serverspec [9] hilfreich sein. Es dient dazu, den Zustand eines Servers zu überprüfen. Es kann daher beispielsweise sehr leicht überprüfen, ob bestimmte Port belegt sind oder ein bestimmter

Service läuft.

11.9 Fazit

Gründe für Tests sind einmal das Risiko, dass Probleme in Produktion erst erkannt werden, und dann der Test als genaue Spezifikation des Systems ([Abschnitt 11.1](#)).

[Abschnitt 11.2](#) hat mit der Testpyramide gezeigt, wie Tests aufgebaut sein sollten: Der Fokus sollte auf schnellen, einfach automatisierbaren Unit-Tests liegen. Sie decken das Risiko ab, dass es Fehler in der Logik gibt. Integrationstests und UI-Tests sichern nur noch die Integration der Microservices und die korrekte Integration in die UI ab.

Wie [Abschnitt 11.3](#) gezeigt hat, können Microservices noch auf andere Art mit dem Risiko von Problemen in Produktion umgehen: Microservice Deployments sind schneller, sie beeinflussen nur einen kleinen Teil des Systems und Microservices können sogar blind in Produktion mitlaufen. Dadurch sinkt das Risiko des Deployments. Es kann daher sinnvoll sein, statt ausführlicher Tests lieber das Deployment in Produktion so weit zu optimieren, dass es praktisch risikolos ist. Ebenso hat der Abschnitt erläutert, dass es bei Microservice-Systemen zwei Arten von Testpyramiden gibt: eine pro Microservice und eine für das Gesamtsystem.

Das Problem beim Test des Gesamtsystems ist, dass bei jeder Änderung an jedem Microservice ein Durchlauf durch diesen Test notwendig ist. Daher kann dieser Test ein Flaschenhals sein und sollte sehr schnell sein. Ein Ziel bei Microservice-Tests ist daher die Reduktion der Integrationstests über alle Microservices ([Abschnitt 11.4](#)).

Bei der Ablösung von Legacy-Anwendungen darf nicht nur deren Funktionalität in die Microservices verschoben werden, sondern auch die Tests für die Funktionalitäten müssen in die Tests der Microservices verschoben werden ([Abschnitt 11.5](#)). Jede Änderung an einem Microservice muss außerdem in der Integration mit der Version der Legacy-Anwendung aus Produktion getestet werden. Die Legacy-Anwendung hat meistens einen wesentlichen langsameren Release-Zyklus als die Microservices. Daher muss die Version der Legacy-Anwendung, die gerade in Entwicklung ist, zusammen mit den Microservices getestet werden.

Für die Tests einzelner Microservices müssen die anderen Microservices durch Stubs ersetzt werden. So können die Tests der Microservices voneinander entkoppelt werden. [Abschnitt 11.6](#) hat zur Erzeugung von Stubs einige konkrete Technologien vorgestellt.

In [Abschnitt 11.7](#) wurden Consumer-driven Contract Tests vorgestellt. Mit diesem Ansatz werden die Verträge zwischen den Microservices explizit. So kann ein Microservice überprüfen, ob er die Anforderungen der anderen Microservices erfüllt – ohne dass dazu ein Integrationstest notwendig wäre. Auch für diesen Bereich gibt es einige Werkzeuge.

[Abschnitt 11.8](#) schließlich hat gezeigt, dass technische Anforderungen an die Microservices ebenfalls automatisiert getestet werden können. Dadurch kann eindeutig herausgefunden werden, ob ein Microservice alle technischen Standards erfüllt.

Wesentliche Punkte

- Etablierte Best Practices wie die Testpyramide sind auch für Microservices sinnvoll.
- Gemeinsame Tests über alle Microservices können ein Flaschenhals werden und sollten daher beispielsweise durch Consumer-driven Contract Tests entlastet werden.
- Stubs können mit passenden Werkzeugen aus Microservices erstellt werden.

11.10 Links & Literatur

- [1] <http://martinfowler.com/articles/consumerDrivenContracts.html>
- [2] <http://thoughtworks.github.io/pacto/>
- [3] <https://github.com/realestate-com-au/pact>
- [4] <https://github.com/DiUS/pact-jvm>
- [5] <http://www.mbtest.org/>
- [6] <http://wiremock.org/>
- [7] <https://github.com/dreamhead/moco>
- [8] <https://github.com/azagniotov/stubby4j>
- [9] <http://serverspec.org/>

12 Betrieb und Continuous Delivery von Microservices

Deployment und Betrieb sind weitere Bestandteile der Continuous-Delivery-Pipeline (siehe [Abschnitt 11.1](#)). Nach dem Testen der Software im Rahmen der Pipeline gehen die Microservices in Produktion. Dort sammeln Monitoring und Logging Informationen, die für die weitere Entwicklung der Microservices genutzt werden können.

Der Betrieb eines Microservice-Systems ist aufwendiger als der Betrieb eines Deployment-Monolithen. Es gibt viel mehr deploybare Artefakte, die auch alle überwacht werden müssen. [Abschnitt 12.1](#) zeigt die typischen Herausforderungen konkret auf. Logging behandelt der [Abschnitt 12.2](#). In [Abschnitt 12.3](#) steht das Monitoring der Microservices im Mittelpunkt. Das Deployment beschreibt schließlich der [Abschnitt 12.4](#). [Abschnitt 12.5](#) zeigt notwendige Möglichkeiten zur Steuerung des Microservice von außen und [Abschnitt 12.6](#) schließlich geeignete Infrastrukturen für den Betrieb.

Die Herausforderungen im Betrieb sind nicht zu unterschätzen. In diesem Bereich lauern oft die komplexesten Probleme.

12.1 Herausforderungen beim Betrieb von Microservices

Wer bisher nur Deployment-Monolithen betrieben hat, ist bei Microservices damit konfrontiert, dass es sehr viel mehr deploybare Artefakte gibt, weil jeder Microservice unabhängig in Produktion gebracht wird. Fünfzig, hundert oder mehr Microservices sind durchaus realistisch. Die konkrete Zahl hängt von der Größe des Projekts und der Größe der Microservices ab. Solche Größenordnungen an deploybaren Artefakten sind außerhalb von Microservices-Architekturen kaum anzutreffen. Alle diese Artefakte müssen einzeln versioniert werden, weil nur so nachvollziehbar ist, welcher Code gerade in Produktion läuft. Außerdem kann so jeder Microservice unabhängig in einer neuen eigenen Version in Produktion gebracht werden.

Wenn es so viel mehr Artefakte gibt, muss es entsprechend mehr Continuous-Delivery-Pipelines geben. Sie umfassen nicht nur das Deployment in Produktion, sondern auch die verschiedenen Testphasen. Ebenso müssen viel mehr Artefakte in Produktion mit Logging und Monitoring überwacht werden. Das ist nur möglich, wenn alle diese Prozesse weitgehend automatisiert sind. Wo es nur wenige Artefakte gibt, sind manuelle Nacharbeiten oder Eingriffe vielleicht noch akzeptabel. Bei der großen Anzahl von Artefakten einer Microservices-Architektur ist ein solcher Weg nicht mehr möglich.

Die Herausforderungen im Bereich Deployment und Infrastruktur sind sicher die größten bei der Einführung von Microservices. Viele Organisationen beherrschen das notwendige Maß von Automatisierung nicht, obwohl es auch ohne Microservices erhebliche Vorteile bringt und längst Gang und Gabe sein sollte.

Um die notwendige Automatisierung zu erreichen, gibt es unterschiedliche Ansätze:

Die einfachste Möglichkeit ist es, diese Herausforderung In Teams delegieren in die Teams zu delegieren, die für die Entwicklung der Microservices zuständig sind. Die Teams müssten dann neben der Entwicklung auch den Betrieb der eigenen Microservices übernehmen. Es ist ihnen freigestellt, dazu geeignete Automatisierungen zu nutzen oder von anderen Teams zu übernehmen.

Das Team muss noch nicht einmal alle Bereiche abdecken. Wenn für den zuverlässigen Betrieb keine Auswertung von Log-Daten notwendig ist, so kann das Team entscheiden, keine Auswertung von Logs umzusetzen. Ein zuverlässiger Betrieb ohne Überwachung der Log-Ausgaben ist zwar eigentlich so gut wie ausgeschlossen – aber dieses Risiko trägt dann das Team.

Dieser Ansatz funktioniert nur, wenn die Teams viel Betriebs-Know-how haben. Ein weiteres Problem ist, dass durch diesen Ansatz das Rad mehrfach erfunden wird: Jedes Team setzt die Automatisierung selber um und nutzt dabei möglicherweise andere Werkzeuge. Bei diesem Ansatz besteht die Gefahr, dass der sowieso schon aufwendige Betrieb der Microservices durch uneinheitliche Ansätze noch aufwendiger wird. Der Aufwand wird in die Teams verlagert und behindert die schnelle Umsetzung neuer Features. Die dezentrale Entscheidung über Technologien erhöht aber die Unabhängigkeit der Teams.

Wegen der höheren Effizienz kann für das Deployment Vereinheitlichung Vereinheitlichung sinnvoll sein. Der einfachste Weg zu einheitlichen Werkzeugen ist, für alle Bereiche – Deployment, Test, Monitoring, Logging, Deployment-Pipeline – je ein Werkzeug vorzuschreiben. Dazu kommen noch Regeln und Best Practices wie beispielsweise Immutable Server oder die Trennung von Build-Umgebung und Deployment-Umgebung. Dann werden alle Microservices exakt identisch aufgesetzt. Das vereinfacht den Betrieb, weil die Teams nur ein Werkzeug für jeden Bereich beherrschen müssen.

Eine andere Möglichkeit ist es, das Verhalten des Systems Verhalten spezifizieren zu spezifizieren. Ein Beispiel: Wenn Log-Ausgaben serviceübergreifend einheitlich ausgewertet werden sollen, reicht es aus, ein einheitliches Log-Format zu definieren. Es muss nicht unbedingt das Log-Framework vorgegeben werden. Natürlich ist es sinnvoll, mindestens für ein Log-Framework eine Konfiguration anzubieten, die dieses Ausgabeformat erzeugt. Das erhöht die Motivation für die Teams, dieses Log-Framework zu benutzen. So wird die Einheitlichkeit nicht erzwungen, sondern sie ergibt sich, weil die Teams ihren eigenen Aufwand minimieren. Wenn ein Team einen Vorteil in einem anderen Log-Framework oder einer anderen Programmiersprache sieht, die andere Log-Frameworks erzwingt, kann es diese Technologien nach wie vor nutzen.

Die Definition eines einheitlichen Formats für Log-Ausgaben hat noch einen weiteren Vorteil: Die Informationen können an unterschiedliche Werkzeuge geliefert werden, die Log-Dateien unterschiedlich aufbereiten. So kann der Betrieb die Log-Dateien nach Fehlern durchsuchen, während die Geschäftsseite Statistiken erzeugt. Betrieb und

Geschäftsseite können unterschiedliche Werkzeuge verwenden, die das einheitliche Format als gemeinsame Basis haben.

Welche Entscheidungen das Team treffen kann und welche für das gesamte Projekt getroffen werden, entspricht der Teilung der Architektur in Mikro- und Makro-architektur (siehe [Abschnitt 13.3](#)). Zur Mikroarchitektur zählen die Entscheidungen, die das Team treffen kann, und zur Makro-Architektur zählen Entscheidungen, die übergreifend über alle Teams getroffen werden. Die Technologien oder das gewünschte Verhalten für Logging können entweder ein Teil der Makro-Architektur oder Teil der Mikro-Architektur sein.

Templates bieten eine Möglichkeit, Microservices in diesen Bereichen zu vereinheitlichen und die Produktivität der Teams zu verbessern. Ein Template kann anhand eines sehr einfachen Microservice zeigen, wie die Technologien genutzt werden können und wie sich Microservices in die Betriebsinfrastruktur integrieren können. Das Beispiel kann einfach eine Anfrage mit einem konstanten Wert beantworten – es geht ja nicht um die fachliche Komplexität.

Wenn ein Team einen neuen Microservice implementiert, ist das mit dem Template einfach und schnell möglich. Gleichzeitig kann so jedes Team den Standard-Technologie-Stack besonders einfach nutzen, sodass einheitliche technische Lösungen gleichzeitig die für die Teams attraktivsten Lösungen sind. Templates erreichen eine große technische Einheitlichkeit zwischen den Microservices, ohne dass sie erzwungen wird. Und die falsche Benutzung des Technologie-Stacks wird vermieden, wenn in dem Template die korrekte Nutzung umgesetzt wird.

Ein Template sollte neben dem Code für einen beispielhaften Microservice auch die komplette Infrastruktur enthalten. Das betrifft die Continuous-Delivery-Pipeline, den Build, die Continuous-Integration-Plattform, das Deployment in Produktion und die Ressourcen für den Ablauf des Microservice. Gerade Build und Continuous-Delivery-Pipeline sind besonders wichtig, weil das Deployment einer großen Anzahl von Microservices nur möglich ist, wenn diese Bereiche automatisiert sind.

Wenn das Template tatsächlich die komplette Infrastruktur enthält, kann es sehr komplex sein – selbst wenn der Microservice sehr einfach ist. Es ist aber nicht unbedingt notwendig, gleich eine vollständige und perfekte Lösung zu haben, sondern das Template kann durchaus schrittweise aufgebaut werden.

Das Template dient als Kopiervorlage. Dadurch ergibt sich natürlich das Problem, dass Änderungen an dem Template sich in den existierenden Microservices nicht wiederfinden. Dafür ist dieser Ansatz aber einfacher zu realisieren als ein Ansatz, bei dem die Änderungen automatisch übernommen werden. Außerdem würde ein solcher Ansatz Abhängigkeiten zwischen dem Template und praktisch allen Microservices erzeugen. Solche Abhängigkeiten sollen Microservices aber möglichst vermeiden.

Durch die Templates wird das Anlegen neuer Microservices wesentlich einfacher. Dadurch werden Teams eher neue Microservices anlegen. So können sie Microservices auch einfacher in mehrere Microservices aufteilen. Also helfen Templates dabei, Microservices klein zu halten. Wenn die Microservices eher klein sind, können die

Vorteile der Microservices-Architektur noch besser ausgenutzt werden.

12.2 Logging

Mit Logging kann eine Anwendung sehr einfach Informationen darüber zur Verfügung stellen, welche Ereignisse aufgetreten sind. Das können Fehler sein, aber auch Ereignisse wie die Registrierung eines neuen Benutzers, die vor allem für Statistiken interessant sind. Schließlich können Log-Daten Entwicklern bei der Fehlersuche helfen, indem sie detaillierte Informationen zur Verfügung stellen.

In normalen Systemen haben Logs den Vorteil, dass sie sehr einfach geschrieben werden können und dass die Daten ohne komplexe Maßnahmen persistent sind. Außerdem können Log-Dateien von Menschen gelesen und auch mit einfachen Werkzeugen durchsucht werden.

Bei Microservices sind das Schreiben von Log-Daten in eine Datei und die Analyse der Datei kaum ausreichend:

- Viele Anfragen können nur durch das Zusammenspiel von mehreren Microservices behandelt werden. Dann ist die Log-Datei eines einzelnen Microservice nicht ausreichend, um den Gesamtlauf zu verstehen.
- Die Last wird oft auf mehrere Instanzen eines Microservice verteilt. Daher hilft die Information aus der Log-Datei einer einzelnen Instanz nur wenig.
- Schließlich können wegen höherer Last, neuen Releases oder dem Ausfall von Instanzen ständig neue Instanzen eines Microservice gestartet werden. Die Daten aus einer Log-Datei können sogar verloren gehen, wenn eine virtuelle Maschine herunterfahren und in der Folge die Festplatte der virtuellen Maschine gelöscht wird.

Es ist nicht notwendig, dass Microservices Logs in ihr Dateisystem schreiben, weil die Informationen dort sowieso nicht ausgewertet werden. Nur das Speichern im zentralen Log-Server ist auf jeden Fall notwendig. Das hat auch den Vorteil, dass die Microservices weniger Speicher auf der Festplatte verwenden.

Meistens loggen Anwendungen einfach Texte. Das zentralisierte Logging parst die Texte. Beim Parsen werden relevante Informationen wie Zeitstempel oder Servernamen extrahiert. Oft geht das Parsen sogar weiter und betrachtet die Texte genauer. Ist es möglich, aus den Logs z. B. die Identität des aktuellen Nutzers zu ermitteln, dann können aus dem Log-Datenbestand alle Informationen zu einem Benutzer herausgesucht werden. Der Microservice versteckt sozusagen die relevanten Informationen in einem Text, den das Log-System dann wieder auseinanderschneidet. Um das Parsen der Informationen zu vereinfachen, können die Log-Daten in einem Datenformat wie JSON übertragen werden. Dann können beim Loggen die Daten strukturiert werden. Sie werden nicht erst in einen Text verpackt, der dann aufwendig geparsst werden muss. Ebenso ist es sinnvoll, einheitliche Standards zu haben: Wenn ein Microservice etwas als Fehler loggt, sollte das auch tatsächlich ein Fehler sein. Ebenso sollte die Semantik der anderen Log-Levels über

alle Microservices einheitlich sein.

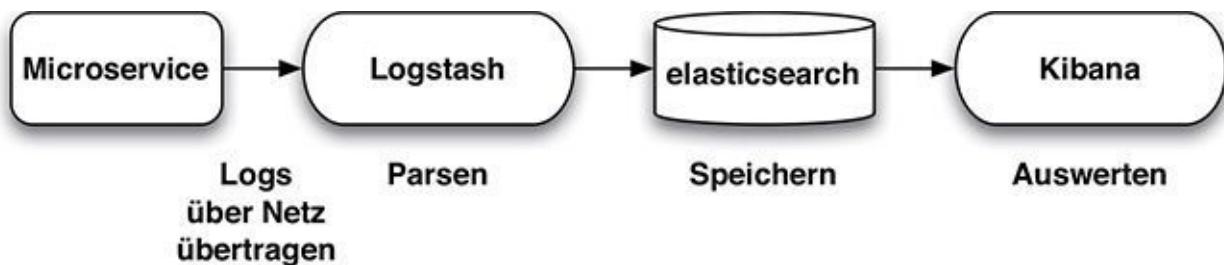
Microservices können das zentrale Logging unterstützen, indem sie Log-Daten direkt über das Netzwerk verschicken. Die meisten Log-Bibliotheken unterstützen so ein Vorgehen. Dabei können spezielle Protokolle wie GELF (Graylog Extended Log Format) [21] zum Einsatz kommen oder lange etablierte Protokolle wie syslog, das Grundlage für das Logging in UNIX-Systemen ist. Werkzeuge wie der logstash-forwarder [15], Beaver [16] oder Woodchuck [17] sind dazu gedacht, lokale Dateien über das Netzwerk an einen zentralen Log-Server zu schicken. Sie sind dann sinnvoll, wenn die Log-Daten auch in Dateien lokal vorgehalten werden sollen.

Technologien zum Loggen über das Netz

Als Werkzeuge für das Sammeln und Verarbeiten der Logs auf einem zentralen Server können beispielsweise Logstash, Elasticsearch und Kibana dienen.

Abb. 12–1 Aufbau einer ELK-Log-Infrastruktur

ELK für zentralisiertes Logging



- Mit *Logstash* [18] können die Log-Dateien geparsst und von den Servern im Netz eingesammelt werden. Logstash ist ein sehr mächtiges Werkzeug. Es kann Daten aus einer Quelle auslesen, die Daten verändern oder filtern und schließlich in eine Senke schreiben. Logstash unterstützt neben dem Einlesen von Logs aus dem Netzwerk und dem Speichern in Elasticsearch viele andere Datenquellen und Datensenken. So können Daten aus Message Queues oder Datenbanken gelesen oder in sie geschrieben werden. Schließlich kann Logstash die Daten auch parsen und ergänzen – beispielsweise kann ein Zeitstempel zu jedem Log-Eintrag hinzugefügt oder einzelne Felder können ausgeschnitten und weiterverarbeitet werden.
- *Elasticsearch* [19] speichert die Log-Daten und macht sie für eine Analyse verfügbar. Elasticsearch kann die Daten nicht nur als Volltextsuche durchsuchen, sondern es kann auch in einzelnen Feldern von strukturierten Daten suchen und die Daten ähnlich wie eine Datenbank dauerhaft speichern. Schließlich bietet Elasticsearch Statistikfunktionen an und kann damit die Daten auswerten. Elasticsearch ist als Suchmaschine auf schnelle Antwortzeiten optimiert, sodass die Daten quasi interaktiv analysiert werden können.
- *Kibana* [20] ist eine Weboberfläche, mit der die Daten aus Elasticsearch analysiert werden können. Neben einfachen Queries sind auch statistische Auswertungen, Visualisierung und Diagramme möglich.

Die Tools bilden zusammen den ELK-Stack (Elasticsearch, Logstash, Kibana). Alle drei sind Open-Source-Projekte und stehen unter der Apache 2.0-Lizenz.

Gerade bei Microservices fallen Log-Daten oft in großen Mengen an. Daher sollte ein System für die

ELK skalieren

zentrale Verarbeitung von Logs für Microservices sehr skalierbar sein. Ein Vorteil des ELK-Stacks ist die gute Skalierbarkeit:

- *Elasticsearch* kann die Indizes in Shards aufteilen. Jeder Eintrag wird einem Shard zugeteilt. Da die Shards auf unterschiedlichen Servern liegen können, kann so die Last verteilt werden. Zusätzlich können Shards über mehrere Server repliziert werden, um so die Ausfallsicherheit zu verbessern. Außerdem kann ein lesender Zugriff auf ein beliebiges Replikat der Daten erfolgen. Replikate können so der Skalierung lesender Zugriffe dienen.
- *Logstash* kann Logs in verschiedene Indizes schreiben. Ohne zusätzliche Konfiguration würde Logstash die Daten jedes Tages in einen anderen Index schreiben. Da die aktuellen Daten üblicherweise öfter gelesen werden, kann so die für eine typische Anfrage zu durchsuchende Datenmenge reduziert und die Performance verbessert werden. Außerdem gäbe es noch andere Möglichkeiten, die Daten auf Indizes aufzuteilen – beispielsweise nach Herkunft der Benutzer. Auch damit können zu durchsuchende Datenmengen optimiert werden.
- Log-Daten können in einem *Broker* zwischengespeichert werden, bevor Logstash sie prozessiert. Der Broker dient als Buffer. Er speichert die Nachrichten, falls so viele Log-Nachrichten anfallen, dass sie nicht sofort prozessiert werden können. Als Broker kommt häufig Redis [20] zum Einsatz – eine schnelle In-Memory-Datenbank.

Der ELK-Stack ist nicht die einzige Lösung zur Analyse von Log-Dateien. Graylog [21] ist auch eine Open-Source-Lösung und nutzt ebenfalls Elasticsearch zum Speichern der Log-Daten. Außerdem verwendet es MongoDB für Metadaten. Graylog definiert ein eigenes Format für die Log-Nachrichten: Das schon erwähnte GELF (Graylog Extended Log Format), das die über das Netz übertragenen Daten standardisiert. Erweiterungen für GELF gibt es für viele Log-Bibliotheken und Programmiersprachen. Ebenso können die entsprechenden Informationen aus Log-Daten extrahiert oder mit dem UNIX-Werkzeug syslog erhoben werden. Auch Logstash unterstützt GELF als Eingabe- und AusgabefORMAT, sodass Logstash mit Graylog kombiniert werden kann. Graylog hat ein Webinterface, mit dem eine Auswertung der Informationen aus den Logs möglich ist.

Splunk [22] ist eine kommerzielle Lösung und schon sehr lange am Markt präsent. Splunk positioniert sich als eine Lösung, die nicht nur Log-Dateien analysiert, sondern generell zur Analyse von Maschinendaten dienen kann. Für die Verarbeitung von Logs erfasst Splunk die Daten mit einem Forwarder, stellt sie mit einem Indexer für die Suche zur Verfügung und Search Heads übernehmen die Verarbeitung von Suchanfragen. Den Anspruch als Enterprise-Lösung unterstreicht auch das Sicherheitskonzept. Auswertungen, aber auch Alerts bei bestimmten Problemen sind möglich. Splunk kann durch zahlreiche Plug-ins erweitert werden. Außerdem gibt es Apps, die eine schlüsselfertige Lösung für bestimmte Infrastrukturen wie Microsoft-Windows-Server liefern. Die Software muss nicht unbedingt im eigenen Rechenzentrum installiert werden, sondern steht auch als Cloud-Lösung zur Verfügung.

Es gibt für das Logging unterschiedliche Stakeholder. Stakeholder für Logs

Allerdings sind die Auswertungsmöglichkeiten der Log-Server so flexibel und die Auswertungen so ähnlich, dass meistens ein Werkzeug ausreicht. Die Stakeholder können eigene Dashboards mit den für sie relevanten Informationen anlegen. Bei speziellen Anforderungen können die Log-Daten auch in verschiedene Systeme zur Auswertung weitergeleitet werden.

Oft arbeiten mehrere Microservices für einen Request zusammen. Der Weg des Requests durch die Microservices muss für eine Analyse nachvollziehbar sein. Um alle Log-Einträge zu einem bestimmten Kunden oder einem bestimmten Request herauszufiltern, kann eine Correlation ID verwendet werden. Sie identifiziert eindeutig einen Request an das Gesamtsystem und wird bei allen Kommunikationen zwischen Microservices weitergegeben. So können die Log-Einträge zu einem Request für alle Systeme im zentralen Log-System leicht gefunden und die Verarbeitung des Requests über alle Microservices hinweg verfolgt werden.

Ein solcher Ansatz kann beispielsweise implementiert werden, indem in den Headern oder in der Payload selbst für jede Nachricht die ID des Requests übertragen wird. Viele Projekte implementieren das Übertragen im eigenen Code und nutzen kein Framework. Für Java gibt es die Bibliothek tracee [26], welche die Übertragung der ID implementiert. Einige Log-Frameworks unterstützen einen Kontext, der bei jeder Log-Nachricht mit ausgegeben wird. Dann muss nur beim Empfangen der Nachricht die Correlation ID im Kontext gesetzt werden. So ist es nicht notwendig, die Correlation ID von Methode zu Methode durchzureichen. Wenn die Correlation ID an den Thread gebunden wird, kann das zu Problemen führen, wenn die Verarbeitung der Anfrage sich über mehrere Threads erstreckt. Das Setzen der Correlation ID am Kontext sorgt dafür, dass alle Log-Nachrichten die Correlation ID enthalten. Wie die Correlation ID ausgegeben wird, muss über alle Microservices gleich sein, damit die Suche in den Logs nach einem Request für alle Microservices funktioniert.

Auch bezüglich der Performance ist eine Untersuchung über Microservices hinweg notwendig. Wenn der vollständige Weg des Requests nachvollziehbar ist, kann identifiziert werden, welcher Microservice ein Flaschenhals ist und besonders lange für die Verarbeitung der Anfrage braucht. Mit einem verteilten Tracing kann bei einem Request ermittelt werden, welcher Microservice wie lange für die Beantwortung eines Requests braucht und wo eine Optimierung ansetzen sollte. Zipkin [23][24] ermöglicht genau solche Untersuchungen. Es enthält Unterstützung für verschiedene Netzwerkprotokolle, sodass die ID eines Requests über diese Protokolle automatisch weitergereicht wird. Im Gegensatz zu den Correlation IDs ist das Ziel nicht, Log-Einträge zu korrelieren, sondern das Zeitverhalten der Microservices zu untersuchen. Dafür bietet Zipkin passende Analyse-Werkzeuge an.

Selber ausprobieren und experimentieren

Definiere einen Technologie-Stack, mit dem eine Microservices-Architektur Logging umsetzen kann:

- Wie sollen die Log-Nachrichten formatiert sein?

- Definiere gegebenenfalls ein Logging-Framework.
- Lege eine Technologie für das Sammeln und Auswerten der Logs fest.

Dieser Abschnitt hat einige Werkzeuge für die verschiedenen Bereiche aufgelistet. Welche Eigenschaften sind besonders wichtig? Das Ziel ist keine vollständige Produktevaluation, sondern ein grobes Abwägen der Vor- und Nachteile.

[Kapitel 14](#) zeigt ein Beispiel für eine Microservices-Architektur und in [Abschnitt 14.14](#) gibt es auch einen Hinweis, wie die Architektur um eine Log-Analyse ergänzt werden kann.

Wie geht dein aktuelles Projekt mit Logging um? Lassen sich vielleicht Teile dieser Ansätze und Technologien auch in diesem Projekt umsetzen?

12.3 Monitoring

Monitoring überwacht Metriken eines Microservice und nutzt andere Informationsquellen als Logging. Monitoring verwendet meistens numerische Werte, die Aufschluss über den aktuellen Zustand der Anwendung geben und zeigen, wie der Zustand sich über die Zeit ändert. Solche Werte können die Anzahl der verarbeiteten Anfragen über eine bestimmte Zeit sein, die Dauer der Bearbeitung der Anfragen oder auch Systemwerte wie die CPU- oder Speicherauslastung. Werden bestimmte Schwellwerte unter- oder überschritten, deutet das auf ein Problem hin und kann einen Alarm auslösen, damit jemand das Problem beheben kann. Noch besser ist es, wenn das Problem automatisch gelöst wird. Beispielsweise kann eine Überlastung durch das Starten neuer Instanzen automatisch gelöst werden.

Monitoring bietet ein Feedback aus der Produktion, das nicht nur für den Betrieb, sondern auch für die Entwicklung oder die Fachbereiche relevant ist. Sie können auf Basis der Informationen aus dem Monitoring das System besser verstehen und informierte Entscheidungen treffen, wie das System weiter entwickelt werden soll.

Grundlegende Monitoring-Informationen sollten für Grundlegende Informationen alle Microservices verpflichtend sein. Das erleichtert, sich einen Überblick über den Zustand des Systems zu verschaffen. Alle Microservices sollten die dafür relevanten Informationen im gleichen Format liefern. Außerdem können Bestandteile des Microservice-Systems die Werte ebenfalls nutzen. So kann beispielsweise das Load Balancing durch einen Health-Wert vermeiden, auf Microservices zuzugreifen, die keine Anfragen bearbeiten können.

Zu den grundlegenden Werten können zählen:

- Es sollte einen Wert geben, der die Verfügbarkeit des Microservice anzeigt. Damit signalisiert der Microservice, ob er überhaupt dazu in der Lage ist, Anfragen zu bearbeiten (»Alive«).
- Detaillierte Informationen über die Verfügbarkeit des Microservice sind eine weitere wichtige Metrik. Dazu kann beispielsweise die Information gehören, ob alle von dem Microservice genutzten Microservices erreichbar und alle anderen Ressourcen

verfügbar sind (»Health«). Diese Informationen zeigen nicht nur an, ob der Microservice funktioniert, sondern geben Hinweise darauf, welcher Teil des Microservice gerade nicht zur Verfügung steht und warum er ausgefallen ist. Insbesondere wird offensichtlich, ob der Microservice wegen des Ausfalls eines anderen Microservice nicht zur Verfügung steht oder weil in dem Microservice selbst ein Problem besteht.

- Informationen über die Version des Microservice und weitere Meta-Informationen wie der Ansprechpartner oder verwendete Libraries und ihre Versionen sowie andere genutzte Artefakte können auch als Metriken zur Verfügung gestellt werden. Das kann einen Teil der Dokumentation (siehe [Abschnitt 8.13](#)) abdecken. Oder es kann überprüft werden, welche Version des Microservice tatsächlich gerade in Produktion ist, was die Fehlersuche vereinfacht. Außerdem ist eine automatisierte kontinuierliche Bestandsaufnahme der Microservices und anderen verwendeten Software möglich, die einfach nur diese Werte abfragt.

Weitere Metriken kann das Monitoring ebenfalls erfassen.

Weitere Metriken

Zu den möglichen Werten können beispielsweise

Antwortzeiten, die Anzahl bestimmter Fehler oder die Anzahl von Anfragen gehören. Diese Werte sind meistens spezifisch für einen Microservice, sodass sie nicht unbedingt von allen Microservices angeboten werden müssen. Es kann ein Alarm ausgelöst werden, wenn bestimmte Schwellwerte erreicht werden. Solche Schwellwerte sind pro Microservice unterschiedlich.

Eine einheitliche Schnittstelle zur Abfrage der Werte ist dennoch sinnvoll, wenn alle Microservices dasselbe Monitoring-Werkzeug nutzen sollen. Einheitlichkeit kann in diesem Bereich den Aufwand erheblich reduzieren.

Für die Informationen aus dem Monitoring gibt es unterschiedliche Stakeholder:

Stakeholder

- Der *Betrieb* will frühzeitig über Probleme informiert werden, um einen möglichst reibungslosen Betrieb des Microservice zu ermöglichen. Bei akuten Problemen oder Ausfällen will er einen Alarm bekommen – und zwar zu jeder Tages- und Nachtzeit über verschiedene Medien wie Pager oder SMS. Detaillierte Informationen sind erst wichtig, wenn der Fehler näher analysiert werden muss – oft zusammen mit den Entwicklern. Der Betrieb interessiert sich nicht nur für die Werte aus dem Microservice selbst, sondern auch für Monitoring-Werte des Betriebssystems, der Hardware oder des Netzwerks.
- Die *Entwickler* sind vor allem auf die Informationen aus der Anwendung fokussiert. Sie wollen verstehen, wie die Anwendung in Produktion funktioniert und wie sie von den Nutzern verwendet wird. Daraus leiten sie Optimierungen vor allem auf technischer Ebene ab. Dazu sind spezifische Informationen hilfreich. Wenn die Anwendung beispielsweise auf eine bestimmte Art von Anfragen zu langsam reagiert, muss das System für diese Anfragen optimiert werden. Dazu ist es notwendig, möglichst viele Informationen für genau diese Art von Anfragen zu bekommen. Andere Anfragen sind nicht so interessant. Entwickler werten diese Informationen detailliert aus. Sie sind vielleicht sogar daran interessiert, Anfragen

nur eines bestimmten Nutzers oder Nutzerkreises zu analysieren.

- Der *Fachbereich* interessiert sich vor allem für den fachlichen Erfolg und die resultierenden Geschäftszahlen. Solche Informationen kann die Anwendung speziell für den Fachbereich bereitstellen. Aus ihnen erstellen die Fachbereiche Statistiken und bereiten so Geschäftsentscheidungen vor. Technische Details interessieren den Fachbereich meistens nicht.

Die verschiedenen Stakeholder sind nicht nur an unterschiedlichen Werten interessiert, sondern werten sie auch unterschiedlich aus. Eine Standardisierung des Datenformats ist sinnvoll, um verschiedene Werkzeuge zu unterstützen und dennoch allen Stakeholdern Zugriff auf alle Daten zu ermöglichen.

Abb. 12–2 Stakeholder mit eigenen Daten

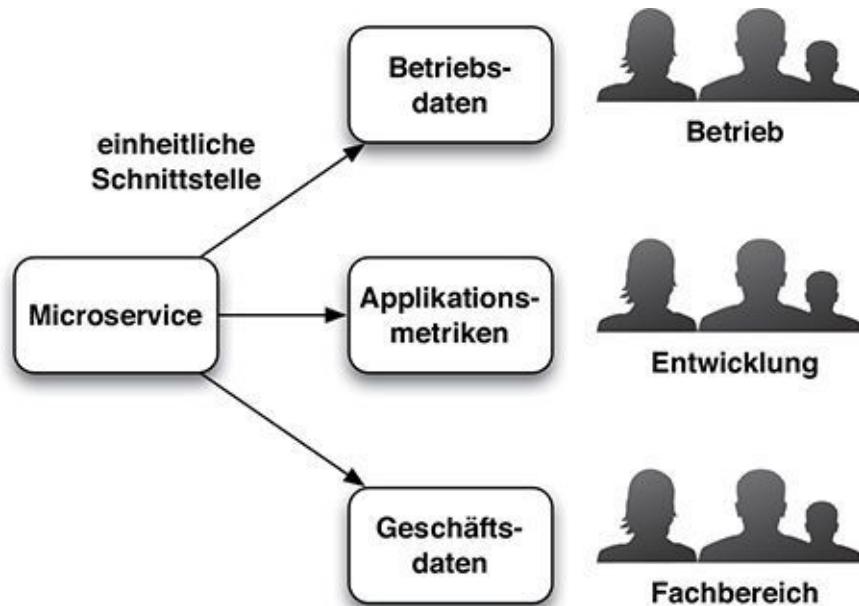


Abbildung 12–2 zeigt ein mögliches Monitoring eines Microservice-Systems im Überblick. Der Microservice bietet die Daten über eine einheitliche Schnittstelle an. Der Betrieb nutzt ein Monitoring, um beispielsweise Schwellwerte zu überwachen. Die Entwicklung verwendet ein detailliertes Monitoring, um die Vorgänge in der Anwendung zu verstehen. Und der Fachbereich lässt sich Geschäftsdaten anzeigen. Wie unterschiedlich die verschiedenen Ansätze sind, ist offen: Die Stakeholder können beispielsweise dieselbe Monitoring-Software mit verschiedenen Dashboards nutzen oder völlig unterschiedliche Software.

Es kann auch sinnvoll sein, Daten mit einem Ereignis wie einem neuen Release zu korrelieren. Dazu müssen Informationen über das Ereignis an das Monitoring übergeben werden. Wenn ein neues Release deutlich mehr Umsatz bringt oder deutlich längere Antwortzeiten verursacht, so ist das sicher eine interessante Erkenntnis.

In gewisser Weise ist Monitoring eine Abwandlung von Tests (siehe Abschnitt 11.4). Während Tests die korrekte Funktion eines neuen Release in einer Testumgebung betrachten, untersucht das Monitoring das Verhalten der Anwendung in einer Produktionsumgebung. Die Integrationstests sollten auch Niederschlag im Monitoring finden. Wenn ein Problem

Mit Ereignissen korrelieren

Monitoring = Tests?

einen Integrationstest zum Scheitern bringt, kann es dafür auch einen Monitoring-Alarm geben. Außerdem sollte das Monitoring auch auf Testumgebungen aktiviert werden, um Probleme schon in den Tests zu finden. Wenn das Risiko eines Deployments durch geeignete Maßnahmen reduziert wird (siehe [Abschnitt 12.4](#)), kann das Monitoring sogar einen Teil der Tests übernehmen.

Eine weitere Herausforderung bei Microservices-Architekturen ist, dass Microservices kommen und gehen.

Dynamische Umgebung

Beim Deployment eines neuen Release kann eine Instanz gestoppt und mit einer neuen Software-Version wieder gestartet werden. Beim Ausfall von Servern beenden sich Instanzen und neue werden gestartet. Aus diesem Grund muss das Monitoring getrennt von den Microservices geschehen. Sonst würde das Beenden eines Microservice die Monitoring-Infrastruktur beeinflussen oder gar zum Ausfall bringen. Außerdem sind Microservices ein verteiltes System. Die Werte einer einzigen Instanz sind überhaupt nicht aussagekräftig. Erst durch das Sammeln der Werte mehrerer Instanzen werden die Monitoring-Informationen relevant.

Verschiedene Technologien können für das Monitoring von Microservices verwendet werden:

Konkrete Technologien

- *Graphite* [1] kann numerische Daten speichern und ist auf die Verarbeitung von Zeitreihen ausgelegt. Solche Daten treten beim Monitoring besonders häufig auf. Die Daten können in einer Webanwendung dargestellt werden. Graphite speichert die Daten in einer eigenen Datenbank. Nach einiger Zeit werden die Daten automatisch gelöscht. Monitoring-Werte nimmt Graphite über eine Socket-Schnittstelle in einem sehr einfachen Format an.
- *Grafana* [2] erweitert Graphite um alternative Dashboards und andere grafische Elemente.
- *Seyren* [3] erweitert Graphite um eine Funktionalität zum Auslösen von Alarmen.
- *Nagios* [4] ist eine umfangreiche Lösung für das Monitoring und kann eine Alternative zu Graphite sein.
- *Icinga* [5] war ursprünglich ein Fork von Nagios und deckt daher auch einen sehr ähnlichen Einsatzbereich ab.
- *Riemann* [6] setzt auf die Verarbeitung von Event-Streams. Dabei nutzt es eine funktionale Programmiersprache, um Logik für die Reaktion auf bestimmte Events zu definieren. Dafür kann ein passendes Dashboard konfiguriert werden. Nachrichten lassen sich per SMS oder E-Mail verschicken.
- *Packetbeat* [25] nutzt einen Agenten, der auf dem zu überwachenden Rechner den Netzwerkverkehr mitschneidet. So ermittelt Packetbeat mit minimalem Aufwand, welche Anfragen wie lange dauern und welche Knoten miteinander kommunizieren. Besonders interessant ist, dass Packetbeat Elasticsearch zum Speichern der Daten nutzt und Kibana für die Analyse. Diese Werkzeuge sind auch für die Analyse von Log-Daten weit verbreitet (siehe [Abschnitt 12.2](#)). Nur ein Stack für die Speicherung und Analyse von Logs und Monitoring reduziert die Komplexität der Umgebung.
- Es gibt außerdem verschiedene kommerzielle Werkzeuge. Dazu zählen *HPs*

Operations Manager [7], *IBM Tivoli* [8], *CA Opscenter* [9] und *BMC Remedy* [10]. Diese Werkzeuge sind sehr umfangreich, sehr lange am Markt und bieten Unterstützung für viele verschiedene Software- und Hardware-Produkte. Solche Plattformen werden oft unternehmensweit eingeführt – und solche Einführungen sind sehr komplexe Projekte. Einige dieser Lösungen können auch Log-Dateien analysieren und überwachen. Für Microservices kann es aufgrund ihrer großen Anzahl und der hohen Dynamik der Umgebung sinnvoll sein, eigene Monitoring-Werkzeuge zu etablieren, selbst wenn bereits ein unternehmensweiter Standard existiert. Wenn die etablierten Prozesse und Werkzeuge einen hohen manuellen Aufwand für die Administration verlangen, kann dieser Aufwand für die große Anzahl Microservices und die Dynamik der Microservice-Umgebung nicht mehr tragfähig sein.

- Monitoring kann in die Cloud verlagert werden. So muss keine Infrastruktur installiert werden. Das vereinfacht die Einführung der Werkzeuge und das Überwachen der Anwendungen. Ein Beispiel ist *NewRelic* [11].

Diese Werkzeuge sind vor allem für den Betrieb und für Entwickler nützlich. Fachliche Auswertungen können mit anderen Werkzeugen vorgenommen werden. Solche Auswertungen basieren nicht nur auf aktuellen Trends und Daten, sondern auch auf historischen Werten. Die Datenmenge ist wesentlich größer als bei Betrieb und Entwicklung. Dazu können die Daten in eine eigene Datenbank exportiert oder mit Big-Data-Lösungen untersucht werden. Gerade die Analyse von Daten beispielsweise aus Webservern ist einer der Bereiche, bei denen Big-Data-Lösungen zuerst genutzt worden sind.

Microservices müssen Daten liefern, die in den Monitoring-Lösungen angezeigt werden. Es ist möglich, die Daten über eine einfache Schnittstelle wie HTTP mit einem Datenformat wie JSON anzubieten. Dann können die Monitoring-Werkzeuge diese Daten auslesen und importieren. Dazu können Adapter als Skripte selbst geschrieben werden. So ist es möglich, verschiedene Werkzeuge mit derselben Schnittstelle mit Daten zu versorgen.

Monitoring im Microservice ermöglichen

In der Java-Welt kann das Metrics-Framework hilfreich sein [12]. Es bietet Funktionalitäten, um eigene Werte zu erfassen und an ein Monitoring-Werkzeug zu schicken. Das ermöglicht, Metriken in der Anwendung zu erfassen und an ein Monitoring-Werkzeug zu übergeben.

Metrics

StatsD [13] kann Werte aus verschiedenen Quellen einsammeln, Berechnungen anstellen und die Ergebnisse an Monitoring-Werkzeuge weitergeben. So lassen sich Werte verdichten, bevor sie im Monitoring landen, um das Monitoring-Werkzeug zu entlasten. Für StatsD gibt es auch viele Client-Bibliotheken, die es vereinfachen, Daten an StatsD zu schicken.

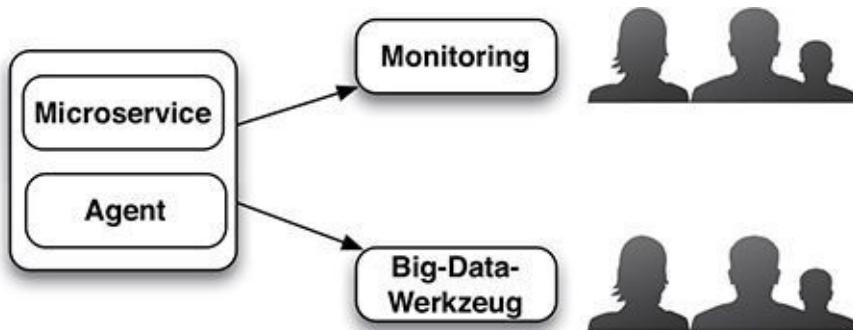
StatsD

collectd [14] sammelt Statistiken über ein System – wie beispielsweise die CPU-Auslastung. Diese Daten können mit einem eigenen Frontend analysiert oder in Monitoring-Werkzeugen abgelegt werden. collectd kann Daten aus einer HTTP-JSON-Datenquelle einsammeln und an das

collectd

Monitoring-Werkzeug weiterschicken. Über verschiedene Plug-ins kann collectd Daten aus dem Betriebssystem und Basisprozessen einsammeln.

Abb. 12–3 Bestandteile einer Monitoring-Lösung



Ein Technologie-Stack für das Monitoring umfasst verschiedene Bestandteile (Abb. 12–3):

Technologie-Stack für Monitoring

- In dem Microservice selber müssen Daten erfasst und für das Monitoring zur Verfügung gestellt werden. Dazu kann entweder eine Bibliothek verwendet werden, die das Monitoring-Werkzeug direkt anspricht, oder die Daten werden mit einer einheitlichen Schnittstelle – beispielsweise JSON über HTTP – angeboten und ein anderes Werkzeug sammelt die Daten ein und übermittelt sie an das Monitoring-Werkzeug.
- Dazu kommt gegebenenfalls ein Agent, um Daten aus dem Betriebssystem und dem System selbst zu erfassen und ebenfalls in das Monitoring einzuspeisen.
- Das Monitoring-Werkzeug speichert und visualisiert die Daten und kann gegebenenfalls Alarne auslösen. Hierbei können verschiedene Aspekte von unterschiedlichen Monitoring-Anwendungen abgedeckt werden.
- Für Analysen historischer Daten oder mit komplexeren Algorithmen kann parallel eine Lösung mit Big-Data-Werkzeugen aufgebaut werden.

Ein Microservice muss also in die Infrastruktur integriert werden. Insbesondere muss er Monitoring-Daten an die Monitoring-Infrastruktur übergeben und einige obligatorische Daten so zur Verfügung stellen. Das kann durch ein passendes Template für einen Microservice und Tests abgesichert werden.

Auswirkungen auf den einzelnen Microservice

Selber ausprobieren und experimentieren

Definiere einen Technologie-Stack, mit dem eine Microservices-Architektur Monitoring umsetzen kann. Dazu zählt eine Definition der Stakeholder und der für sie relevanten Daten. Jeder der Stakeholder muss ein Werkzeug haben, mit dem er die für ihn relevanten Daten analysieren kann. Schließlich muss definiert werden, mit welchen Werkzeugen die Daten erfasst und wie sie gespeichert werden. Dieser Abschnitt hat einige Werkzeuge für die verschiedenen Bereiche aufgelistet. Durch weitere Recherche ist es möglich, einen möglichst gut geeigneten Technologie-Stack zusammenzustellen.

Kapitel 14 zeigt ein Beispiel für eine Microservices-Architektur und in Abschnitt 14.14 gibt es auch einen Hinweis, wie die Architektur um Monitoring ergänzt werden

kann.

Wie geht dein aktuelles Projekt mit Monitoring um? Können einige der Technologien aus diesem Abschnitt auch in diesem Projekt Vorteile bringen? Welche? Warum?

12.4 Deployment

Unabhängiges Deployment ist ein zentrales Ziel von Microservices. Außerdem muss das Deployment automatisiert sein, weil ein manuelles Deployment oder auch nur manuelle Nacharbeiten aufgrund der großen Anzahl Microservices nicht umgesetzt werden können.

Um das Deployment zu automatisieren, gibt es *Deployment-Automatisierung* verschiedene Möglichkeiten:

- Zur Installation können *Skripte* dienen, die lediglich die Software auf dem Rechner installieren. Solche Skripte können beispielsweise als Shell-Skripte umgesetzt werden. Sie können notwendige Software-Pakete installieren, Konfigurationsdateien erstellen und Benutzer anlegen. Solche Skripte können problematisch sein, wenn sie wiederholt aufgerufen werden. Dann findet die Installation einen Rechner vor, auf dem bereits Software installiert ist. Ein Update ist aber anders als eine Neuinstallation. In so einer Situation kann ein Skript fehlschlagen, weil zum Beispiel Benutzer oder Konfigurationsdateien schon vorhanden sind und nicht ohne Weiteres überschrieben werden können. Wenn es mit einem Update umgehen soll, werden die Entwicklung und der Test des Skripts aufwendiger.
- *Immutable Server* (unveränderliche Server) sind eine Möglichkeit, mit dem Problem umzugehen. Statt die Software auf den Servern zu aktualisieren, werden Server komplett neu aufgesetzt. Das vereinfacht nicht nur die Deployment-Automatisierung, sondern erleichtert auch die Reproduktion des Standes der Software auf einem Server. Dazu müssen bei Immutable Servern nämlich nicht mehr Updates, sondern nur Neuinstalltionen betrachtet werden. Eine Neuinstallation ist einfacher reproduzierbar als ein Update, das von verschiedenen Ständen ausgehen kann. Ansätze wie Docker [27] ermöglichen es, den Aufwand für die Installation der Software erheblich zu reduzieren. Docker ist eine Art leichtgewichtige Virtualisierung. Es optimiert auch den Umgang mit virtuellen Festplatten. Wenn bereits eine virtuelle Festplatte mit den richtigen Daten vorliegt, wird sie wiederverwendet, statt die Software neu zu installieren. Bei einer Installation eines Pakets wie Java wird zunächst nach einer virtuellen Festplatte gesucht, die bereits diese Installation hat. Nur wenn es sie nicht gibt, wird die Installation tatsächlich ausgeführt. Sollte sich von einer Version eines Immutable Servers zu einer neuen Version des Immutable Servers nur eine Konfigurationsdatei ändern, wird Docker die alten virtuellen Festplatten größtenteils hinter den Kulissen wiederverwenden und nur die neue Konfigurationsdatei ergänzen. Das reduziert nicht nur den Verbrauch an Festplattenplatz, sondern beschleunigt die Installation der Server auch erheblich. Docker verkürzt auch die Zeit, die eine virtuelle Maschine benötigt, bis sie gebootet hat. Auch diese Optimierung macht Immutable Server gerade mit Docker zu einer

interessanten Option. Das Neuaufsetzen der Server ist mit Docker also sehr schnell und der neue Server kann auch rasch gebootet werden.

- Eine andere Möglichkeit sind Werkzeuge wie *Puppet* [28], *Chef* [29], *Ansible* [30] oder *Salt* [31]. Sie sind auf die Installation von Software spezialisiert. Skripte für diese Werkzeuge beschreiben, wie das System nach der Installation aussehen soll. Bei einem Installationslauf unternimmt das Werkzeug die nötigen Schritte, um das System in diesen Zustand zu versetzen. Beim ersten Lauf auf einem frischen System installiert das Werkzeug die Software vollständig. Lässt man die Installation direkt danach ein zweites Mal durchlaufen, ändert sie das System gar nicht – schließlich ist das System schon im gewünschten Zustand. Außerdem können diese Werkzeuge auch eine große Anzahl Server automatisiert einheitlich installieren und Änderungen auch auf viele Server ausrollen.
- Betriebssysteme aus dem Linux-Bereich haben Package Manager wie *rpm* (*RedHat*), *dpkg* (*Debian/Ubuntu*) oder *zypper* (*SuSE*). Sie ermöglichen es, Software zentralisiert auf eine große Anzahl Server auszurollen. Die genutzten Fileformate sind sehr einfach, sodass sehr leicht ein Package in dem passenden Format erzeugt werden kann. Ein Problem ist die Konfiguration der Software. Package Manager unterstützen üblicherweise Skripte, die bei der Installation ausgeführt werden. Solche Skripte können die notwendigen Konfigurationsdateien generieren. Es kann aber auch für jeden Host ein eigenes Package mit den individuellen Konfigurationen geben. Auch die Installationswerkzeuge aus dem letzten Spiegelstrich können Package Manager für die Installation der eigentlichen Software nutzen, sodass sie selber nur noch die Konfigurationen erstellen.

[Abschnitt 8.8](#) hat schon Werkzeuge beschrieben, die für die Konfiguration von Microservices genutzt werden können. Installation ist generell von der Konfiguration der Software nur schwer zu trennen. Die Installation muss eine Konfiguration erstellen. Daher können viele der Werkzeuge wie beispielsweise Puppet, Chef, Ansible oder Salt auch Konfigurationen erzeugen und auf Servern ausrollen. Damit sind diese Lösungen eine Alternative zu den auf Microservices spezialisierten Konfigurationslösungen.

Installation und Konfiguration

Microservices sollen ein einfaches und unabhängiges Deployment ermöglichen. Dennoch lässt es sich nie ausschließen, dass es in der Produktion zu Problemen kommt. Die Microservices-Architektur alleine hilft schon, das Risiko zu reduzieren. Wenn in Folge eines Problems mit einer neuen Version ein Microservice ausfällt, sollte der Ausfall auf die Funktionalität dieses Microservice begrenzt sein. Davon abgesehen sollte das System weiterfunktionieren. Dazu dienen die Stabilitäts-Pattern und Resilience aus [Abschnitt 10.5](#). Schon alleine aus diesem Grund ist das Deployment eines Microservice wesentlich weniger riskant als das Deployment eines Monolithen. Bei einem Monolithen kann ein Ausfall viel schlechter auf bestimmte Funktionalitäten beschränkt werden. Wenn eine neue Version des Deployment-Monolithen ein Speicherleck hat, wird dadurch der gesamte Prozess abstürzen und damit der gesamte Monolith nicht mehr zur Verfügung stehen. Ein Speicherleck in einem Microservice beeinflusst nur diesen. Es gibt andere Herausforderungen, bei denen Microservices nicht helfen: Schema-Änderungen in

Risiko eines Microservice-Deployments

relationalen Datenbanken sind beispielsweise problematisch, weil sie oft sehr lange dauern und es auch zu Abbrüchen kommen kann – vor allem, wenn die Datenbank schon viele Daten enthält. Da Microservices eigene Datenhaushalte haben, ist eine Schemamigration aber auch immer auf nur einen Microservice begrenzt.

Um das Risiko des Deployments eines Microservice weiter zu reduzieren, gibt es verschiedene Strategien:

- Ein *Rollback* bringt die alte Version des Microservice wieder in Produktion. Der Umgang mit der Datenbank kann problematisch sein: Oft arbeitet die alte Version des Microservice nicht mehr mit dem Datenbankschema, das die neue Version gegebenenfalls erzeugt hat. Wenn schon Daten in der Datenbank stehen, die das neue Schema ausnutzen, kann es sehr schwierig sein, den alten Zustand wieder herzustellen, ohne dabei die neuen Daten zu verlieren. Außerdem ist das Rollback nur schwer testbar.
- Beim *Roll Forward* wird eine neue Version des Microservice in Produktion gebracht, die den Fehler nicht mehr enthält. Das Vorgehen ist identisch mit dem Vorgehen beim Deployment jeder anderen neuen Version des Microservice, sodass keine besonderen Maßnahmen notwendig sind. Die Änderung ist eher klein, sodass Deployment und der Durchlauf durch die Continuous-Delivery-Pipeline schnell erfolgen sollten.
- Noch radikaler ist *Continuous Deployment*: Es wird jede Änderung an einem Microservice in Produktion gebracht, wenn sie die Continuous-Delivery-Pipeline erfolgreich durchlaufen hat. Dadurch sinken die Zeiten für das Beheben eines Problems weiter. Außerdem sind so die Änderungen pro Release kleiner, was das Risiko weiter reduziert und leichter nachvollziehbar macht, welche Code-Änderung zu einem Problem geführt hat. Continuous Deployment ist die logische Konsequenz, wenn der Deployment-Prozess so gut funktioniert, dass der Schritt in die Produktion nur noch eine Formalität ist. Das Team arbeitet auch viel qualitätsbewusster, wenn jede Änderung tatsächlich in Produktion geht.
- Ein *Blue/Green-Deployment* baut eine komplett neue Umgebung mit der neuen Version des Microservice auf. Das Team kann die neue Version vollständig testen und dann in Produktion nehmen. Sollte es Probleme geben, kann die alte Version wieder genutzt werden, die parallel weiter bereitsteht. Auch in diesem Fall gibt es Herausforderungen bei Änderung am Schema der Datenbank. Beim Umschwenken muss auch die Datenbank umgeschwenkt werden. Daten, die zwischen dem Aufbau der neuen Umgebung und dem Umschwenken in die alte Datenbank geschrieben worden sind, müssen in die neue Datenbank übernommen werden.
- *Canary Releasing* basiert auf der Idee, zunächst einen Server aus einem Cluster auf die neue Version zu bringen. Wenn sich die Version bewährt, können die anderen Server auch auf die neue Version gebracht werden. Die Datenbank muss parallel die alte und die neue Version des Microservice unterstützen.
- *Microservices* können auch blind in Produktion laufen. Dann werden sie mit allen Anfragen versorgt, aber sie dürfen keine Daten ändern und ausgehende Aufrufe werden ebenfalls nicht weitergeleitet. Durch Monitoring, Analyse der Logs und Vergleich mit der alten Version kann herausgefunden werden, ob der neue Service

korrekt implementiert ist.

Theoretisch können diese Vorgehensweisen auch mit Deployment-Monolithen umgesetzt werden. In der Praxis ist das aber sehr schwierig. Bei Microservices ist es einfacher, denn sie sind viel kleinere Deployment-Einheiten. Microservices benötigen weniger umfangreiche Tests. Installation und Hochfahren der Microservices laufen schneller. Daher können Microservices schneller durch die Continuous-Delivery-Pipeline in Produktion gebracht werden. Das kommt beispielsweise Roll Forward oder Rollback zugute, weil bei einem Problem weniger Zeit bis zu einem Fix vergeht. Ein Microservice benötigt im Betrieb weniger Ressourcen. Das ist bei Canary Releasing oder Blue/Green Deployment hilfreich, weil neue Umgebungen aufgebaut werden müssen. Wenn das mit weniger Ressourcen möglich ist, kommt es diesen Ansätzen zugute. Bei einem Deployment-Monolithen ist es oft sehr schwierig, überhaupt eine Umgebung aufzubauen.

Gemeinsames oder getrenntes Deployment?

von Jörg Müller, Hypoport AG

Die Frage, ob verschiedene Services gemeinsam oder unabhängig voneinander ausgerollt werden, spielt eine größere Rolle, als man vielleicht vermuten würde. Diese Erfahrung mussten wir im Rahmen eines Projektes machen, das vor ungefähr fünf Jahren gestartet ist.

Microservices spielten als Begriff noch keine Rolle, eine gute Modularisierung war aber von Anfang an unser Ziel. Die gesamte Applikation bestand anfangs aus einer Reihe von Webmodulen in Form von typischen Java-Web-Application-Archiven (WAR). Diese bestanden wiederum jeweils aus mehreren Modulen, die sowohl fachlich als auch technisch geschnitten waren. Neben der Modularisierung setzten wir von Beginn an auf Continuous Deployment als Methode, die Anwendung auszurollen. Jeder Commit jedes Entwicklers geht also tatsächlich auch in Produktion.

Es lag am Anfang nahe, für die gesamte Anwendung eine gemeinsame integrierte Deployment-Pipeline zu bauen. Dadurch wurde ein integratives Testen aller Bestandteile erleichtert. Eine gemeinsame Version der gesamten Anwendung ermöglichte kontrollierbares Verhalten, auch wenn mehrere Bestandteile der Anwendung gleichzeitig geändert wurden. Schließlich konnte auch die Pipeline selbst einfacher erstellt werden. Letzteres war ein wichtiges Argument, da zu dem Zeitpunkt relativ wenige Tools für Continuous Deployment existierten und das meiste von uns selbst erstellt werden musste.

Nach einiger Zeit zeigten sich allerdings die Schattenseiten dieses Vorgehens. Die erste Konsequenz war eine immer längere Laufzeit der Deployment-Pipeline. Je größer die Zahl der Komponenten war, die gebaut, getestet und ausgerollt werden mussten, desto länger dauerte dieser Prozess. Die Vorteile eines Continuous Deployments nahmen sehr schnell ab, als die Laufzeit der Pipeline ein bestimmtes Maß überschritt. Die erste Gegenmaßnahme war eine Optimierung, dass nur noch die Komponenten gebaut und getestet wurden, die sich geändert hatten. Dies erhöhte jedoch die Komplexität der Deployment Pipeline erheblich. Gleichzeitig konnten bestimmte Probleme, wie die Laufzeit bei Änderung an zentralen Komponenten oder die Größe

der Artefakte, dadurch nicht verbessert werden.

Es existierte auch noch ein subtileres Problem. Ein gemeinsames Rollout mit integrativen Tests bot ein starkes Sicherheitsnetz. Es war leicht möglich, gemeinsame Änderungen in mehreren Modulen durchzuführen. Das aber verleitete dazu, Schnittstellen zwischen den Modulen oft zu ändern, weil es so leicht machbar war. Das ist grundsätzlich eine gute Sache, führte aber dazu, dass es sehr oft notwendig wurde, das gesamte System integrativ zu starten. Speziell beim lokalen Arbeiten wurde dies irgendwann zu einer Bürde. Die Anforderungen an die individuelle Hardware des Entwicklers wurden sehr hoch und die Turnaround-Zeiten verlängerten sich erheblich.

Noch komplizierter wurde dieses Vorgehen, als mehr als ein Team mit dieser integrierten Pipeline arbeiteten. Je mehr Komponenten in einer Pipeline getestet wurden, desto häufiger wurden auch Fehler aufgedeckt. Damit stockte die Pipeline, da zuerst die Fehler behoben werden mussten. Als nur ein Team von der Pipeline abhängig war, war es noch leicht, jemanden zu finden, der die Verantwortung übernahm und das Problem behob. Bei mehreren Teams war diese Verantwortung nicht mehr so klar. Das führte dazu, dass Fehler in der Pipeline länger bestanden. Gleichzeitig stieg die Vielfalt der eingesetzten Technologien. Wiederum erhöhte sich die Komplexität. Diese Pipeline konnte nur noch mit sehr individuellen Lösungen gebaut werden. Es stieg also einerseits der Wartungsaufwand und andererseits sank die Stabilität so weit, dass der Wert von Continuous Deployment nur noch schwer realisiert werden konnte.

Spätestens zu diesem Zeitpunkt war klar, dass das gemeinsame Deployment in einer Pipeline nicht mehr fortgeführt werden konnte. Alle neuen Services, egal ob Microservices oder größere Module, wurden jetzt nur noch separat ausgerollt. Die bisherige, auf gemeinsamem Deployment basierende Pipeline in mehrere Pipelines zu trennen, bedeutete jedoch einen hohen Aufwand.

Bei einem neuen Projekt mit einem gemeinsamen Deployment zu starten, kann die richtige Entscheidung sein. Dies gilt vor allem, wenn die Abgrenzung der einzelnen Services und deren Schnittstellen noch nicht eindeutig feststehen. Dann können gute integrative Tests und leichtes Refactoring sehr hilfreich sein. Ab einer gewissen Größe ist ein getrenntes Deployment aber zwingend notwendig. Indikatoren dafür sind die Zahl der Module oder Services, die Laufzeit und Stabilität der Deployment Pipeline und nicht zuletzt die Frage, wie viele Teams am Gesamtsystem arbeiten. Übersieht man die Hinweise und verpasst den richtigen Zeitpunkt, das Deployment zu trennen, kann man leicht einen Monolithen bauen, der aus vielen kleinen Microservices besteht.

12.5 Steuerung

Zur Laufzeit können Eingriffe in einen Microservice notwendig sein. Beispielsweise kann es erforderlich sein, bei einem Problem mit einem Microservice diesen neu zu starten. Ebenso kann ein Start oder ein Stopp eines Microservice notwendig sein. So kann der Betrieb bei einem Problem eingreifen oder ein Load Balancer Instanzen beenden, die keine Anfragen mehr bearbeiten können.

Die Steuerung ist über verschiedene Maßnahmen möglich:

- Wenn ein Microservice in einer *virtuellen Maschine* läuft, kann die virtuelle Maschine herunterfahren oder neu gestartet werden. Der Microservice selber muss dann keine speziellen Vorkehrungen treffen.
- Das Betriebssystem unterstützen *Dienste*, die beim Start des Betriebssystems mit gestartet werden. Meistens können Dienste mit Betriebssystemmitteln auch gestoppt, gestartet oder neu gestartet werden. Die Installation muss den Microservice dann nur als Dienst registrieren. Das Arbeiten mit Diensten ist für den Betrieb nichts Ungewöhnliches, was für diesen Ansatz reicht.
- Schließlich eine *Schnittstelle*, die z. B. über REST einen Neustart oder ein Herunterfahren zulässt. Eine solche Schnittstelle muss der Microservice selber implementieren. Das unterstützen einige Bibliotheken gerade aus dem Microservice-Bereich – wie beispielsweise Spring Boot, mit dem das Beispiel in [Kapitel 14](#) umgesetzt ist. Eine solche Schnittstelle kann mit einfachen HTTP-Werkzeugen wie curl angesprochen werden.

Technisch ist die Umsetzung dieser Steuerungsmechanismen kein großes Problem, aber sie muss für einen Betrieb der Microservices vorhanden sein. Wenn sie für alle Microservices identisch umgesetzt ist, kann das den Aufwand für den Betrieb des Systems reduzieren.

12.6 Infrastrukturen

Microservices müssen auf einer geeigneten Plattform laufen. Es empfiehlt sich, jeden Microservice in einer eigenen virtuellen Maschine (VM) zu betreiben. Sonst kann ein unabhängiges Deployment und Betrieb der einzelnen Microservices nur schlecht umgesetzt werden.

Wenn mehrere Microservices auf einer virtuellen Maschine laufen, kann ein Deployment eines Microservice einen anderen Microservice beeinflussen. Das Deployment kann eine hohe Last erzeugen oder Änderungen an der virtuellen Maschine vornehmen, die auch andere Microservices auf der virtuellen Maschine betreffen.

Außerdem sollten Microservices voneinander isoliert sein, um so bessere Stabilität und Resilience zu erreichen. Wenn mehrere Microservices auf einer virtuellen Maschine laufen, kann ein Microservice so viel Last erzeugen, dass die anderen Microservices ausfallen. Genau das soll aber verhindert werden: Wenn ein Microservice ausfällt, sollte der Ausfall auf einen Microservice begrenzt sein und nicht weitere Microservices in Mitleidenschaft ziehen. Die Isolation virtueller Maschinen hilft dabei, den Ausfall oder die Last auf einen Microservice zu beschränken.

Die Skalierung der Microservices ist ebenfalls einfacher, wenn jeder Microservice in einer eigenen VM läuft. Wenn die Last zu hoch ist, muss nur eine neue virtuelle Maschine gestartet und in das Load Balancing eingetragen werden.

Bei einem Problem fällt es leichter, den Fehler zu analysieren, wenn alle Vorgänge auf einem virtuellen Rechner zu einem Microservice gehören. Jede Metrik auf dem System gehört dann eindeutig zu diesem Microservice.

Schließlich kann der Microservice als Festplatten-Image ausgeliefert werden, wenn jeder Microservice auf einer eigenen VM läuft. Ein solches Deployment hat den Vorteil, dass die komplette Umgebung der VM genau den Vorgaben des Microservice entspricht und der Microservice einen eigenen Technologie-Stack bis hin zum Betriebssystem mitbringen kann.

Es ist kaum möglich, beim Deployment eines neuen Microservice neue physische Hardware zu installieren.

Virtualisierung oder Cloud

Microservices profitieren von Virtualisierung oder Cloud außerdem, weil dadurch die Infrastrukturen viel flexibler sind. Neue virtuelle Maschinen für Skalierung oder Testumgebungen können problemlos zur Verfügung gestellt werden. In der Continuous-Delivery-Pipeline werden ständig Microservices gestartet, um verschiedene Tests durchzuführen. Ebenso müssen in Produktion abhängig von der Last neue Instanzen gestartet werden.

Dazu sollte es möglich sein, eine neue virtuelle Maschine vollautomatisch zu starten. Der Start neuer Instanzen mit einfachen API-Aufrufen ist das, was eine Cloud anbietet. Es sollte eine Cloud-Infrastruktur zur Verfügung stehen, um eine Microservices-Architektur tatsächlich umsetzen zu können. Virtuelle Maschinen, die vom Betrieb durch manuelle Prozesse bereitgestellt werden, reichen nicht aus. Auch hier zeigt sich, dass Microservices ohne moderne Infrastrukturen kaum betrieben werden können.

Wenn es für jeden Microservice eine eigene virtuelle Maschine gibt, ist es aufwendig, eine Testumgebung mit allen Microservices zu erstellen. Schon eine Umgebung mit relativ wenigen Microservices kann einen Entwicklerrechner überfordern. Der Verbrauch an RAM und CPU ist bei einem solchen Vorgehen sehr hoch. Tatsächlich ist es kaum sinnvoll, einen kompletten virtuellen Rechner für einen Microservice zu nutzen. Schließlich soll nur der Microservice laufen und sich in das Logging und Monitoring integrieren. Daher bietet sich eine Lösung wie Docker an: Viele sonst übliche Betriebssystem-Services sind bei Docker nicht enthalten.

Docker

Docker [27] bietet eine sehr leichtgewichtige Virtualisierung an. Dazu nutzt Docker verschiedene Technologien:

- Statt einer vollständigen Virtualisierung nutzt Docker Linux Container (LXC – LinuX Container) [32]. Eine Unterstützung für ähnliche Mechanismen unter Windows ist angekündigt. Dadurch kann eine leichtgewichtige Alternative zu virtuellen Maschinen implementiert werden: Alle Container nutzen denselben Kernel. Es ist nur eine Instanz des Kernels im Speicher. Prozesse, Netzwerk, Dateisystem und Benutzer sind voneinander getrennt. Im Vergleich zu einer virtuellen Maschine (VM) mit eigenem Kernel und oft auch vielen Betriebssystem-Services hat ein Container einen wesentlich geringeren Overhead. Es ist ohne Weiteres möglich, Hunderte von Linux-Containern auf einem einfachen Laptop zu betreiben. Außerdem startet ein Container wesentlich schneller als eine virtuelle Maschine mit eigenem Kernel und vollständigem Betriebssystem. Der Container muss kein ganzes Betriebssystem booten, sondern startet nur einen neuen Prozess. Der Container selbst fällt nicht besonders ins Gewicht, da er nur eine eigene Konfiguration der Betriebssystemressourcen benötigt.

- Ebenso ist das Dateisystem optimiert: Es lassen sich Basisdateisysteme nutzen, von denen nur gelesen wird. Gleichzeitig können weitere Dateisysteme in den Container eingeblendet werden, auf die auch geschrieben werden kann. Dabei kann ein Dateisystem ein anderes überlagern. Beispielsweise kann ein Basisdateisystem erstellt werden, das ein Betriebssystem enthält. Wird im laufenden Container Software installiert oder werden Dateien geändert, muss der Container nur diese zusätzlichen Dateien in einem kleinen containerspezifischen Dateisystem ablegen. So wird der Speicherbedarf auf der Festplatte für die Container deutlich reduziert.

Außerdem ergeben sich weitere interessante Möglichkeiten: Beispielsweise kann ein Basisdateisystem mit einem Betriebssystem gestartet und dann Software installiert werden. Wie erwähnt werden nur die Änderungen am Dateisystem gespeichert, die bei der Installation der Software vorgenommen werden. Von diesem Delta kann ein Dateisystem erzeugt werden. Dann kann ein Container gestartet werden, der neben dem Basisdateisystem mit dem Betriebssystem auch dieses Delta einblendet – und danach kann weitere Software installiert werden. So kann jede »Schicht« in dem Dateisystem spezifische Änderungen enthalten. Das tatsächliche Dateisystem zur Laufzeit kann aus vielen solchen Schichten komponiert werden. Dadurch können Software-Installationen sehr effizient wiederverwendet werden.

[Abbildung 12–4](#) zeigt ein Beispiel für das Dateisystem eines laufenden Containers: Auf der untersten Ebene liegt eine Ubuntu-Linux-Installation. Darüber sind die Änderungen, die durch die Installation von Java vorgenommen worden sind. Dann folgt die Beispielanwendung. Damit der laufende Container Änderungen in das Dateisystem schreiben kann, ist darüber noch ein Dateisystem, in das der Container Dateien schreibt. Wenn nun der Container eine Datei lesen will, geht er von oben nach unten durch die Schichten, bis er die entsprechende Datei gefunden hat.

Abb. 12–4 Dockers Dateisysteme



Docker-Container bieten eine sehr effiziente Alternative zu Virtualisierung. Es ist aber keine »echte« Virtualisierung, denn jeder Container hat getrennte Ressourcen, eigenen Speicher und eigene Dateisysteme, aber alle teilen sich beispielsweise den Kernel. Daher hat dieser Ansatz einige Nachteile. Ein Docker-Container kann nur Linux nutzen und nur denselben Kernel wie das Host-Betriebssystem – also können beispielsweise keine Windows-Anwendungen betrieben werden. Die Trennung der Container ist nicht so strikt wie bei echten virtuellen Maschinen. Ein Fehler im Kernel würde beispielsweise alle Container betreffen. Ebenso läuft Docker auch nicht auf Mac OS X oder Windows. Docker kann

Docker-Container vs. Virtualisierung

dennnoch auf diesen Plattformen direkt installiert werden. Hinter den Kulissen kommt eine virtuelle Maschine mit Linux zum Einsatz. Für Windows hat Microsoft eine eigene Version angekündigt, die Windows-Container betreiben kann.

Docker-Container müssen miteinander kommunizieren. Beispielsweise kommuniziert eine Webanwendung mit ihrer Datenbank. Dazu exportieren Container Netzwerk-Ports, die andere Container nutzen. Außerdem lassen sich Dateisysteme gemeinsam nutzen. Dort schreiben Container Daten, die andere Container lesen können.

Docker Images umfassen die Daten einer virtuellen Festplatte. Docker Registries erlauben es, Docker Images zu speichern und herunterzuladen. So ist es möglich, Docker Images als Ergebnis eines Build-Prozesses zu speichern und anschließend auf Servern auszurollen. Wegen der effizienten Speicherung von Images ist es problemlos möglich, auch komplexere Installationen performant zu verteilen. Außerdem entstehen viele Cloud-Angebote, die Docker-Container direkt ablaufen lassen können.

Mit Docker haben Microservices eine ideale Ablaufumgebung. Sie schränkt die genutzte Technologie kaum ein, weil jede Art von Linux-Software in einem Docker-Container laufen kann. Mit den Docker Registries können die Docker-Container einfach verteilt werden. Gleichzeitig ist der Overhead eines Docker-Containers gegenüber einem normalen Prozess vernachlässigbar. Da Microservices eine Vielzahl von virtuellen Maschinen benötigen, sind diese Optimierungen sehr wertvoll. Docker ist auf der einen Seite sehr effizient und auf der anderen Seite schränkt es die Technologien nicht ein.

Selber ausprobieren und experimentieren

Unter <http://www.docker.com/tryit/> findet sich das Docker-Online-Tutorial. Arbeitet es durch – es zeigt den grundlegenden Umgang mit Docker. Das Tutorial kann man sehr schnell durcharbeiten. Es läuft vollständig im Browser, sodass keine Installation von Software notwendig ist.

Es gibt unterschiedliche Möglichkeiten, Docker für Server zu nutzen:

- Auf einem *Linux-Server* kann Docker installiert und dann können ein oder mehrere Docker-Container ausgeführt werden. Docker dient dann als Lösung für die Provisionierung der Software. Für einen Cluster werden neue Server gestartet, auf denen wieder Docker-Container installiert werden. Docker dient nur zur Installation der Software auf den Servern.
- Docker-Container werden direkt auf einem *Cluster* ausgeführt. Welcher physische Rechner einen bestimmten Container beheimatet, entscheidet die Software zur Cluster-Verwaltung. Ein solcher Ansatz unterstützt der Scheduler Apache Mesos [37]. Er verwaltet einen Cluster von Servern und weist Jobs jeweils einem Server zu. Mesosphere [38] erlaubt die Ausführung von Docker-Containern mithilfe des Mesos Scheduler. Außerdem unterstützt Mesos viele weitere Arten von Jobs.

- *Kubernetes* [39] unterstützt ebenfalls die Ausführung von Docker-Containern in einem Cluster. Der Ansatz ist jedoch anders als bei Mesos. Kubernetes bietet einen Dienst an, der Pods im Cluster verteilt. Pods sind zusammenhängende Docker-Container, die auf einem physischen Server laufen sollten. Als Basis benötigt Kubernetes nur eine einfache Betriebssysteminstallation – das Cluster-Management setzt Kubernetes um.
- *CoreOS* [40] ist ein sehr leichtgewichtiges Server-Betriebssystem. Mit etcd unterstützt es die clusterweite Verteilung von Konfigurationen. fleetd ermöglicht es, dass Dienste im Cluster deployt werden – bis hin zu redundanter Installation, Ausfallsicherheit, Abhängigkeiten und gemeinsamem Deployment auf einem Knoten. Alle Dienste müssen als Docker-Container deployt werden, während das Betriebssystem selber im Wesentlichen unverändert bleibt.
- *Docker Machine* [41] erlaubt die Installation von Docker auf verschiedenen Virtualisierungs- und Cloud-Systemen. Außerdem kann Docker Machine das Docker-Kommandozeilenwerkzeug so konfigurieren, dass es mit einem solchen System kommuniziert. Zusammen mit *Docker Compose* [42] können mehrere Docker-Container zu einem Gesamtsystem kombiniert werden. Die Beispielanwendung nutzt diesen Ansatz, siehe [Abschnitt 14.6](#) und [Abschnitt 14.7](#).

Kubernetes, CoreOS, Docker Compose, Docker Machine und Mesos beeinflussen natürlich den Betrieb der Software, sodass die Lösungen im Vergleich zu Virtualisierung Änderungen im Betriebsablauf erfordern. Diese Technologien lösen Herausforderungen, die zuvor Virtualisierungslösungen gelöst haben, nämlich einen Cluster von Servern zu verwalten, sodass Container bzw. virtuelle Maschinen in dem Cluster verteilt werden.

PaaS (Platform as a Service) verfolgt einen grundlegend anderen Ansatz. Das Deployment einer Anwendung kann dadurch erfolgen, dass die Anwendung in der Versionskontrolle aktualisiert wird. Das PaaS holt sich die Änderungen, baut die Anwendung und rollt sie auf den Servern aus. Diese Server werden vom PaaS installiert und stellen eine standardisierte Umgebung dar. Die eigentliche Infrastruktur – also die virtuellen Rechner – sind vor der Anwendung versteckt. Das PaaS bietet der Anwendung eine standardisierte Umgebung. Die Umgebung sorgt beispielsweise auch für das Skalieren und kann Dienste wie Datenbanken oder Messaging-Systeme anbieten. Durch die einheitliche Plattform schränken PaaS-Systeme die Technologiefreiheit ein, die sonst bei Microservices möglich ist. Es können nur Technologien verwendet werden, die von dem PaaS unterstützt werden. Dafür werden das Deployment und die Skalierung weiter vereinfacht.

Microservices stellen hohe Anforderungen an die Infrastruktur. Automatisierung ist eine wesentliche Voraussetzung für den Betrieb der vielen Microservices. Ein PaaS bietet dafür eine gute Basis, da es die Automatisierung deutlich vereinfacht. Daher kann ein PaaS vor allem sinnvoll sein, wenn die Entwicklung einer eigenen Automatisierung zu aufwendig und nicht genügend Wissen über den Aufbau der notwendigen Infrastruktur vorhanden ist. Allerdings müssen sich die Microservices auf die Features begrenzen, die das PaaS anbietet. Wenn die Microservices von Anfang an für das PaaS entwickelt worden sind, ist das nicht besonders aufwendig. Müssen sie aber portiert werden, kann das erhebliche Aufwände nach sich ziehen.

Nanoservices (Kap. 15) haben eigene Ablaufumgebungen, die beispielsweise die Technologie-Auswahl noch weiter einschränken. Dafür sind sie oft noch einfacher zu betreiben und noch effizienter bezüglich des Ressourcenverbrauchs.

12.7 Fazit

Der Betrieb eines Microservice-Systems ist eine der entscheidenden Herausforderungen bei Microservices (Abschnitt 12.1). Es gibt eine sehr große Anzahl an Prozessen. Fünfzig oder hundert virtuelle Maschinen sind keine Seltenheit. Die Verantwortung für den Betrieb kann in die Teams delegiert werden. Der Ansatz führt aber zu einem höheren Gesamtaufwand. Sinnvoller ist eine Vereinheitlichung des Betriebs. Templates sind eine Möglichkeit, um eine Vereinheitlichung auch ohne Zwang zu erreichen. Der einheitliche Weg wird durch Templates zum einfachsten Weg.

Für das Logging (Abschnitt 12.2) muss eine zentrale Infrastruktur bereitgestellt werden, die Logs aus den Microservices sammelt. Dafür sind verschiedene Technologien verfügbar. Um einen Aufruf über die verschiedenen Microservices zu verfolgen, kann eine Correlation ID verwendet werden, die den Aufruf eindeutig identifiziert.

Monitoring (Abschnitt 12.3) muss mindestens grundlegende Informationen anbieten wie die Verfügbarkeit des Microservice. Weitere Metriken können beispielsweise einen Überblick über das Gesamtsystem ermöglichen oder für Lastverteilung nützlich sein. Zusätzliche Metriken kann jeder Microservice individuell definieren. Es gibt unterschiedliche Stakeholder für das Monitoring: Betrieb, Entwickler und Fachbereich. Sie sind an unterschiedlichen Werten interessiert und nutzen gegebenenfalls eigene Werkzeuge für die Auswertung der Monitoring-Daten. Jeder Microservice muss eine Schnittstelle anbieten, mit der die verschiedenen Werkzeuge Werte aus der Anwendung abholen können. Die Schnittstelle sollte für alle Microservices identisch sein.

Das Deployment von Microservices (Abschnitt 12.4) muss automatisiert sein. Dazu können einfache Skripte vor allem zusammen mit Immutable Server, spezielle Deployment-Werkzeuge und Package Manager genutzt werden.

Microservices sind kleine Deployment-Einheiten. Sie sind durch Stabilität und Resilience gegen den Ausfall anderer Microservices abgesichert. Daher ist das Risiko eines Deployments schon durch die Microservices-Architekturen an sich reduziert. Strategien wie Rollback, Roll Forward, Continuous Deployment, Blue/Green-Deployment oder ein blindes Mitlaufen in Produktion können das Risiko weiter reduzieren. Mit Microservices sind solche Strategien einfach umsetzbar, weil die Deployment-Einheiten klein sind und der Ressourcenverbrauch der Microservices gering ist. Daher sind Deployments schneller und Umgebungen für Blue/Green-Deployment oder Canary Releasing viel einfacher aufzubauen.

Die Steuerung (Abschnitt 12.5) umfasst einfache Eingriffsmöglichkeiten wie das Starten, Stoppen und Neustarten von Microservices.

Als Infrastruktur für Microservices bietet sich Virtualisierung oder Cloud an (Abschnitt 12.6). Auf jeder VM sollte nur ein Microservice laufen, um eine bessere Isolation, Stabilität und Skalierung zu erreichen. Besonders interessant ist Docker, weil der

Ressourcenverbrauch eines Docker-Containers gegenüber einer VM deutlich reduziert ist. So ist es möglich, auch bei einer großen Anzahl Microservices jedem Microservice einen eigenen Docker-Container zur Verfügung zu stellen. PaaS-Clouds sind ebenfalls interessant. Sie erlauben eine sehr einfache Automatisierung. Allerdings schränken sie die Auswahl an Technologien auch ein.

Dieser Abschnitt fokussiert nur auf die Besonderheiten von Continuous Delivery und Betrieb in einem Microservices-Umfeld. Continuous Delivery ist einer der wichtigsten Gründe für die Einführung von Microservices. Gleichzeitig liegen im Betrieb die größten Herausforderungen. Details zu Continuous Delivery finden sich beispielsweise in [35] oder [36].

Wesentliche Punkte

- Betrieb und Continuous Delivery sind zentrale Herausforderungen für Microservices.
- Die Microservices sollten einheitlich mit Monitoring, Logging und Deployment umgehen. Nur so bleiben die Aufwände im Rahmen.
- Virtualisierung, Cloud, PaaS und Docker sind interessante Alternativen als Infrastruktur für Microservices

12.8 Link & Literatur

- [1] <http://graphite.wikidot.com/>
- [2] <http://grafana.org/>
- [3] <https://github.com/scobal/seyren>
- [4] <http://www.nagios.org/>
- [5] <https://www.icinga.org/>
- [6] <http://riemann.io/>
- [7] <http://www8.hp.com/us/en/software-solutions/operations-manager-infrastructure-monitoring/>
- [8] <http://www-01.ibm.com/software/tivoli/>
- [9] <http://www.ca.com/us/opscenter.aspx>
- [10] <http://www.bmc.com/it-solutions/remedy-itsm.html>
- [11] <http://newrelic.com/>
- [12] <https://github.com/dropwizard/metrics>
- [13] <https://github.com/etsy/statsd>
- [14] <https://collectd.org/>
- [15] <https://github.com/elastic/logstash-forwarder>
- [16] <https://github.com/josegonzalez/beaver>

- [17] <https://github.com/danryan/woodchuck>
- [18] <http://logstash.net/>
- [19] <https://www.elastic.co/products/kibana>
- [20] <http://redis.io/>
- [21] <https://www.graylog.org/>
- [22] <http://www.splunk.com/>
- [23] <https://github.com/twitter/zipkin>
- [24] <https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin>
- [25] <http://packetbeat.com/>
- [26] <https://github.com/tracee/tracee>
- [27] <https://www.docker.com/>
- [28] <http://puppetlabs.com/>
- [29] <https://www.chef.io/>
- [30] <http://www.ansible.com/>
- [31] <http://www.saltstack.com/>
- [32] <https://linuxcontainers.org/>
- [33] <http://12factor.net/>
- [34] <https://www.heroku.com/>
- [35] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, dpunkt.verlag, 2014, ISBN 978-3864902086
- [36] Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9
- [37] <http://mesos.apache.org/>
- [38] <http://mesosphere.com/>
- [39] <http://kubernetes.io/>
- [40] <http://coreos.com/>
- [41] <https://docs.docker.com/machine/>
- [42] <http://docs.docker.com/compose/>

13 Organisatorische Auswirkungen der Architektur

Ein wesentlicher Teil des Microservices-Ansatzes ist, dass jeder Microservice einem Team zugeteilt wird. Daher müssen bei Microservices neben der Architektur auch die Aufteilung der Teams und die Zuständigkeiten für die Microservices betrachtet werden. Dieses Kapitel erläutert die organisatorischen Auswirkungen von Microservices.

In [Abschnitt 13.1](#) geht es um die organisatorischen Vorteile von Microservices. [Abschnitt 13.2](#) zeigt, dass Collective Code Ownership eine Alternative zum Aufstellen der Teams nach dem Gesetz von Conway sein kann. Die Unabhängigkeit der Teams ist eine wichtige Konsequenz von Microservices. [Abschnitt 13.3](#) definiert Mikro- und MakroArchitektur und zeigt, wie diese Ansätze den Teams eine möglichst hohe Autonomie bieten, um eigenständige Entscheidungen zu treffen. Eng verbunden damit ist die Frage nach der Rolle der technischen Führung ([Abschnitt 13.4](#)). DevOps ist ein organisatorischer Ansatz, bei dem Entwicklung (Dev) und Betrieb (Ops) zusammengelegt werden ([Abschnitt 13.5](#)). DevOps hat Synergien mit Microservices. Da es bei Microservices um fachlich unabhängige Entwicklung geht, beeinflussen Microservices auch Product Owner und Fachbereiche. [Abschnitt 13.6](#) erläutert, wie diese Gruppen mit Microservices umgehen können. Wiederverwendbarer Code ist in Microservice-Systemen nur mit organisatorischen Maßnahmen machbar, wie [Abschnitt 13.7](#) verdeutlicht. Und schließlich geht der [Abschnitt 13.8](#) der Frage nach, ob eine Einführung von Microservices ohne Änderungen an der Organisation tragfähig ist.

13.1 Organisatorische Vorteile von Microservices

Microservices sind ein Ansatz, um auch große Projekte mit kleinen Teams anzugehen. Da die Teams unabhängig voneinander sind, ist weniger Koordination zwischen den Teams notwendig. Gerade der Kommunikationsoverhead macht die Arbeit großer Teams so aufwendig. Microservices sind ein Ansatz auf Architekturebene, um dieses Problem zu lösen. Die Architektur hilft, die Kommunikation zu reduzieren und viele kleine Teams im Projekt arbeiten zu lassen statt eines großen. Jedes fachliche Team kann die ideale Größe haben: Die Scrum Guide nennt drei bis neun Mitglieder [5].

Außerdem setzen moderne Unternehmen auf Selbstorganisation und Teams, die selber direkt am Markt aktiv sein können. Microservices unterstützen diesen Ansatz, weil die Services entsprechend dem Gesetz von Conway ([Abschnitt 4.2](#)) jeweils von einem eigenen Team verantwortet werden. Daher passen Microservices gut zu Selbstorganisation. Jedes Team kann unabhängig von anderen neue Features implementieren und den Erfolg am Markt selbst bewerten.

Damit stehen Microservices in einem Spannungsfeld zwischen Unabhängigkeit und Standardisierung: Wenn die Teams selbstständig sein sollen, müssen sie unabhängig sein. Eine Standardisierung beschneidet die Unabhängigkeit. Das betrifft beispielsweise die

Entscheidung, welche Technologie genutzt werden soll. Wird ein Technologie-Stack vorgegeben, können die Teams keine unabhängigen Entscheidungen über die Technologien mehr treffen. Ebenso steht die Unabhängigkeit in einer Spannung zu Redundanzfreiheit: Wenn das System redundanzfrei sein soll, muss es eine Koordination zwischen den Teams geben, um die Redundanzen zu identifizieren und zu eliminieren. Das beeinflusst wiederum die Unabhängigkeit der Teams.

Ein wichtiger Aspekt ist eine weitgehende technische Entkopplung. Microservices können unterschiedliche Technologien nutzen und intern einen völlig unterschiedlichen Aufbau haben. Dadurch müssen sich die Entwickler weniger miteinander abstimmen. Nur grundlegende Entscheidungen müssen gemeinsam getroffen werden. Sonst können Teams die technischen Entscheidungen treffen. Technische Unabhängigkeit

Jeder Microservice kann unabhängig von den anderen Microservices in Produktion gebracht werden. Es ist auch keine Koordination von Release-Terminen oder Testphasen zwischen Teams notwendig. Jedes Team kann seine eigene Geschwindigkeit und seine eigenen Termine wählen. Eine Verzögerung in einem Termin eines Teams beeinflusst die anderen Teams nicht. Separates Deployment

Die Teams sollten jeweils unabhängige Stories und Anforderungen umsetzen. Dann kann jedes Team seine eigenen geschäftlichen Ziele verfolgen. Separate Anforderungsströme

Microservices ermöglichen auf drei Ebenen Unabhängigkeit: Drei Ebenen der Unabhängigkeit

- *Entkopplung durch unabhängige Releases:*

Jedes Team hat einen oder mehrere Microservices. Das Team kann sie unabhängig von den anderen Teams und den anderen Microservices in Produktion bringen.

- *Technologische Entkopplung:*

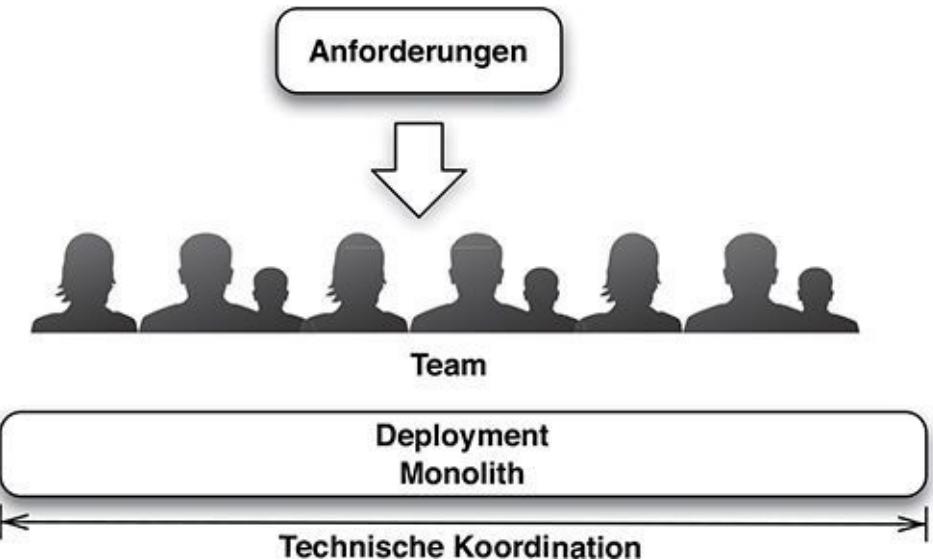
Die technischen Entscheidungen betreffen in erster Linie die eigenen Microservices.

- *Fachliche Entkopplung:*

Durch die Aufteilung der Fachlichkeit in separate Komponenten kann jedes Team eigene Anforderungen umsetzen.

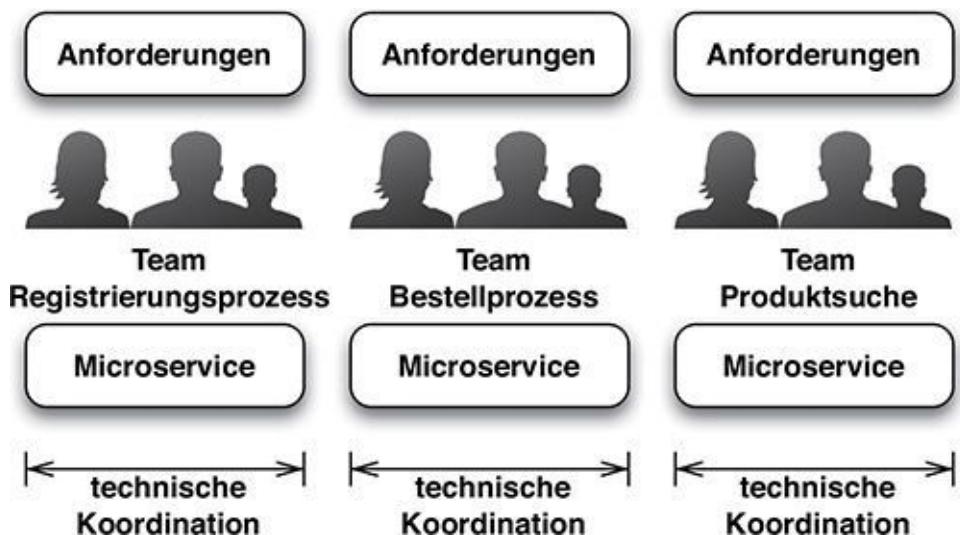
Im Gegensatz dazu betreffen bei einem Deployment-Monolithen die technische Koordination und das Deployment den gesamten Monolithen ([Abb. 13–1](#)). Das macht so enge Koordination zwischen den Entwicklern notwendig, dass alle Entwickler im gesamten Monolithen wie ein Team agieren müssen.

Abb. 13–1 Deployment-Monolith



Voraussetzung für die Unabhängigkeit der Microservices-Teams ist, dass die Architektur auch tatsächlich die notwendige Unabhängigkeit der Microservices bietet. Dazu muss es vor allem eine gute fachliche Architektur geben. Diese Architektur ermöglicht auch die unabhängigen Ströme von Anforderungen für jedes Team.

Abb. 13–2 Aufteilung in Microservices



Im Beispiel aus Abbildung 13–2 gibt es folgende Teams:

- Das Team »Benutzerregistrierung« kümmert sich darum, wie Benutzer sich in dem E-Commerce-Shop registrieren können. Das geschäftliche Ziel kann eine möglichst hohe Anzahl an Registrierungen sein. Neue Features zielen darauf ab, diese Anzahl zu optimieren. Die Komponenten des Teams sind die für die Registrierung notwendigen Prozesse und UI-Elemente, die sie nach Belieben ändern und optimieren können.
- Das Team »Bestellprozess« kümmert sich darum, wie aus dem Einkaufswagen eine Bestellung wird. Ziel ist, dass möglichst viele Einkaufswagen zu Bestellungen werden. Der gesamte Prozess wird von diesem Team implementiert.
- Das Team »Produktsuche« verbessert die Suche nach Produkten. Sie werden daran gemessen, wie viele Suchvorgänge dazu führen, dass Waren in einen Einkaufswagen gelegt werden.

Natürlich kann es weitere Teams mit anderen Aufgaben geben. So wird das Problem der Weiterentwicklung eines E-Commerce-Shops auf mehrere Teams aufgeteilt, die jeweils eigene Ziele haben. Die können sie weitgehend unabhängig voneinander verfolgen, weil die Architektur das System in Microservices aufgeteilt hat, die jedes Team unabhängig weiterentwickeln kann – ohne viel Koordination.

Kleine Projekte haben noch viele weitere Vorteile:

- Die Schätzungen sind besser, weil Schätzungen über kleine Umfänge einfacher sind.
- Kleine Projekte lassen sich besser planen.
- Das Risiko sinkt – wegen der besseren Schätzungen und der besseren Planbarkeit.
- Wenn es doch ein Problem gibt, sind die Auswirkungen kleiner, weil das Projekt kleiner ist.

Microservices bieten außerdem viel mehr Flexibilität. So kann schneller und einfacher entschieden werden. Denn das Risiko ist kleiner und die Änderungen können schneller umgesetzt werden. Das unterstützt agile Software-Entwicklung ideal, die auf solche Flexibilität angewiesen ist.

13.2 Alternativer Umgang mit dem Gesetz von Conway

Abschnitt 4.2 hat das Gesetz von Conway eingeführt. Demnach kann eine Organisation nur Architekturen hervorbringen, die ihren Kommunikationsstrukturen entspricht. In Microservices-Architekturen werden die Teams entsprechend den Microservices aufgestellt. Jedes Team entwickelt einen oder mehrere Microservices. Jeder Microservice wird also von nur genau einem Team entwickelt. Das stellt sicher, dass die fachliche Architektur nicht nur durch die Aufteilung in Microservices umgesetzt wird, sondern durch die organisatorische Aufteilung unterstützt wird. So sind Verstöße gegen die Architektur praktisch unmöglich. Und die Teams können unabhängig Features entwickeln, wenn die Features auf einen Microservice begrenzt sind. Dazu muss der fachliche Schnitt zwischen den Microservices entsprechend gut sein.

Dieses Vorgehen hat jedoch auch Nachteile:

Die Herausforderungen des Gesetzes von Conway

- Die Teams müssen langfristig stabil bleiben. Gerade wenn die Microservices unterschiedliche Technologien nutzen, ist die Einarbeitungszeit in einem Microservice sehr lang. Entwickler können nicht einfach zwischen Teams wechseln. Gerade in Teams mit externen Beratern ist eine langfristige Konstanz oft nur schwer sicherzustellen. Schon die normale Personalfluktuation kann bei Microservices eine Herausforderung werden. Wenn niemand mehr einen Microservice warten kann, ist es immer noch möglich, den Microservice neu zu schreiben. Microservices sind aufgrund ihrer Größe einfach ersetzbar. Das ist natürlich mit Aufwand verbunden.
- Nur das Team versteht die Komponente. Wenn Teammitglieder kündigen, kann das Wissen über einen oder mehrere Microservices verloren gehen. Dann kann der Microservice nicht mehr geändert werden. Solche Wissensinseln müssten eigentlich vermieden werden. In diesem Fall hilft auch das Ersetzen des Microservice nichts,

weil dazu ein genaues Wissen der Fachlichkeit notwendig ist.

- Änderungen sind schwierig, wenn die Zusammenarbeit mehrerer Teams notwendig ist. Wenn ein Team alle Änderungen für ein Feature in den eigenen Microservices umsetzen kann, funktionieren die Architektur und die Skalierung der Entwicklung sehr gut. Wenn ein Feature aber auch einen anderen Microservice und damit ein anderes Team betrifft, muss das andere Team die Änderungen an dem betroffenen Microservice vornehmen. Das bedeutet nicht nur Kommunikation, sondern die notwendigen Änderungen müssen auch gegen die anderen Anforderungen des Teams priorisiert werden. Wenn die Teams nach Sprints arbeiten, kann das Team ohne Abbruch des Sprints frühestens im nächsten Sprint die Zulieferung leisten – das bedeutet eine erhebliche Verzögerung. Bei einer Springlänge von zwei Wochen kann der Verzug zwei Wochen betragen – wenn das Team die Änderungen so hoch priorisiert, dass sie im nächsten Sprint gemacht werden. Andernfalls kann es zu noch längeren Verzögerungen kommen.

Wenn nur jeweils das zuständige Team Änderungen an Collective Code Ownership einem Microservice durchführen darf, ergeben sich also durchaus einige Herausforderungen. Daher lohnt es sich, Alternativen in Betracht zu ziehen. Aus agilen Prozessen stammt das Konzept »Collective Code Ownership« (gemeinsamer Besitz des Codes). Bei diesem Konzept hat jeder Entwickler nicht nur das Recht, sondern sogar die Pflicht, den gesamten Code zu ändern – beispielsweise wenn er die Qualität des Codes an einer bestimmten Stelle für unzureichend hält. So kümmern sich alle Entwickler um die Qualität des Codes. Außerdem werden technische Entscheidungen besser kommuniziert, weil sie im Code von mehr Entwicklern nachvollzogen werden. Dabei werden die Entscheidungen kritisch hinterfragt, sodass die Qualität des Systems steigt.

Collective Code Ownership kann sich auf ein Team und seine Microservices beziehen. Da die Teams relativ frei in ihrer Organisation sind, ist ein solches Vorgehen ohne große Abstimmung möglich.

Aber im Prinzip können Teams auch Microservices ändern, die anderen Teams gehören. Dieses Vorgehen nutzen einige Microservice-Projekte, um mit den genannten Herausforderungen umzugehen. Denn es hat einige Vorteile:

Vorteile von Collective Code Ownership

- Änderungen an einem Microservice eines anderen Teams können schneller und einfacher umgesetzt werden. Wenn eine Änderung notwendig ist, muss die Änderung nicht durch das andere Team erfolgen. Stattdessen kann das anfordernde Team die Änderung selber durchführen. Es ist nicht mehr notwendig, die Änderung gegenüber anderen Änderungen in der Komponente zu priorisieren.
- Die Zusammenstellung der Teams ist flexibler. Die Entwickler kennen einen größeren Teil des Codes – zumindest oberflächlich durch Änderungen, die sie in dem Code vorgenommen haben. Das macht es einfacher, Teammitglieder oder gar ein ganzes Team zu ersetzen – oder ein Team aufzustocken. Die Entwickler müssen sich nicht ganz neu einarbeiten. Die Stabilität der Teams ist immer noch die beste Option – aber oft kann das nicht erreicht werden.

- Die Aufteilung in Microservices ist leichter änderbar. Wegen des breiteren Wissens der Entwickler ist es einfacher, Microservices zwischen den Teams anders aufzuteilen und zu verschieben. Das kann sinnvoll sein, wenn Microservices auf der Software-Architekturebene sehr eng zusammenhängen, aber von verschiedenen Teams verantwortet werden, die sich dann intensiv und aufwendig abstimmen müssen. Wird die Verantwortung für die Microservices so geändert, dass dasselbe Team eng zusammenhängende Microservices weiterentwickelt, ist die Abstimmung wesentlich einfacher, als das zwischen zwei Teams der Fall wäre. In einem Team sind die Teammitglieder oft in denselben Räumlichkeiten und können einfach direkt miteinander kommunizieren.

Aber es gibt auch Nachteile bei diesem Vorgehen:

Nachteile von Collective Code Ownership

- Collective Code Ownerships steht im Widerspruch zu der Technologiefreiheit: Wenn jedes Team andere Technologien nutzt, ist es für Entwickler außerhalb eines Teams schwierig, die Microservices zu ändern.
- Der Fokus der Teams kann verloren gehen. Die Entwickler bekommen zwar einen größeren Überblick – aber es ist vielleicht besser, wenn die Entwickler sich auf ihre eigenen Microservices konzentrieren können.
- Die Architektur ist nicht mehr ganz so fest. Durch die Kenntnis des Codes anderer Komponenten können Entwickler die Interna ausnutzen und so schnell Abhängigkeiten schaffen, die in der Architektur eigentlich nicht erwünscht sind. Schließlich soll eine Aufteilung der Teams nach dem Gesetz von Conway die Architektur unterstützen, indem Schnittstellen zwischen fachlichen Komponenten auch Schnittstellen zwischen Teams werden. Die Schnittstellen zwischen den Teams werden aber weniger wichtig, wenn jeder den Code jedes anderen Teams ändern kann.

Die Kommunikation zwischen den Teams ist immer noch notwendig: Schließlich hat das für den Microservice

Pull Requests zur Abstimmung

verantwortliche Team besonders viel Wissen über den Microservice, sodass Änderungen mit dem jeweiligen verantwortlichen Team abgestimmt werden sollten. Das kann technisch abgesichert werden: Die Änderungen der externen Teams können zunächst von anderen Änderungen getrennt vorgenommen und dann durch einen Pull Request an das für den Microservice zuständige Team geschickt werden. Pull Requests bündeln Änderungen an Source Code. Sie sind vor allem in der Open-Source-Community ein beliebter Ansatz, um externe Zulieferungen zu ermöglichen, ohne die Kontrolle über das Projekt abzugeben. Das verantwortliche Team kann den Pull Request annehmen oder Nachbesserungen verlangen. Damit gibt es ein formales Review des verantwortlichen Teams für jede Änderung. So kann das verantwortliche Team auch weiterhin den Microservice in eine bestimmte Richtung weiterentwickeln.

Da immer noch Kommunikation zwischen den Teams notwendig ist, wird das Gesetz von Conway durch dieses Vorgehen nicht verletzt. Es ist nur eine andere Spielweise. Bei einem schlechten Schnitt in einer Microservices-Architektur gibt es nur Optionen, die alle erhebliche Nachteile haben. Den Schnitt im Nachhinein zu korrigieren, ist schwierig. Größere Änderungen über Microservices hinweg sind nämlich aufwendig, wie [Abschnitt](#)

[8.4](#) gezeigt hat. Durch den ungeeigneten Schnitt sind die Teams gezwungen, viel miteinander zu kommunizieren, und dadurch entstehen Reibungsverluste. Die Aufteilung so zu lassen, wie sie ist, kann also auch nicht die Lösung sein. Mit Collective Code Ownership kann die Kommunikation eingeschränkt werden. Die Teams setzen direkt Anforderungen im Code anderer Teams um. So entstehen weniger Reibungsverluste. Dafür sollte die Technologiefreiheit eingeschränkt werden. Die Änderungen an den Microservices müssen nach wie vor koordiniert werden – mindestens Reviews sind definitiv notwendig. Wäre die Architektur aber geeignet aufgesetzt, wäre diese Maßnahme als Workaround gar nicht notwendig.

Selber ausprobieren und experimentieren

- Hast du schon Collective Code Ownership erlebt? Wie waren die Erfahrungen?
- Welche Beschränkungen gibt es in deinem aktuellen Projekt, wenn ein Entwickler irgendwelchen Code ändert, der von einem anderen Entwickler im selben Team oder von einem Entwickler in einem anderen Team entwickelt worden ist? Oder sind keine Änderungen am Code anderer Teams vorgesehen? Wie können dann notwendige Änderungen dennoch umgesetzt werden? Welche Probleme gibt es mit dem Vorgehen?

13.3 Spielräume schaffen: Mikro- und Makro-Architektur

Microservices erlauben einen weitgehenden Verzicht auf übergreifende Architekturentscheidungen. So kann jedes Team die aus seiner Sicht optimale Architektur für seine Microservices wählen.

Grundlage ist die Microservices-Architektur. Sie erlaubt viele technische Freiheiten. Während sonst schon aus technischen Gründen einheitliche Technologien notwendig sind, sind bei Microservices diese Zwänge nicht vorhanden. Aber Einheitlichkeit kann aus anderen Gründen wünschenswert sein. Die Frage ist, welche Entscheidung von wem getroffen wird. Bei den Entscheidungen kann man zwei Ebenen unterscheiden:

- Die *Makro-Architektur* sind die Entscheidungen, die das gesamte System betreffen. Das sind auf jeden Fall die in [Kapitel 8](#) dargestellten Entscheidungen über die fachliche Aufteilung und grundlegende Technologien, die alle Microservices nutzen müssen, sowie die Kommunikationsprotokolle ([Kap. 9](#)). Die Eigenschaften und Technologien einzelner Microservices können auch festgelegt werden ([Kap. 10](#)). Das ist aber nicht zwingend. Keine Entscheidung über die Interna der Microservices muss in der Makro-Architektur getroffen werden.
- Bei der *Mikro-Architektur* geht es um die Entscheidungen, die jedes Team selber treffen kann. Es sollten Belange sein, die nur die eigenen Microservices betreffen. Dazu können alle Aspekte aus [Kapitel 10](#) zählen, wenn sie nicht als Teil der Makro-Architektur bereits fest definiert sind.

Die Makro-Architektur muss nicht einmal definiert, sondern sie muss ständig weiterentwickelt werden. Neue Features können einen anderen fachlichen Schnitt

erfordern oder neue Technologien notwendig machen. Die Arbeit an der Makro-Architektur ist ein permanenter Prozess.

Die Frage ist, wer Makro- und Mikro-Architektur festlegt und weiterentwickelt. Wichtig ist, dass jede Entscheidung mit Verantwortung einhergeht. Wer die Entscheidung trifft, ist für eine fehlerhafte Entscheidung verantwortlich. Umgekehrt geht mit der Verantwortung für einen Microservice die Verpflichtung einher, Entscheidungen für den Microservice zu treffen. Wenn in der Makro-Architektur ein bestimmter Technologie-Stack vorgeschrieben wird, liegt die Verantwortung für ihn bei denjenigen, die diese Entscheidung gefällt haben – und nicht beim einzelnen Team, das gegebenenfalls die Probleme mit dem Technologie-Stack hat. Daher ist eine starke Einschränkung der Freiheiten der einzelnen Microservices durch die Makro-Architektur kontraproduktiv. So werden Entscheidungen und Verantwortung auf eine Ebene gehoben, die mit den einzelnen Microservices nicht mehr viel zu tun hat. So kann eine Elfenbeinturm-Architektur entstehen, die mit der Realität nicht mehr viel zu tun hat. Hat man Glück, wird sie ignoriert. Hat man Pech, führt sie zu ernsthaften Problemen in der Anwendung. Microservices ermöglichen den weitgehenden Verzicht auf Makro-Architekturentscheidungen, um so eine Elfenbeinturm-Architektur zu vermeiden.

Für die Definition der Makro-Architektur müssen Entscheidungen gefällt werden, die alle Microservices betreffen. Solche Entscheidungen kann kein Team alleine treffen, da die Teams nur jeweils für ihre Microservices verantwortlich sind. Die Entscheidungen in der MakroArchitektur gehen darüber hinaus.

Die Makro-Architektur kann durch ein Team festgelegt werden, das aus Mitgliedern der einzelnen Teams besteht. Dieser Ansatz scheint zunächst auf der Hand zu liegen: So können die Teams ihre Sichtweisen einbringen. Niemand diktiert den Teams bestimmte Ansätze. Die Teams werden bei Entscheidungen nicht übergangen. Es gibt viele Microservice-Projekte, die diesen Ansatz sehr erfolgreich nutzen.

Aber der Ansatz hat auch Nachteile:

- Für Entscheidungen auf der Makro-Architekturebene ist ein Überblick über das gesamte System notwendig und ein Interesse, das System als Ganzes weiterzuentwickeln. Mitglieder der einzelnen Teams haben oft einen Fokus auf die eigenen Microservices, was auch sehr sinnvoll ist, denn daran sollten sie auch gemessen werden. Dann können Mitglieder der Teams übergreifende Entscheidungen nur schwer treffen, weil es nicht zu ihrer Perspektive passt.
- Die Gruppe kann zu groß sein. Effektive Teams haben fünf, maximal zehn Mitglieder. Wenn es viele Teams gibt und jedes Team mit mindestens einem Mitglied an den Entscheidungen beteiligt werden soll, wird das Team für die übergreifenden Entscheidungen zu groß und kann nicht mehr effektiv arbeiten. Makro-Architektur zu definieren und zu pflegen ist inhaltliche Arbeit, zu der so große Teams kaum in der Lage sind.

Die Alternative ist ein Architekt oder ein Architekturteam, das nur für die Makro-Architektur zuständig ist. Bei einem größeren Projekt ist diese Aufgabe so umfassend, dass ein ganzes Architekturteam sich mit dieser Aufgabe beschäftigen muss. Dieses

Architekturteam hat die Sicht auf das Gesamtprojekt. Aber es gibt die Gefahr, dass das Architekturteam sich von der Arbeit der anderen Teams entfernt und dann weltfremde Entscheidungen trifft oder Probleme löst, die Teams gar nicht haben. Daher muss das Team eher moderieren und Standpunkte zwischen den Teams abgleichen, als selber eine bestimmte Richtung vorzugeben. Schließlich werden die Teams am Ende mit den Konsequenzen der Entscheidungen leben müssen.

Es gibt bei der Aufteilung in Mikro- und Makro-Architektur keine generelle Aussage. Die Kultur des Unternehmens, der Grad der Selbstbestimmung und andere organisatorische Gegebenheiten spielen eine Rolle. Eine sehr hierarchische Organisation wird den Teams weniger Freiheiten lassen. Wenn möglichst viele Entscheidungen auf die Ebene der Mikro-Architektur verschoben werden, bekommen die Teams mehr Verantwortung. Das hat oft positive Auswirkungen, weil die Teams sich dann tatsächlich zuständig fühlen und entsprechend agieren. Beispielsweise war das NUMMI-Autowerk in den USA ein sehr unproduktives Werk mit Drogenmissbrauch und Sabotage. Durch einen stärkeren Fokus auf Teamwork und Vertrauen konnten dieselben Arbeiter zu einem der produktivsten Standorte werden [2]. Wenn die Teams mehr entscheiden und verantworten, beeinflusst es Klima und Produktivität positiv.

Außerdem wird durch die Delegation in die Teams weniger Zeit mit Koordination verbracht, sodass die Teams produktiver arbeiten können. Kommunikation zu vermeiden, indem mehr Entscheidungen in die Teams und damit in die Mikro-Architektur delegiert werden, ist ein wesentlicher Punkt für die Skalierung der Architektur.

Wenn hingegen die Teams sehr stark eingeschränkt werden, wird ein Hauptvorteil der Microservices nicht realisiert. Microservices erhöhen die technische Komplexität des Systems. Das ist nur sinnvoll, wenn die Vorteile der Microservices auch ausgenutzt werden. Nach einer Entscheidung für Microservices sollte daher konsequenterweise eine Entscheidung für möglichst viel Mikro-Architektur und möglichst wenig Makro-Architektur getroffen werden.

Die Entscheidung für mehr oder weniger Makro-Architektur kann für jeden Bereich unterschiedlich getroffen werden.

Bei den Technologien ergibt sich folgendes Bild:

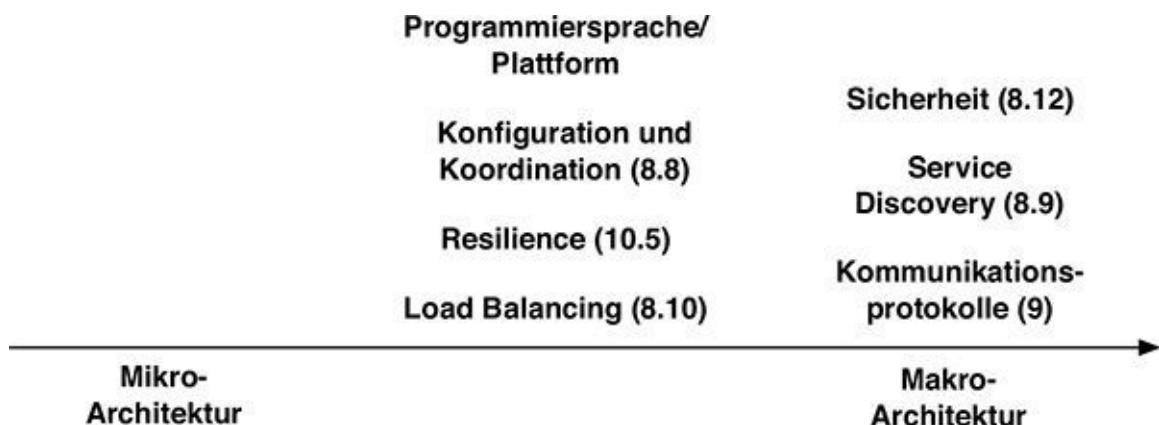
Technologie: Makro-/Mikro-Architektur

- Einheitliche *Sicherheit* ([Abschnitt 8.12](#)), *Service Discovery* ([Abschnitt 8.9](#)) und *Kommunikationsprotokolle* ([Kap. 9](#)) sind notwendig, damit die Microservices miteinander kommunizieren können. Die Entscheidungen in diesem Bereich liegen klar in der Makro-Architektur. Dazu gehören zum Beispiel auch die Entscheidungen für die Nutzung und genaue Ausgestaltung abwärtskompatibler Schnittstellen, ohne die ein getrenntes Deployment nicht möglich ist.
- *Konfiguration und Koordination* ([Abschnitt 8.8](#)) müssen nicht zwangsläufig übergreifend geregelt werden. Wenn jeder Microservice von dem jeweiligen Team betrieben wird, kann auch die Konfiguration von dem Team erstellt werden und dazu kann jedes Team ein eigenes Werkzeug nutzen. Ein einheitliches Werkzeug hat aber deutliche Vorteile. Außerdem gibt es wohl kaum einen sinnvollen Grund, warum

jedes Team einen anderen Mechanismus nutzen muss.

- Die Nutzung von *Resilience* ([Abschnitt 10.5](#)) oder *Load Balancing* ([Abschnitt 8.10](#)) kann in der Makro-Architektur festgelegt werden. Entweder kann die Makro-Architektur eine bestimmte Standardtechnologie definieren oder nur, dass diese Punkte bei der Implementierung von Microservices beachtet werden müssen. Das kann beispielsweise durch Tests abgesichert werden ([Abschnitt 11.8](#)). Die Tests können überprüfen, ob ein Microservice bei einem Ausfall eines abhängigen Microservice immer noch verfügbar ist. Und sie können überprüfen, ob die Last auf mehrere Microservices verteilt wird. Die Entscheidung über die Nutzung von Resilience oder Load Balancing kann den Teams theoretisch freigestellt werden. Wenn sie für die Verfügbarkeit und die Performance des Service verantwortlich sind, muss man ihnen die Freiheit geben, beliebige technologische Ansätze dazu zu nutzen. Wenn ohne Resilience und Load Balancing ihre Microservices ausreichend verfügbar sind, ist das akzeptabel. Allerdings sind solche Szenarien wohl kaum denkbar.
- Bei der *Plattform* und *Programmiersprache* kann die Entscheidung auf Ebene der Makro- oder Mikro-Architektur getroffen werden. Die Wahl kann neben den Teams auch den Betrieb beeinflussen, da der Betrieb die technologische Basis verstehen und auch bei Ausfällen beherrschen muss. Es muss nicht unbedingt eine Programmiersprache festgelegt werden, sondern die Technologie kann auf die JVM (Java Virtual Machine) eingeschränkt sein, die verschiedene Programmiersprachen unterstützt. Bei der Plattform kann ein Kompromiss sein, dass eine bestimmte Datenbank vom Betrieb bereitgestellt wird, andere aber von den Teams genutzt und betrieben werden können. Ob die Makro-Architektur Plattform und Programmiersprache vorgibt, hängt davon ab, ob Entwickler zwischen Teams wechseln sollen. Eine gemeinsame Plattform erleichtert es, die Verantwortung für Microservices von einem Team zu einem anderen zu verschieben.

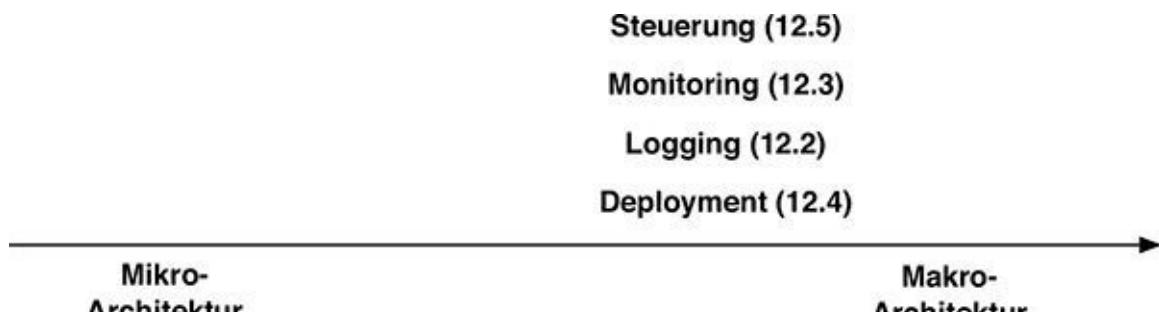
Abb. 13–3 Technologie: Makro-/Mikro-Architektur



Im Bereich des Betriebs gibt es die Steuerung ([Abschnitt 12.5](#)), Monitoring ([Abschnitt 12.3](#)), Logging ([Abschnitt 12.2](#)) und Deployment ([Abschnitt 12.4](#)). Um die Komplexität der Umgebung zu reduzieren und einen einheitlichen Betrieb zu ermöglichen, müssen die Lösungen in diesem Bereich in der Makro-Architektur vorgegeben werden. Ähnliches gilt für Plattform und Programmiersprache. Zwingend ist das aber nicht: Wenn der vollständige Betrieb der Microservices in der Hand der Teams liegt, kann jedes Team theoretisch eine eigene Technologie für jeden dieser Bereiche nutzen. Allerdings ergeben sich dadurch kaum

Vorteile, aber dafür eine große technologische Komplexität. Es ist aber zum Beispiel möglich, dass die Teams eigene Sonderlösungen für bestimmte Aufgaben nutzen. Wenn beispielsweise der Umsatz anders in das Monitoring für die Geschäftsseite übertragen werden soll, ist das sicher denkbar.

Abb. 13–4 Betrieb: Makro-/Mikro-Architektur



Bei der fachlichen Architektur ist die Verteilung der Fachlichkeiten auf die Teams ein Teil der Makro-Architektur ([Abschnitt 8.1](#)). Sie beeinflusst nicht nur die Architektur, sondern auch welche Teams für welche Fachlichkeiten verantwortlich sind. Daher kann die Aufgabe nicht in die Mikro-Architektur verschoben werden. Die fachliche Architektur der einzelnen Services dagegen ([Abschnitt 10.1, 10.2, 10.3](#) und [10.4](#)) muss den Teams überlassen werden. Den Teams die fachliche Architektur der einzelnen Microservices vorzuschreiben, würde bedeuten, dass die Microservices auf organisatorischer Ebene wie ein Monolith behandelt werden, weil die gesamte Architektur zentral koordiniert wird. Dann kann man aber auch gleich ein Deployment-Monolith entwickeln, der technisch einfacher ist. Eine solche Entscheidung wäre nicht sinnvoll.

Abb. 13–5 Fachliche Architektur: Makro-/Mikro-Architektur



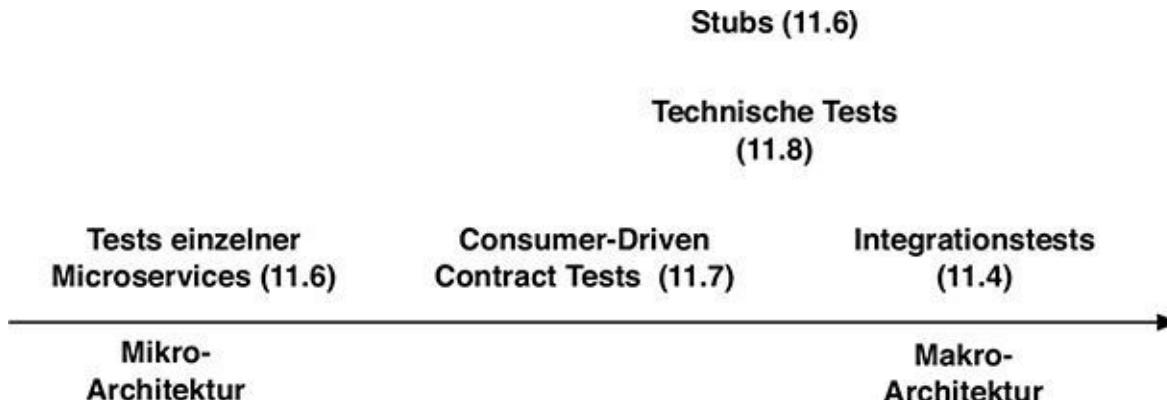
Im Bereich Tests sind die Integrationstests ([Abschnitt 11.4](#)) auf Ebene der Makro-Architektur angesiedelt.

Konkret muss entschieden werden, ob es für eine bestimmte Fachlichkeit einen Integrationstest geben soll und wer ihn implementiert. Integrationstests sind nur sinnvoll, wenn sie teamübergreifende Funktionen betreffen. Alle anderen Funktionalitäten kann ein Team alleine testen. Integrationstests müssen daher über Teamgrenzen hinweg global koordiniert werden. Die technischen Tests ([Abschnitt 11.8](#)) kann die Makro-Architektur den Teams vorschreiben. Sie sind eine gute Möglichkeit, um globale Standards und technische Bereiche der Makro-Architektur durchzusetzen und zu überprüfen. Consumer-Driven Contract Tests ([Abschnitt 11.7](#)) und Stubs ([Abschnitt 11.6](#)) können die Teams untereinander gegebenenfalls abstimmen. Eine gemeinsame technologische Grundlage als Teil der Makro-Architektur kann die Entwicklung wesentlich vereinfachen. Einheitliche Technologien sind in diesem Bereich besonders sinnvoll, weil ein Team die CDCs und Stubs der anderen Teams nutzen muss. Wenn dafür nur eine Technologie genutzt wird, erleichtert das die Arbeit deutlich. Die Technologien müssen aber nicht zwingend in der

Makro-Architektur fest vorgegeben werden.

Die Tests der einzelnen Microservices sollten den Teams freigestellt werden, denn sie tragen auch die Verantwortung für die Qualität der Microservices.

Abb. 13–6 Tests: Makro-/Mikro-Architektur



In vielen Bereichen können Entscheidungen entweder auf Ebene der Makro- oder Mikro-Architektur getroffen werden. Ziel von Microservices-Architekturen ist die größtmögliche Unabhängigkeit der Teams. Daher sollten möglichst viele Entscheidungen auf die Mikro-Architekturebene und damit auf die Teams verteilt werden. Allerdings kann man bei den Betriebsaspekten die Frage stellen, ob die Teams durch eigene unterschiedliche Werkzeuge einen Vorteil haben. Wahrscheinlich wird der Technologie-Zoo nur größer. In diesem Bereich gibt es einen Zusammenhang mit DevOps ([Abschnitt 13.5](#)). Abhängig von dem Grad der Kooperation zwischen Entwicklern und Betrieb kann es unterschiedliche Freiheitsgrade geben. Bei einer klaren Trennung zwischen Entwicklung und Betrieb wird der Betrieb in der Makro-Architektur viele Vorgaben festschreiben. Schließlich muss der Betrieb am Ende die Microservices in Produktion betreuen. Wenn alle Microservices einheitliche Technologien nutzen, wird diese Aufgabe einfacher.

Bei der Definition von Programmiersprache und Plattform gilt es ebenfalls, die Vorteile spezialisierter Technologie-Stacks gegen die Nachteile von heterogenen Technologien im Gesamtsystem abzuwägen. Den Stack den Teams vorzuschreiben, kann genauso sinnvoll sein, wie den Teams die Wahl der Technologie freizustellen. Ein einheitlicher Technologie-Stack kann den Betrieb vereinfachen und es Entwicklern eher erlauben, zwischen Microservices und Teams zu wechseln. Spezialisierte Technologie-Stacks ermöglichen eine bessere Reaktion auf besondere Herausforderungen und motivieren Mitarbeiter, die so aktuelle Technologien nutzen können.

Ob ein Microservice die Makro-Architektur tatsächlich einhält, kann ein Test entscheiden (siehe [Abschnitt 11.8](#)). Dieser Test kann ein Artefakt sein, das ebenfalls Teil der Makro-Architektur ist. Die Gruppe, die für die Makro-Architektur zuständig ist, kann mit diesem Artefakt die Makro-Architektur eindeutig definieren. Das ermöglicht, dass alle Microservices auf die Einhaltung der Makro-Architektur überprüft werden.

13.4 Technische Führung

Die Aufteilung in Mikro- und Makro-Architektur ändert das Verhalten der technischen Führung komplett und ist ein wesentlicher Vorteil von Microservices. Die Makro-

Architektur definiert die technischen Pflichten und Freiheiten. Mit den Freiheiten geht auch die Verantwortung für die jeweiligen Entscheidungen einher.

Beispielsweise kann eine Datenbank vorgeschrieben werden. Dann kann das Team die Verantwortung für die Datenbank an die technische Führung delegieren. Wäre es in der Mikro-Architektur, würde das Team sie betreiben, da es die Entscheidung für eine Technologie fällt. Die Konsequenzen können keinem anderen Team aufgebürdet werden (siehe [Abschnitt 8.7](#)). Wer die Entscheidung trifft, hat auch die Verantwortung. Die technische Führung kann solche Entscheidungen sicher treffen, aber entbindet die Teams dann von der Verantwortung und damit von der Selbstständigkeit.

Mit den größeren Freiheiten geht eine größere Verantwortung einher. Damit müssen die Teams auch umgehen können und die Freiheiten auch wollen. Leider ist das nicht immer der Fall. Das kann für mehr Makro-Architektur sprechen oder für organisatorische Fortentwicklung, die in mehr Selbstorganisation und damit weniger Makro-Architektur münden. Ziel der technischen Führung ist es, den Abbau der Makro-Architektur zu ermöglichen und einen Prozess hin zu mehr Selbstorganisation zu moderieren.

Noch radikaler bezüglich der Freiheiten der Teams ist Developer Anarchy der Ansatz Developer Anarchy [2][3]. Er legt die komplette Verantwortung in die Hände der Entwickler. Sie können nicht nur Technologien frei wählen, sondern sogar Code neu schreiben, wenn sie das für notwendig halten. Außerdem kommunizieren sie direkt mit den Anforderern. Dieser Ansatz wird in einem sehr stark wachsenden Unternehmen genutzt und funktioniert dort sehr gut. Dahinter steckt Fred George, der auf 40 Jahre Berufserfahrung bei Unternehmen wie IBM oder Thoughtworks zurückblicken kann und zu den Pionieren agiler Ansätze gehört. In diesem Modell werden die Makro-Architektur und Deployment-Monolithen abgeschafft, sodass die Entwickler tun, was sie für richtig halten. Dieser Ansatz ist sehr radikal und zeigt, wie weit man gehen kann.

Selber ausprobieren und experimentieren

In den [Abbildungen 13–3, 13–4, 13–5](#) und [13–6](#) sind jeweils Bereiche bezeichnet, die zur Mikro- oder Makro-Architektur gehören können. Das sind jene Elemente, die in den Abbildungen in der Mitte stehen. Gehe diese Elemente durch und entscheide, ob du sie in der Mikro- oder Makro-Architektur festlegen würdest. Wichtig ist vor allem die Begründung für die eine oder andere Alternative. Beachte, dass Entscheidungen auf der Mikro-Architekturebene eher der Microservices-Idee entsprechen.

13.5 DevOps

DevOps bezeichnet das Zusammenwachsen von Entwicklung (Dev) und Betrieb (Ops) zu einer Einheit (DevOps). In erster Linie ist es eine organisatorische Änderung: Jedes Team hat Entwickler und Betriebsexperten. Sie arbeiten zusammen, um einen Microservice zu entwickeln und zu betreiben. Das bedingt ein anderes Mindset, denn Betriebsthemen sind Entwicklern oft fremd, während Betriebler oft die Arbeit in Projekten nicht kennen, sondern Systeme unabhängig von den Projekten betreiben. Dabei gleichen sich die

technischen Skills an: Betrieb arbeitet mehr an Automatisierung und den passenden Tests für die Automatisierung – und das ist letztendlich Software-Entwicklung. Monitoring, Log-Analyse oder Deployment werden gleichzeitig mehr und mehr auch ein Thema für Entwickler.

DevOps und Microservices ergänzen sich ideal:

DevOps und Microservices

- Die Teams können nicht nur die Entwicklung, sondern auch den Betrieb der Microservices übernehmen. Dazu muss Wissen aus dem Bereich Betrieb und Entwicklung in den Teams vorhanden sein.
- Die Orientierung der Teams an fachlichen Features und Microservices stellt eine sinnvolle organisatorische Alternative zu der Aufteilung in Betrieb und Entwicklung dar.
- Die Kommunikation zwischen Betrieb und Entwicklung wird einfacher, wenn Mitglieder beider Bereiche in einem Team zusammenarbeiten. Die Kommunikation innerhalb eines Teams ist einfacher als die über Teamgrenzen. Das entspricht dem Ziel von Microservices, die Koordination zu reduzieren.

DevOps und Microservice passen sehr gut zusammen. Genau genommen kann das Ziel, dass Teams Microservices bis in Produktion deployen und sie auch in Produktion überwachen, nur mit DevOps-Teams erreicht werden. Nur so können die Teams das notwendige Know-how aus beiden Bereichen haben.

DevOps ist eine so grundlegende Änderung in der Organisation, dass viele Unternehmen den Schritt noch scheuen. Daher ist die Frage, ob Microservices auch ohne DevOps umgesetzt werden können. Das ist machbar:

Erzwingen Microservices DevOps?

- Durch die Makro-/Mikro-Architektur-Aufteilung kann der Betrieb Standards definieren. Zu der Makro-Architektur können technische Elemente wie Logging, Monitoring oder Deployment gehören. Wenn die Standards eingehalten werden, kann der Betrieb die Software übernehmen und in die Betriebsprozesse eingliedern.
- Ebenso können die Plattform und Programmiersprache so weit vorgegeben werden, wie das für den Betrieb notwendig ist. Wenn der Betrieb sich nur zutraut, Java-Anwendungen auf einem Tomcat zu betreiben, kann das als Plattform festgeschrieben werden. Ähnliches gilt für Infrastrukturelemente wie Datenbanken oder Messaging-Systeme.
- Ebenso kann es organisatorische Anforderungen geben: Beispielsweise kann der Betrieb fordern, dass Mitarbeiter aus den Teams zu bestimmten Zeiten erreichbar sind, sodass Probleme in die Teams eskaliert werden können. Konkret: Wer selber deployen will, muss seine Telefonnummer hinterlegen und wird bei einem Problem auch nachts angerufen. Wird der Anruf nicht beantwortet, kann als Nächstes das Management angerufen werden. Das erhöht die Chance, dass Entwickler die Anrufe auch wirklich entgegennehmen.

Die Teams können in einem solchen Kontext nicht mehr dafür zuständig sein, alle Microservices bis in die Produktion zu bringen. Der Zugriff und die Verantwortung liegen beim Betrieb. Es muss eine Stelle in der Continuous-Delivery-Pipeline geben, an der die

Microservices an den Betrieb übergeben und dann in Produktion ausgerollt werden. An der Stelle geht der Microservice dann in die Verantwortung des Betriebs über, der sich mit dem Team über den Microservice abstimmen wird. Die typische Stelle für die Übergabe an den Betrieb ist nach den Testphasen, gegebenenfalls vor den explorativen Tests. Der Betrieb ist mindestens für die letzte Phase zuständig, also das Ausrollen in Produktion. Der Betrieb kann zu einem Flaschenhals werden, wenn sehr viele geänderte Microservices in Produktion gebracht werden müssen.

Insgesamt haben DevOps und Microservices Synergien – aber es ist nicht unbedingt notwendig, für Microservices auch DevOps einzuführen.

Wenn Microservices auf klassische IT-Organisationen treffen

von Alexander Heusingfeld, innoQ Deutschland GmbH

Das Thema »Microservices« ist mittlerweile in zahlreichen IT-Abteilungen angekommen und wird dort diskutiert. Interessanterweise werden Initiativen zur Einführung oft von Führungskräften aus dem mittleren Management initiiert. Jedoch macht man sich häufig zu wenige Gedanken, welche Auswirkungen ein Microservices-Architekturstil auf die (IT-)Organisation eines Unternehmens hat. Deshalb möchte ich hier von einigen »Überraschungen« berichten, die ich bei der Einführung eines solchen Architekturstils erlebt habe.

Pets vs. Cattle

»Pets vs. Cattle« ist ein Slogan, der zu Beginn der DevOps-Bewegung einige Bekanntheit erlangt hat [1]. Die Grundaussage ist, dass Server in Zeiten von Cloud und Virtualisierung nicht wie ein Haustier, sondern wie eine Herde von Tieren behandelt werden sollten. Ist ein Haustier krank, tut man alles, um es wieder gesund zu pflegen. Kranke Herdentiere hingegen tötet man, um die Herde nicht zu gefährden.

Es geht also darum, die Personifizierung von Servern zu vermeiden – etwa durch die Vergabe von Eigennamen (wie Leviathan, Pollux, Berlin oder Lorsch). Gibt man seinen Servern derartige »Haustier«-Namen, tendiert man dazu, diesen auch Haustierpflege wie individuelle Updates, Skriptanpassungen oder sonstige einmalige Änderungen zukommen zu lassen. Dies ist aber bekanntlich negativ für die Wiederholbarkeit von Installationen. Gerade bei Auto-Skalierung und Failover-Szenarien, wie sie von Microservices-Architekturen gefordert sind, ist das ein K.o.-Kriterium.

Eines meiner Projekte begegnete diesem Problem auf sehr interessante Art: Zwar hatten die Server und VMs immer noch Eigennamen, die Verwaltung dieser Systeme war jedoch vollständig über Puppet automatisiert. Puppet lud die entsprechenden Skripte aus einem SVN-Repository. In diesem Repository lagen individuelle Skripte für jeden Server. Dieses Szenario könnte man »Handpuppen zur Automatisierung der Haustierpflege« nennen. Hierdurch gewinnt man, dass ausgefallene Server schnell durch exakte Kopien ihrer selbst ersetzt werden können.

Anforderungen an Skalierbarkeit werden aber in keiner Weise berücksichtigt, denn es kann immer nur eine Instanz des Haustier-Servers mit dem Namen *Leviathan* geben. Eine Alternative ist, auf parametrisierte Skripte zu wechseln und Templates wie beispielsweise »Production-VM for app XYZ« zu nutzen. So ermöglicht man

gleichzeitig auch flexiblere Deployment-Szenarien wie Blue-Green-Deployments. Dann ist nicht mehr relevant, ob die VM *app-xyz-prod08.zone1.company.com* oder *app-xyz-prod045.zone1.company.com* die Arbeit erledigt. Es geht nur darum, dass jederzeit acht Instanzen dieser VM verfügbar sind und zu Lastspitzen weitere Instanzen hochgefahren werden können. Wie diese Instanzen heißen, ist egal.

[1] <http://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds>

Wir und Ihr

- »Alarming ist unsere Sache!«
- »Das kann euch egal sein!«
- »Das geht euch nichts an, das ist unser Bereich!«

Solche Sätze höre ich leider häufiger in sogenannten Cross-Functional Teams. Das sind Teams, die aus Architekten, Entwicklern, Testern und Betriebelern zusammengesetzt wurden. Gerade wenn die beteiligten Personen früher in anderen, rein funktionalen Teams im gleichen Unternehmen gearbeitet haben, werden – oft unterbewusst – alte Grabenkriege und Vorurteile mit ins neue Team gebracht. Es ist deshalb wichtig, sich der sozialen Aspekte von Anfang an bewusst zu sein und diesen proaktiv zu begegnen. Beispielsweise habe ich positive Erfahrungen damit gemacht, solche Teams in den ersten zwei bis vier Wochen im gleichen Büro zusammen arbeiten zu lassen. So können sich die Menschen im Team gegenseitig kennenlernen, ihre Körpersprache, ihren Charakter und Humor unmittelbar erfahren. Das vereinfacht im späteren Projektverlauf die Kommunikation signifikant, Missverständnisse werden vermieden.

Auch Teambuilding-Maßnahmen in den ersten Wochen, bei denen sich die Teammitglieder zwangsweise aufeinander verlassen müssen, können dabei helfen, das Eis zu brechen, die Stärken und Schwächen der Einzelnen einschätzen zu können und Vertrauen im Team aufzubauen und zu stärken. Werden diese Punkte vernachlässigt, macht sich das im weiteren Projektverlauf deutlich bemerkbar. Denn Menschen, die sich nicht mögen oder nicht vertrauen, werden sich, wenn auch nur unterbewusst, nicht aufeinander verlassen. Und das bedeutet, sie arbeiten nicht wirklich 100 % als Team.

Entwicklung vs. Test vs. Betrieb: Perspektivenwechsel

In vielen Unternehmen gibt es Initiativen für einen Perspektivenwechsel. Hier dürfen beispielsweise Mitarbeiter aus dem Vertrieb einen Tag im Einkauf mitarbeiten, um die Personen und die Prozesse dort kennenzulernen. Man verspricht sich hiervon, dass die Mitarbeiter ein Verständnis für ihre Kollegen entwickeln und dies in ihre tägliche Arbeit einfließen lassen, sodass Prozesse besser harmonisieren. Das Motto ist: »Auf der >anderen Seite< lernt man neue Perspektiven kennen!«

Auch in der IT kann ein solcher Perspektivenwechsel Vorteile bringen. Ein Entwickler kann beispielsweise eine neue Perspektive auf die Use- bzw. Test-Cases gewinnen, was ihn bei der Entwicklung dazu veranlassen kann, eine Modularisierung durchzusetzen, die leichter testbar ist. Oder er denkt bereits während der Entwicklung daran, welche Kriterien er braucht, um die Software in Produktion besser überwachen oder Fehler finden zu können. Einem Betriebler kann ein tieferer Einblick in die internen Abläufe der Anwendung dabei helfen, ein Verständnis zu entwickeln, mit dem

er gezielteres, effizienteres Monitoring aufsetzen kann. Jede Perspektive, die von der eigenen abweicht, kann Fragen aufwerfen, an die vorher in diesem Abschnitt des Lebenszyklus einer Anwendung niemand gedacht hätte. Und so etwas hilft am Ende dem Team, bessere Software zu liefern.

Für Ops gibt es nie eine »ganz grüne Wiese«

Sicherlich sind Microservices ein aktuelles Thema und bringen neue Technologien, Konzepte und organisatorische Veränderungen. Was man dabei aber nicht außer Acht lassen darf, ist, dass Unternehmen, die Microservices einführen, fast nie bei null anfangen! Es gibt immer irgendwelche Legacy-Systeme oder ganze IT-Landschaften, die bereits vorhanden sind und besser nicht als Big Bang abgelöst werden. Und diese Legacy-Systeme müssen im Zweifelsfall in die schöne neue Welt der Microservices integriert werden, zumindest aber koexistieren.

Aus diesem Grunde ist es wichtig, diese Systeme bei der Planung der Microservices-Architektur gerade im Bezug auf IT-Kosten zu berücksichtigen. Kann die bestehende Hardware-Infrastruktur wirklich für die Microservices umgebaut werden oder verlässt sich ein Bestandssystem auf genau diese Infrastruktur? Dies sind oft Fragen, die beim Infrastruktur-Operations-team hängen bleiben – falls es eine derartige Organisationseinheit im Unternehmen gibt. Andernfalls kann es auch passieren, dass diese Fragen erst auftreten, wenn das erste Deployment Richtung Systemtest oder Produktion erfolgen soll. Gerade um diese Fragen möglichst frühzeitig zu erkennen, rate ich dazu, die Deployment-Pipeline möglichst früh im Reorganisationsprojekt anzugehen. Die Deployment-Pipeline sollte schon stehen, bevor die erste fachliche Funktionalität von den Teams implementiert wird. Hierzu reicht oft schon ein simples Hello World, das dann vom ganzen Team mit vereinten Kräften Richtung Produktion gebracht wird. Fast immer stößt das Team dabei auf offene Fragen, die im schwersten Fall Auswirkungen auf das Design der Systeme haben. Da aber noch so gut wie nichts implementiert ist, sind derartige Änderungen früh in der Projektphase eben noch vergleichsweise günstig.

Fazit

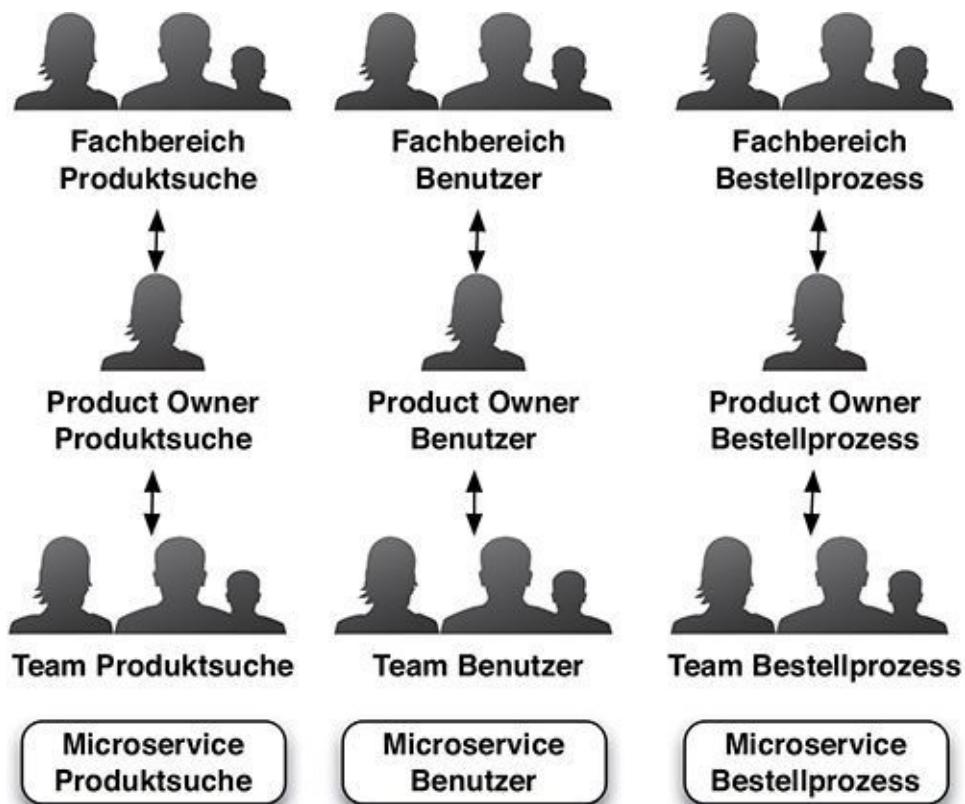
Die organisatorischen Veränderungen (vgl. »Conway's Law«), welche die Einführung von Microservices mit sich bringen, werden bisher oft unterschätzt. Alte Gewohnheiten, Vorurteile und eventuell sogar Grabenkämpfe stecken oft noch in den Köpfen. Gerade dann, wenn die neuen Teamkollegen früher in unterschiedlichen Abteilungen waren. »One Team« muss aber mehr als nur ein Buzzword sein. Schafft es das Team, die Vorurteile zu begraben und die unterschiedlichen Erfahrungen zu nutzen, kann es gemeinsam weiterkommen. Alle müssen verstehen, dass man nun gemeinsam die Aufgabe und Verantwortung hat, für den Kunden eine stabile Software in Produktion zu bringen. Hierbei kann jeder von den Erfahrungen des anderen profitieren, wenn man der Prämisse folgt: »*Jeder erklärt seine Bedenken, aber wir lösen das gemeinsam.*

13.6 Schnittstelle zu den Fachbereichen

Um zu gewährleisten, dass die Entwicklung tatsächlich auf mehrere Teams und Microservices skaliert werden kann, muss jedes Team einen eigenen Product Owner haben. Entsprechend des Scrum-Ansatzes ist er dafür zuständig, den Microservice fachlich weiterzuentwickeln. Dazu definiert er Stories, die im Microservice implementiert werden. Der Product Owner ist die Quelle aller Anforderungen und gibt die Priorisierung vor. Das ist besonders einfach, wenn ein Microservice dem entspricht, wofür auf der Geschäftsseite nur ein Fachbereich zuständig ist (Abb. 13–7). Üblicherweise wird dieses Ziel dadurch erreicht, dass die Microservices und die Teams sich der Organisation der Fachbereiche angleichen. Jeder Fachbereich bekommt »seinen« Product Owner und damit »sein« Team und »seine« Microservices.

Wenn die fachliche Architektur der Microservices gut ist, dann ist eine unabhängige Entwicklung der Microservices möglich. Schließlich soll jede Fachlichkeit genau in einem Microservice implementiert sein und die Fachlichkeit sollte auch nur einen Fachbereich interessieren. Die Architektur muss die Organisation der Fachbereiche bei der fachlichen Aufteilung in Microservices beachten, um zu gewährleisten, dass jeder Fachbereich eigene Microservices hat.

Abb. 13–7 Fachbereich, Product Owner und Microservices



Leider ist die Architektur aber oft nicht perfekt. Und außerdem haben Microservices Schnittstellen – ein Hinweis darauf, dass Funktionalitäten sich über mehrere Microservices erstrecken. Wenn also mehrere Funktionalitäten einen Microservice betreffen und daher mehrere Fachbereiche Einfluss auf einen Microservice nehmen wollen, muss der Product Owner eine zwischen den Fachbereichen abgestimmte Priorisierung sicherstellen. Das kann eine Herausforderung sein, weil die Fachbereiche unterschiedliche Prioritäten haben können. Dann muss der Product Owner zwischen den Fachbereichen vermitteln.

Nehmen wir an, dass es einen Fachbereich gibt, der in einem E-Commerce-Shop Sonderaktionen betreut. Er startet eine Aktion, bei der Bestellungen mit einer bestimmten Ware billiger versendet werden sollen. Die dafür notwendigen Änderungen betreffen das Bestellungsteam: Es muss herausfinden, ob die Bestellung eine solche Ware enthält. Diese Information muss sie dem Lieferungs-Microservice übergeben, der die Kosten für die Lieferung ermittelt. Also müssen die Product Owner dieser beiden Teams die Änderungen gegenüber den Änderungen der Fachbereiche priorisieren, die für Lieferungen und Bestellung zuständig sind. Leider kombinieren viele der Sonderaktionen verschiedene Funktionalitäten, sodass eine solche Priorisierung oft notwendig ist. Die Fachbereiche für Bestellung und Lieferung haben also ihre eigenen Microservices, während der Fachbereich für Sonderaktionen keine eigenen Microservices hat, sondern die Features in die anderen Microservices einbringen muss.

Die Architektur der Microservices kann sich also aus Architektur ergibt Fachbereiche der Organisation der Fachbereiche ergeben. Es gibt aber Fälle, in denen sich rund um ein IT-System ein Fachbereich gründet, der dieses System auf Geschäftsseite betreut. In einem solchen Fall kann man davon sprechen, dass die Microservices-Architektur die Organisation beeinflusst. Wenn es beispielsweise einen elektronischen Marktplatz gibt, der durch ein IT-System implementiert ist, kann ein Fachbereich entstehen, der diesen Marktplatz weiterentwickelt. Der Fachbereich wird das IT-System fachlich und aus einer Geschäftssicht weiterentwickeln. In diesem Fall ist zunächst der Marktplatz entwickelt worden und dann hat der Fachbereich sich gegründet. Also hat die Architektur des Systems die Organisation in Fachbereiche vorgegeben.

13.7 Wiederverwendbarer Code

Code wiederzuverwenden ist eigentlich ein technisches Problem. [Abschnitt 8.3](#) hat schon die Herausforderungen dargestellt, die entstehen, wenn zwei Microservices eine gemeinsame Bibliothek verwenden: Wenn die Microservices die Bibliothek so nutzen, dass ein neues Release der Bibliothek auch ein neues Deployment der Microservices erzwingt, ist das Ergebnis eine Deployment-Abhängigkeit – und die gilt es zu vermeiden, um auch weiterhin ein eigenständiges Deployment zu ermöglichen. Zusätzlicher Aufwand entsteht, weil die Teams der Microservices Änderungen an der Bibliothek koordinieren müssen. Neue Features für die verschiedenen Microservices müssen priorisiert und entwickelt werden. Auch das sind Abhängigkeiten zwischen den Teams, die man eigentlich vermeiden will.

Client Libraries, die den Aufruf eines Microservice kapseln, können akzeptabel sein. Wenn die Schnittstellen der Microservices abwärtskompatibel sind, muss die Client Library bei einer neuen Version des Microservice nicht durch eine neue Version ersetzt werden. Dann sind Client Libraries problemlos. Wenn die Client Library aber auch Domänenobjekte enthält, kann es problematisch werden. Wenn ein Microservice das Domänenmodell ändern will, muss das Team die Änderung mit den anderen Nutzern der Client Library koordinieren und kann daher nicht mehr unabhängig entwickeln. Die Grenzen zwischen einer vereinfachten Nutzung der Schnittstelle, die sinnvoll sein kann, und einer gemeinsamen Modellierung von Logik oder anderen Deployment-Abhängigkeiten, die problematisch sind, ist

fließend. Eine Lösung kann sein, gemeinsamen Code vollständig zu verbieten.

Offensichtlich können Projekte aber Code wiederverwenden. Kein Projekt kommt mehr ohne die eine oder andere Open-Source-Bibliothek aus. Die Nutzung dieses Codes ist offensichtlich einfach und hilft den Teams weiter. Probleme wie bei der Wiederverwendung von Code zwischen Microservices sind nicht zu befürchten.

Dafür gibt es verschiedene Gründe:

- Die Open-Source-Projekte haben eine hohe Qualität. Entwickler aus verschiedenen Unternehmen nutzen den Code und finden dadurch Fehler. Oft beseitigen sie die Fehler sogar, sodass die Qualität ständig steigt. Quellcode zu veröffentlichen und damit einen Einblick in die Iterra zu geben, ist oft schon Motivation genug, um die Qualität zu erhöhen.
- Die Dokumentation erlaubt einen Einstieg in die Nutzung, ohne dass man mit den Entwicklern direkt kommunizieren muss. Ohne eine gute Dokumentation finden Open-Source-Projekte kaum genügend Nutzer oder gar weitere Entwickler, weil der Einstieg viel zu aufwendig ist.
- Es gibt eine koordinierte Entwicklung mit einem Bug Tracker und einem Prozess zur Akzeptanz von Codeänderungen externer Entwickler. So sind Fehler und ihre Bearbeitung nachvollziehbar. Ebenso ist klar, wie Änderungen von außen in die Code-Basis eingebaut werden können.
- Und es ist bei einer neuen Version der Open-Source-Bibliothek nicht notwendig, dass alle Nutzer die neue Version verwenden. Die Abhängigkeiten zu der Bibliothek sind nicht so ausgeprägt, dass es eine Deployment-Abhängigkeit gibt.
- Schließlich gibt es klare Regeln, wie eigene Ergänzungen ihren Weg in die Open-Source-Bibliothek finden können.

Letztendlich ist der Unterschied zwischen einer gemeinsam genutzten Bibliothek und einem Open-Source-Projekt eine höhere Qualität in Bezug auf verschiedene Aspekte. Außerdem gibt es einen organisatorischen Aspekt: Es gibt ein Team, das sich um das Open-Source-Projekt kümmert. Es steuert das Projekt und entwickelt es weiter. Dieses Team macht nicht unbedingt alle Änderungen, aber koordiniert sie. Idealerweise hat das Team Mitglieder aus verschiedenen Organisationen und Projekten, sodass das Projekt aus verschiedenen Einsatzkontexten heraus weiterentwickelt wird.

Aus dem Vorbild der Open-Source-Projekte ergeben sich verschiedene Optionen für wiederverwendbaren Code in einem Microservices-Projekt:

- Die Organisation rund um wiederverwendbare Bibliotheken wird so aufgestellt wie ein Open-Source-Projekt. Es gibt verantwortliche Mitarbeiter, die den Code weiterentwickeln, Anforderungen konsolidieren und Änderungen von anderen Mitarbeitern einarbeiten. Die Teammitglieder kommen idealerweise aus verschiedenen Microservice-Teams.
- Aus dem wiederverwendbaren Code wird ein echtes Open-Source-Projekt.

Entwickler außerhalb der Organisation können das Projekt nutzen und erweitern.

Hinter beiden Entscheidungen kann ein signifikantes Investment stehen, weil wesentlich mehr in Qualität, Dokumentation usw. investiert werden muss. Außerdem müssen die Mitarbeiter, die an dem Projekt arbeiten, dazu auch genügend Freiräume in ihren Teams bekommen. Die Teams können die Priorisierung in dem Open-Source-Projekt steuern, indem sie Mitarbeiter nur für bestimmte Aufgaben freistellen. Wegen der Höhe des Investments und der möglichen Probleme bei der Priorisierung, sollte der Schritt zum Etablieren eines Open-Source-Projekts wohlüberlegt sein. Die Idee ist jedoch nicht neu – es gibt allerdings schon länger Erfahrungen in diesem Bereich. [4]

Wenn das Investment sehr hoch ist, dann ist der Code im Moment kaum wiederverwendbar und eine Nutzung des Codes im aktuellen Zustand führt zu erheblichen Reibungsverlusten. Der Code ist wahrscheinlich nicht nur schwer wiederverwendbar, sondern auch schwer verwendbar. Dann ist die Frage, warum sich die Teammitglieder mit der schlechten Qualität des Codes zufriedengeben. Eine Investition in die Qualität des Codes, um ihn wiederverwendbar zu machen, kann sich bereits durch die einfache Verwendbarkeit amortisieren.

Es erscheint zunächst nicht besonders sinnvoll, Code auch Entwicklern außerhalb des Projekts oder der eigenen Organisation zur Verfügung zu stellen. Dazu muss die Qualität von Code und Dokumentation so gut sein, dass externe Entwickler ohne direkten Kontakt zu den Entwicklern des Open-Source-Projekts den Code nutzen können. Davon scheinen nur die externen Entwickler einen Vorteil zu haben – sie bekommen schließlich den Code geschenkt. Aber ein echtes Open-Source-Projekt hat einige Vorteile:

- Externe nutzen den Code und finden so Schwachstellen. Außerdem nutzen sie den Code auch in anderen Projekten, sodass der Code weiter generalisiert wird. Das kommt der Qualität, aber auch der Dokumentation zugute.
- Eventuell beteiligen sich Externe an der Weiterentwicklung des Codes. Das ist nicht die Regel, sondern eher ein Glücksfall. Schon das Feedback der Externen durch Bug-Reports und Anforderungen neuer Features kann ein signifikanter Vorteil sein.
- Open-Source-Projekte sind eine gute Werbung für die technische Kompetenz. Das kann sowohl für Personalmarketing als auch für anderes Marketing genutzt werden.

Wichtig ist der Umfang des Projekts. Wenn es nur eine einfache Ergänzung eines vorhandenen Open-Source-Projekts ist, kann das Investment überschaubar sein. Ein komplett neues Open-Source-Framework ist ein ganz anderes Thema.

Sehr einfach wiederzuverwendende Elemente sind Blueprints, also Dokumentationen für bestimmte Ansätze. Das können Elemente aus der Makro-Architektur sein wie beispielsweise ein Dokument, das den richtigen Ansatz für das Logging beschreibt. Ebenso kann es Templates geben, die alle notwendigen Bestandteile eines Microservice enthalten. Neben einem Code-Skelett kann dazu auch ein Build-Skript oder eine Continuous-Delivery-Pipeline gehören. Solche Erweiterungen sind schnell geschrieben und helfen direkt weiter.

Selber ausprobieren und experimentieren

Vielleicht hast du schon eigene technische Bibliotheken in Projekten genutzt oder sogar selbst entwickelt. Versuche abzuschätzen, wie hoch der Aufwand wäre, diese Bibliotheken zu echten Open-Source-Bibliotheken zu machen. Neben einer guten Code-Qualität gehört dazu auch eine Dokumentation über die Nutzung und die Erweiterung des Codes. Außerdem muss es einen Bug Tracker und Foren geben. Wie einfach wäre dann die Wiederverwendung im Projekt selbst? Wie hoch wäre die Qualität der Bibliothek?

13.8 Microservices ohne Organisationsänderung?

Microservices sind mehr als nur ein Ansatz für Software-Architektur. Sie haben ausgeprägte Auswirkungen auf die Organisation. Änderungen an der Organisation sind oft sehr schwierig. Daher ist die Frage, ob Microservices ohne Änderungen an der Organisation umsetzbar sind.

Microservices ermöglichen unabhängige Teams. Die fachlichen Teams verantworten jeweils einen oder mehrere Microservices vollständig – idealerweise einschließlich der Entwicklung und dem Betrieb. In der Theorie wäre es möglich, Microservices ohne Aufteilung in fachliche Teams umzusetzen. Die Entwickler würden dann jeden Microservice ändern können – eine Erweiterung der Ideen aus [Abschnitt 13.2](#). Es wäre sogar denkbar, dass Teams nach technischen Schnitten auf fachlich geschnittenen Microservices arbeiten. Es würde ein UI-, Middle-Tier- und ein Datenbank-Team geben, das fachliche Microservices wie Bestellprozess oder Registrierung bearbeitet. Dann können allerdings einige Vorteile der Microservices nicht mehr realisiert werden. Zum einen ist ein Skalieren der agilen Prozesse durch Microservices nicht mehr möglich, zum anderen wird das Vorgehen nur mit einer Einschränkung der Technologiefreiheit umsetzbar sein. Sonst können die Teams die verschiedenen Microservices mit unterschiedlichen Technologien nicht mehr beherrschen. Außerdem kann jedes Team jeden Microservice ändern. Das birgt die Gefahr, dass zwar ein verteiltes System entsteht, aber die Abhängigkeiten so sind, dass die unabhängige Entwicklung eines einzelnen Microservice nicht mehr möglich ist. Der Zwang zu unabhängigen Microservices ist weg, weil ein Team mehrere Microservices zusammen ändern und so auch mit Microservices mit vielen Abhängigkeiten umgehen kann. Aber selbst unter diesen Bedingungen können eine nachhaltige Entwicklung, ein einfacherer Einstieg in Continuous Delivery, unabhängige Skalierung einzelner Microservices oder ein einfacher Umgang mit Legacy-Systemen noch umgesetzt werden, weil die Deployment-Einheiten kleiner sind.

Um es klar zu sagen: Ein Einführen von Microservices ohne fachliche Teams erzielt die wesentlichen Vorteile nicht und ist daher kein empfehlenswerter Ansatz. Es ist immer problematisch, wenn von einem bestimmten Verfahren nur einige Teile umgesetzt werden, denn die Synergien der verschiedenen Teile ergeben erst den Wert des ganzen. Eine Umsetzung von Microservices ohne fachlich ausgerichtete Teams ist zwar eine denkbare Option – aber es ist ganz klar keine Empfehlung.

Wie schon in [Abschnitt 13.6](#) diskutiert, sollte die Fachbereiche

Ausrichtung anhand der Microservices sich bis in die Fachbereiche fortsetzen. Das ist aber in der Realität manchmal nicht zu erreichen, weil die Architektur der Microservices sich zu sehr von der organisatorischen Aufstellung der Fachbereiche unterscheidet. Es ist unwahrscheinlich, dass die Organisation der Fachbereiche sich der Aufteilung in Microservices anpasst. Wenn die Aufteilung der Microservices nicht angepasst werden kann, müssen die Product Owner die Priorisierung regeln und Wünsche der Fachbereiche, die mehrere Microservices betreffen, so koordinieren, dass für die Teams alle Anforderungen eindeutig priorisiert sind. Wenn das nicht möglich ist, kann ein Ansatz nach Collective Code Ownership ([Abschnitt 13.2](#)) das Problem begrenzen. Dann kann der Product Owner mit seinem Team auch Microservices ändern, die eigentlich nicht zu seinem Einflussbereich gehören. Das kann die bessere Alternative gegenüber einer Koordinierung über Teams hinweg sein – aber beide Lösungen sind sicher nicht optimal.

In vielen Organisationen gibt es ein getrenntes Team, Betrieb das sich um den Betrieb kümmert. Die Teams, die für die Microservices zuständig sind, sollten sich auch um den Betrieb der Microservices kümmern, sodass DevOps umgesetzt wird. Wie schon in [Abschnitt 13.5](#) dargestellt, ist es aber nicht wegen Microservices notwendig, die Organisation auf DevOps umzustellen. Wenn eine Trennung in Betrieb und Entwicklung beibehalten werden soll, muss der Betrieb im Rahmen der Makro-Architektur die notwendigen Anforderungen an die Microservices festlegen, um einen Betrieb des Systems gewährleisten zu können.

Oft werden auch Architektur und Entwicklung Architektur getrennt. In einer Microservices-Umgebung gibt es für die Architekten den Bereich der Makro-Architektur, in dem sie allgemeine Festlegungen für die Teams treffen können. Eine Alternative ist, die Architekten auf die Teams aufzuteilen. Sie können in den Teams mitarbeiten. Zusätzlich können sie ein übergreifendes Gremium gründen, das Themen für die MakroArchitektur festlegt. Dann muss gewährleistet sein, dass die Architekten für diese Aufgaben tatsächlich Zeit haben. Es besteht die Gefahr, dass sie mit der Arbeit im Team vollständig ausgelastet werden.

Selber ausprobieren und experimentieren

- Wie sieht die Organisation eines dir bekannten Projekts aus?
- Gibt es eine eigene Organisationseinheit, die sich um die Architektur kümmert? Wie würde sie in eine Microservices-Architektur passen?
- Wie ist der Betrieb aufgestellt? Wie kann die Organisation des Betriebs Microservices am besten unterstützen?
- Wie gut passt die fachliche Aufteilung zu den Fachbereichen? Wie könnte es optimiert werden?
- Kann jedem Team ein Product Owner mit einem passenden Aufgabenfeld zugewiesen werden?

13.9 Fazit

Microservices ermöglichen die Unabhängigkeit der Teams bezüglich technischer Entscheidungen und des Deployments ([Abschnitt 13.1](#)). Dadurch können die Teams unabhängig Anforderungen umsetzen. Letztendlich können so viele kleine Teams gemeinsam ein großes Projekt bearbeiten. Das reduziert den Kommunikations-Overhead zwischen den Teams. Da die Teams unabhängig deployen können, reduziert es auch das Gesamtrisiko des Projekts.

Ideal ist eine Aufstellung der Teams, sodass die Teams getrennt an Fachlichkeiten arbeiten können. Wenn das nicht möglich oder zu viel Koordination zwischen den Teams notwendig ist, kann Collective Code Ownership eine Alternative sein ([Abschnitt 13.2](#)). Dabei kann jeder Entwickler jeden Code ändern. Dennoch ist ein Team für jeden Microservice zuständig. Mit diesem Team müssen Änderungen an einem Microservice abgesprochen werden.

[Abschnitt 13.3](#) hat dargestellt, dass es bei Microservices eine Makro-Architektur gibt. Sie beinhaltet die Entscheidungen, die alle Microservices betreffen. Davon getrennt ist die Mikro-Architektur, die für jeden Microservice anders sein kann. In den Bereichen Technologie, Betrieb, fachliche Architektur und Tests gibt es Entscheidungen, die entweder auf der Ebene der Mikro- oder Makro-Architektur liegen. Bei vielen Bereichen ist offen, ob die Entscheidungen in die Mikro- oder Makro-Architektur fallen. Jedes Projekt kann sie in die Teams delegieren (Mikro-Architektur) oder zentral vorgeben (Makro-Architektur). Die Delegation in die Teams entspricht dem Ziel einer möglichst großen Unabhängigkeit – und ist daher oft die bessere. Ein separates Architektur-Team kann die Makro-Architektur definieren – oder das Team wird aus Mitgliedern der Teams zusammengestellt, die für die Microservices zuständig sind.

Mit der Verantwortung für die Makro-Architektur geht ein Konzept für die technische Führung einher ([Abschnitt 13.4](#)): Wenig Makro-Architektur bedeutet mehr Verantwortung für die Teams und weniger Verantwortung eines zentralen Teams.

Microservices profitieren zwar von einer Zusammenlegung von Betrieb und Entwicklung zu DevOps ([Abschnitt 13.5](#)). Zwingend ist DevOps für Microservices jedoch nicht. Wenn DevOps nicht möglich oder gewünscht ist, kann der Betrieb im Rahmen der Makro-Architektur Regeln festlegen, die den Betrieb der Microservices erlauben und vereinheitlichen.

Microservices sollten jeweils eigene getrennte Anforderungen umsetzen. Daher ist es gut, wenn jeder Microservice einem Fachbereich zugeordnet werden kann ([Abschnitt 13.6](#)). Ist das nicht möglich, müssen die Product Owner die Anforderungen der Fachbereiche so koordinieren, dass es für jeden Microservice eindeutig priorisierte Anforderungen gibt. Wenn Collective Code Ownership genutzt wird, kann ein Product Owner mit seinem Team auch Microservices anderer Teams ändern – was die Koordination erleichtern kann. Statt die Prioritäten zu koordinieren, nimmt ein Team die für ein Feature notwendigen Änderungen vor – selbst in anderen Microservices. Das für den Microservice zuständige Team kann die Änderungen reviewen und gegebenenfalls anpassen.

Code kann in einem Microservices-Projekt wiederverwendet werden, wenn der Code wie ein Open-Source-Projekt behandelt wird ([Abschnitt 13.7](#)). Es kann als internes Projekt wie ein Open-Source-Projekt geführt werden – oder tatsächlich ein öffentliches Open-Source-Projekt werden. Dabei ist zu beachten, dass der Aufwand für ein Open-Source-Projekt hoch ist. Es kann effizienter sein, Code nicht wiederzuverwenden. Außerdem müssen die Entwickler des Open-Source-Projekts fachliche Anforderungen gegen Änderungen an dem Open-Source-Projekt priorisieren. Das kann im Einzelfall eine schwierige Entscheidung sein.

[Abschnitt 13.8](#) hat erläutert, dass eine Einführung von Microservices ohne Organisationsänderung auf Entwicklungsseite nur theoretisch denkbar ist. Wenn es keine fachlich ausgerichteten Teams gibt, die jeweils eine bestimmte Fachlichkeit unabhängig von den anderen Teams weiterentwickeln, ist es praktisch unmöglich, mehrere Features parallel zu entwickeln und so mehr Features in derselben Zeit auf den Markt zu bringen. Das ist aber ein wesentlicher Vorteil von Microservices. Nachhaltige Entwicklung, ein einfacherer Einstieg in Continuous Delivery, unabhängige Skalierung einzelner Microservices oder ein einfacher Umgang mit Legacy-Systemen sind weiterhin möglich. Betrieb und ein Architektur-Team können die Makro-Architektur definieren, sodass in diesem Bereich keine Änderungen an der Organisation zwingend sind. Idealerweise entsprechen die Anforderungen der Fachbereiche jeweils einem Microservice. Wenn das nicht möglich ist, müssen die Product Owner die Änderungen entsprechend koordinieren und priorisieren.

Wesentliche Punkte

- Microservices haben signifikante Auswirkungen auf die Organisation. Unabhängige kleine Teams, die gemeinsam ein großes Projekt bearbeiten, sind ein wichtiger Vorteil von Microservices.
- Die Betrachtung der Organisation als Teil der Architektur ist eine wesentliche Neuerung von Microservices.
- Eine Kombination von DevOps und Microservices ist vorteilhaft, aber nicht zwingend.

13.10 Links & Literatur

- [1] <http://www.infoq.com/news/2012/02/programmer-anarchy>
- [2] <http://en.wikipedia.org/wiki/NUMMI#Background>
- [3] <https://www.youtube.com/watch?v=uk-CF7klLdA>
- [4] <http://dirkriehle.com/2015/05/20/inner-source-in-platform-based-product-engineering/>
- [5] <http://www.scrumguides.org/scrum-guide.html#team>

Technologien

Dieser Teil des Buchs zeigt, wie mit konkreten Technologien Microservices umgesetzt werden können. Das [Kapitel 14](#) enthält ein vollständiges Beispiel für eine Microservices-Architektur auf der Basis von Java, Spring, Spring Boot, Spring Cloud, dem Netflix-Stack und Docker. Das Beispiel ist eine gute Basis für eigene Implementierungen oder Experimente. Viele der technologischen Herausforderungen aus dem [Teil III](#) werden in diesem Teil mit konkreten Technologien gelöst – beispielsweise Build, Deployment, Services Discovery, Kommunikation, Load Balancing oder Tests.

Noch kleiner als Microservices sind die Nanoservices aus [Kapitel 15](#). Sie erzwingen spezielle Technologien und einige Kompromisse. Daher zeigt das Kapitel Technologien, die sehr kleine Services realisieren können – konkret Amazon Lambda für JavaScript und Java, OSGi aus dem Java-Bereich, Java EE, Vert.x auf der JVM (Java Virtual Machine) mit Unterstützung für Sprachen wie Java, Scala, Clojure, Groovy, Ceylon, JavaScript, Ruby oder Python. Die Programmiersprache Erlang ermöglicht ebenfalls sehr kleine Services, kann aber auch andere Systeme integrieren. Seneca ist ein JavaScript-Framework insbesondere für die Umsetzung von Nanoservices.

Zum Abschluss des Buches zeigt [Kapitel 16](#), was mit Microservices erreicht werden kann.

14 Ein Beispiel für eine Microservices-Architektur

Dieses Kapitel zeigt ein Beispiel für eine mögliche Implementierung einer Microservices-Architektur. Das Ziel ist, konkrete Technologien zu zeigen und so eine Basis für eigene Experimente zu schaffen. Das Beispiel hat eine sehr einfache fachliche Architektur mit einigen Kompromissen. [Abschnitt 14.1](#) geht darauf detailliert ein.

Für ein reales System mit der Komplexität dieses Beispiels sollte ein Ansatz ohne Microservices gewählt werden. Durch die geringe Größe ist das Beispiel aber leicht zu verstehen und kann einfach erweitert werden. Einige Aspekte einer Microservice-Umgebung – wie Security, Dokumentation, Monitoring oder Logging – setzt das Beispiel nicht um – aber diese Aspekte können durch eigene Experimente recht einfach erweitert werden.

Der [Abschnitt 14.2](#) erläutert den Technologie-Stack des Beispiels. Die Werkzeuge für den Build beschreibt [Abschnitt 14.3](#). Der [Abschnitt 14.4](#) beschreibt Docker als Technologie für das Deployment. Docker muss auf einer Umgebung laufen. [Abschnitt 14.5](#) beschreibt Vagrant als Werkzeug, um solche Umgebungen zu erstellen. [Abschnitt 14.6](#) zeigt als alternatives Tool Docker Machine für das Erzeugen einer Docker-Umgebung, das mit Docker Compose zur Koordination mehrerer Docker-Container ([Abschnitt 14.7](#)) kombiniert werden kann. Die Umsetzung der Service Discovery diskutiert [Abschnitt 14.8](#). Die Kommunikation zwischen den Microservices und der Schnittstelle hin zum Benutzer steht im Mittelpunkt von [Abschnitt 14.9](#). Dank Resilience sind andere Microservices nicht beeinträchtigt, wenn ein Microservice ausfällt. Im Beispiel ist Resilience mit Hystrix umgesetzt ([Abschnitt 14.10](#)). Damit eng verwandt ist das Load Balancing ([Abschnitt 14.11](#)), das die Last auf mehrere Instanzen eines Microservice verteilen kann. Möglichkeiten zur Integration von Nicht-Java-Technologien beschreibt [Abschnitt 14.12](#) und das Testen zeigt [Abschnitt 14.13](#).

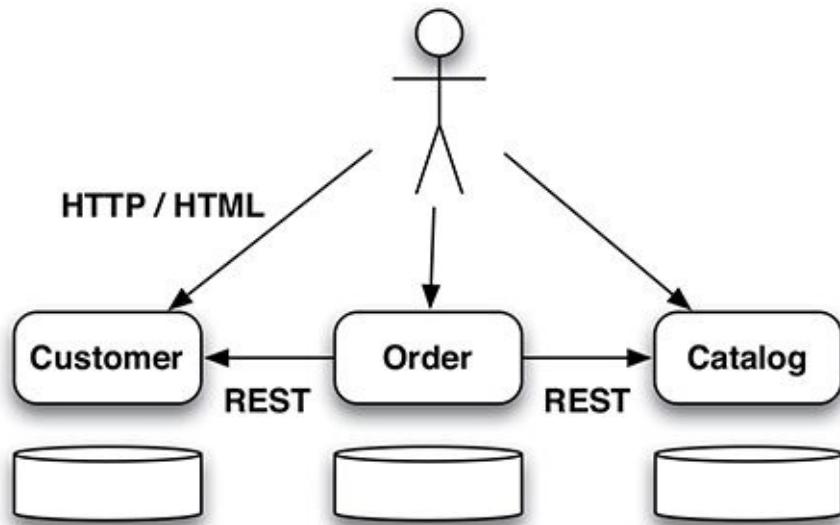
Der Code für das Beispiel findet sich unter [1]. Er steht unter der Apache-Lizenz und kann daher frei für eigene Zwecke genutzt und erweitert werden.

14.1 Fachliche Architektur

Das Beispiel setzt eine einfache Weboberfläche um, mit der Nutzer Bestellungen aufgeben können. Es gibt drei Microservices (siehe [Abb. 14-1](#)):

- Catalog führt Buch über die Waren. Es können Waren erzeugt oder gelöscht werden.
- Customer tut dasselbe für Kunden: Es ist möglich, neue Kunden anzulegen oder vorhandene zu löschen.
- Order kann nicht nur Bestellungen anzeigen, sondern auch neue Bestellungen erfassen.

Abb. 14-1 Architektur des Beispiels



Für die Bestellungen muss der Order-Microservice Zugriff auf die beiden anderen Microservices – Customer und Catalog – haben. Die Kommunikation findet über REST statt. Diese Schnittstelle ist aber nur zur internen Kommunikation zwischen den Microservices gedacht. Der Kunde kann mit allen drei Microservices über die HTML-/HTTP-Schnittstelle interagieren.

Die Datenhaushalte der drei Microservices sind Getrennte Datenhaushalte vollständig getrennt. Nur der jeweilige Microservice kennt alle Informationen über die Geschäftsobjekte. Der Order-Microservice speichert nur die Primärschlüssel der Waren und Kunden, die für den Zugriff über die REST-Schnittstelle der Microservices notwendig sind. Ein reales System sollte lieber künstliche Schlüssel nutzen, weil sonst die internen Primärschlüssel aus der Datenbank nach außen sichtbar werden. Das sind interne Details der Datenhaltung. Um die Primärschlüssel zu exponieren, konfiguriert die Klasse `SpringRestDataConfig` in den Microservices Spring Data Rest entsprechend.

Wenn eine Bestellung angezeigt werden soll, so wird Viel Kommunikation einmal der Customer-Microservice für die Kundendaten aufgerufen und für jede Zeile der Bestellung der Catalog-Service, um den Preis der Ware zu ermitteln. Das kann die Antwortzeiten der Anwendung negativ beeinflussen, denn die Anzeige der Bestellung ist erst möglich, wenn alle diese Anfragen von den anderen Microservices beantwortet worden sind. Da die Anfragen an die anderen Services synchron und sequenziell geschehen, addieren sich die Latenzen auf. Asynchrone parallele Anfragen können dieses Problem lösen.

Hinzu kommt, dass viel Rechenkapazität in das Konvertieren der Daten zum Verschicken und Empfangen gesteckt wird. Für ein kleines Beispiel ist das akzeptabel. Wenn eine solche Anwendung in Produktion laufen soll, muss man über Alternativen nachdenken.

Dieses Problem kann beispielsweise durch Caching gelöst werden. Das ist sogar recht einfach möglich, weil Kundendaten sich nicht sehr häufig ändern. Die Waren können sich häufiger ändern – aber bei Weitem nicht so schnell, dass es mit einem Caching zu Problemen kommt. Höchstens die Datenmenge kann dem Ansatz entgegenstehen. Die Nutzung von Microservices hat den Vorteil, dass ein solcher Cache recht einfach an der Schnittstelle der Microservices umgesetzt werden kann – oder sogar auf Ebene von HTTP,

wenn dieses Protokoll verwendet wird. REST Services können also transparent und ohne zusätzlichen Programmieraufwand mit einem HTTP Cache versehen werden, wie er auch für Websites üblich ist.

Das Caching würde das Problem der Antwortzeiten technisch lösen. Es kann aber auch ein Hinweis auf ein fundamentales Problem sein. [Abschnitt 4.3](#) hat argumentiert, dass ein Microservice einen BOUNDED CONTEXT umfassen soll. Ein bestimmtes Domänenmodell hat nur in einem BOUNDED CONTEXT Gültigkeit. Die Aufteilung auf Microservices in diesem Beispiel verletzt diese Idee: Das Domänenmodell ist aufgeteilt auf die Microservices Order für die Bestellungen, Catalog für die Waren und Customer für die Kunden. Eigentlich sollten die Daten dieser Entitäten in verschiedenen BOUNDED CONTEXT aufgeteilt werden. Die implementierte Aufteilung setzt trotz geringer fachlicher Komplexität ein System aus drei Microservices um. So ist das Beispiel einfach zu verstehen, hat aber mehrere Microservices und zeigt so auch die Kommunikation zwischen Microservices. In einem realen System kann der Order-Microservice die für den Bestellvorgang relevanten Informationen der Waren wie beispielsweise den Preis selber verwalten. Gegebenenfalls kann der Service die Daten auch von einem anderen Microservice in die eigene Datenbank replizieren, um so performant auf die Daten zuzugreifen. Das ist eine weitere Möglichkeit neben dem bereits erwähnten Caching. Es gibt verschiedene Wege, wie die Domänenmodelle in den verschiedenen BOUNDED CONTEXTS Order und Customer bzw. Catalog aufgeteilt werden können.

Übrigens hat diese Modellierung auch fachliche Fehler zur Folge: Wenn in dem System eine Bestellung aufgenommen und anschließend der Preis der Ware geändert wird, ändert sich auch der Betrag der Bestellung – das sollte sicher nicht so sein. Wenn die Ware gelöscht wird, kommt es beim Anzeigen der Bestellung sogar zu einem Fehler. Eigentlich müssten die Informationen über die Ware oder den Kunden in die Bestellung übernommen werden. Dann würden die historischen Daten für die Bestellungen einschließlich Kundendaten und Daten über die Waren in die Hoheit des Order-Service übergehen.

Es ist wichtig, das Problem mit dieser Modellierung zu verstehen, denn oft wird die Aufgabe einer globalen Architektur missverstanden: Das Team entwirft ein Domänenmodell, das beispielsweise Objekte wie Kunden, Bestellungen und Waren enthält. Anhand dieses Modells werden Microservices definiert. Auf diese Art und Weise könnte die Aufteilung der Microservices im Beispiel entstanden sein – und so ergibt sich auch die große Menge an Kommunikation. Besser könnte beispielsweise eine Aufteilung nach Prozessen wie dem Bestellprozess, der Registrierung eines Benutzers oder der Produktsuche sein. Jeder der Prozesse könnte ein BOUNDED CONTEXT sein und ein eigenes Domänenmodell für die wichtigsten fachlichen Objekte haben. So können für die Produktsuche die Kategorien der Produkte wichtig sein – aber für den Bestellprozess vielleicht nur Daten wie das Gewicht oder die Größe.

Allerdings kann diese Aufteilung auch in einem realen System sinnvoll sein. Wenn der Order-Microservice zusammen mit der Verwaltung der Daten der Kunden und Waren zu groß wird, ist eine Abspaltung der Verwaltung der Daten sinnvoll. Außerdem können die Daten auch von anderen Microservices genutzt werden. Bei der Architektur eines Systems

Bounded Context

Keine Aufteilung der Microservices nach Daten!

gibt es eben selten absolute Wahrheiten.

14.2 Basistechnologien

Die Microservices sind mit Java umgesetzt. Grundlegende Funktionalitäten für das Beispiel liefert das Spring-Framework [2]. Es bietet nicht nur Dependency Injection, sondern auch ein Web-Framework, das die Umsetzung REST-basierter Services ermöglicht.

Die Datenbank HSQLDB verwaltet und speichert die Daten. Es ist eine In-Memory-Datenbank, die in Java geschrieben ist. Sie legt die Daten nur im RAM ab, sodass beim erneuten Start der Anwendung die Daten verloren sind. Für einen produktiven Einsatz ist diese Datenbank nicht sinnvoll, auch wenn sie die Daten auf eine Festplatte schreiben kann. Dafür muss kein zusätzlicher Datenbank-Server installiert werden, was die Beispielanwendung sehr einfach hält. Die Datenbank läuft in der jeweiligen Java-Anwendung.

Die Microservices nutzen Spring Data REST [3], um mit wenig Aufwand die Domänenobjekte per REST zur Verfügung zu stellen und in die Datenbank zu schreiben. Ein direktes Herausreichen der Objekte bedeutet, dass die interne Datenrepräsentation in die Schnittstelle der Services wandert. Eine Änderung der Datenstrukturen ist nur schwer möglich, weil auch die Clients angepasst werden müssen. Allerdings kann Spring Data REST bestimmte Elemente der Daten ausblenden und ist flexibel konfigurierbar, sodass die enge Bindung des internen Modells und der Schnittstelle aufgelöst werden kann, wenn das notwendig ist.

Spring Boot [4] erleichtert den Umgang mit Spring weiter. Spring Boot macht den Build eines Spring-Systems sehr einfach: Durch die Spring Boot Starter stehen vordefinierte Pakete zur Verfügung, die alles enthalten, was für eine bestimmte Art von Anwendung notwendig ist. Spring Boot kann WAR-Dateien erzeugen, die auf einem Java-Application- oder Webserver installiert werden können. Es ist aber auch möglich, die Anwendungen ohne Application- oder Webserver zu betreiben. Das Ergebnis des Builds ist dann eine JAR-Datei, die mit einer Java-Laufzeitumgebung (JRE – Java Runtime Environment) ausgeführt werden kann. In der JAR-Datei ist alles enthalten, um die Anwendung auszuführen – auch Code, um mit HTTP-Anfragen umzugehen. Dieser Absatz ist wesentlich weniger aufwendig und einfacher als die Nutzung eines Application Servers [5].

Ein einfaches Beispiel für eine Spring-Boot-Anwendung zeigt Listing 14-1. Das Hauptprogramm `main` gibt die Kontrolle an Spring Boot ab und übergibt dabei die eigene Klasse als Parameter, sodass die Anwendung aufgerufen werden kann. Die Klasse ist mit `@SpringBootApplication` markiert, sodass Spring Boot eine passende Umgebung erstellen kann. Beispielsweise wird ein Webserver aktiviert und eine Umgebung für Spring-Webanwendung aufgebaut, da die Anwendung eine Webanwendung ist. Wegen `@RestController` instanziert das Spring Framework die Klasse und ruft Methoden zur Bearbeitung von REST-Aufrufen auf. `@RequestMapping` zeigt, welche Methode auf welche

Anfrage reagieren soll. Konkret wird bei Aufrufen der URL “/” die Methode hallo() aufgerufen, die als Ergebnis im HTTP-Body die Zeichenkette “hallo” zurückgibt. In einem @RequestMapping können auch URL-Templates wie “/customer/{id}” genutzt werden, sodass eine URL wie “/customer/42” auseinander geschnitten und die 42 an einen mit @PathVariable annotierten Parameter gebunden wird. Als Abhängigkeit nutzt die Anwendung nur spring-boot-starter-web. Sie zieht dann alle nötigen Bibliotheken für die Anwendung nach – beispielsweise den Webserver, das Spring Framework und weitere abhängige Klassen. [Abschnitt 14.3](#) wird das noch detaillierter darstellen.

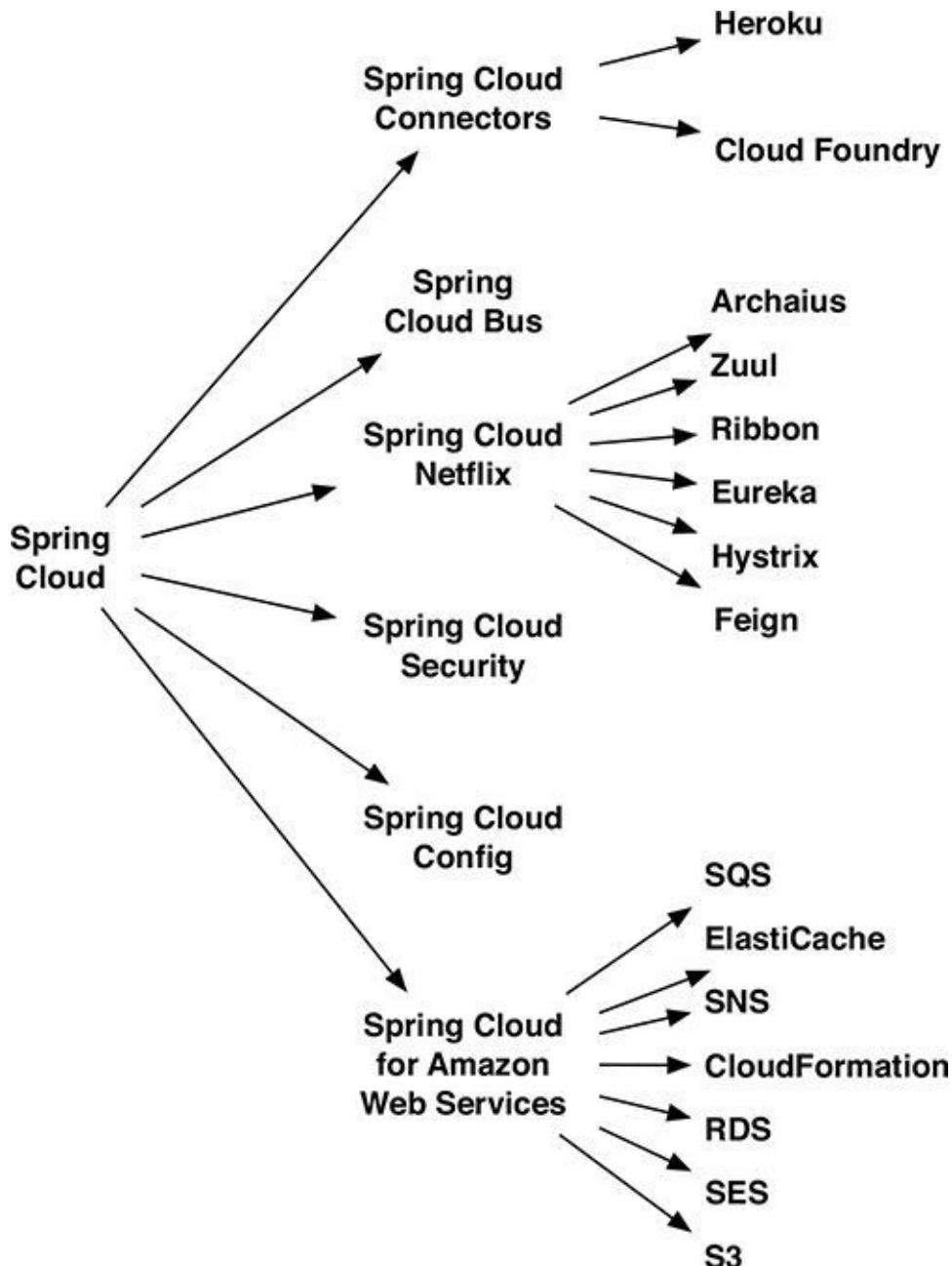
Listing 14–1 Ein einfacher Spring-Boot-REST-Service

```
@RestController  
@SpringBootApplication  
public class ControllerAndMain {  
  
    @RequestMapping("/")  
    public String hallo() {  
        return "hallo";  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(ControllerAndMain.class, args);  
    }  
}
```

Schließlich nutzt die Beispielanwendung Spring Cloud [6] für einfachen Zugriff auf den Netflix-Stack. [Abbildung 14–2](#) zeigt einen Überblick.

Spring Cloud

Abb. 14–2 Überblick über Spring Cloud



Spring Cloud bietet mit den Spring Cloud Connectors Zugriff auf die PaaS (Platform as a Service) Heroku und Cloud Foundry. Spring Cloud for Amazon Web Services bietet eine Schnittstelle für Dienste aus der Amazon Cloud. Dieser Teil von Spring Cloud ist für den Namen des Projekts verantwortlich, aber für die Umsetzung von Microservices nicht hilfreich.

Die anderen Subprojekte von Spring Cloud stellen eine sehr gute Basis für die Umsetzung von Microservices dar:

- *Spring Cloud Security* unterstützt die Umsetzung von Sicherheitsmechanismen, wie sie für Microservices typischerweise notwendig sind. Dazu gehört das Single Sign On in eine Microservices-Landschaft, sodass ein Nutzer jeden der Microservices nutzen kann, ohne sich jedes Mal neu einzuloggen. Außerdem wird das Token des Benutzers bei Aufrufen von anderen REST-Services automatisch weitergereicht, damit auch diese Aufrufe mit den richtigen Rechten arbeiten können.
- *Spring Cloud Config* kann dazu genutzt werden, die Konfiguration von Microservices zu zentralisieren und gegebenenfalls die Konfiguration der Microservices dynamisch

anzupassen. [Abschnitt 12.4](#) hat schon Technologien gezeigt, die Microservices beim Deployment konfigurieren. Um den Zustand eines Servers jederzeit reproduzieren zu können, ist es sinnvoll, bei einer Änderung der Konfiguration einen neuen Server mit einer Instanz des Microservice zu starten, statt einen vorhanden Server dynamisch anzupassen. Werden die Server dynamisch geändert, ist nicht garantiert, dass neue Server mit der richtigen Konfiguration erzeugt werden. Sie werden ja über einen anderen Weg konfiguriert. Wegen dieser Nachteile verzichtet das Beispiel auf diese Technologie.

- *Spring Cloud Bus* kann vor allem für Spring Cloud Config dynamisch Änderungen an der Konfiguration verbreiten. Außerdem können die Microservices darüber kommunizieren. Für das Beispiel wird diese Technologie nicht genutzt, da Spring Cloud Config nicht verwendet wird und die Microservices über REST kommunizieren.

Spring Cloud Netflix bietet einen einfachen Zugriff auf den Netflix-Stack, der speziell für die Implementierung von Microservices entwickelt worden ist. Folgende Technologien sind in diesem Stack enthalten:

- *Archaius* dient zur Konfiguration.
- *Zuul* kann das Routing von Requests an verschiedene Dienste umsetzen.
- *Ribbon* dient als Load Balancer.
- *Hystrix* hilft dabei, Resilience in den Microservices zu implementieren.
- *Turbine* kann die Monitoring-Daten aus mehreren Hystrix-Servern konsolidieren.
- *Feign* ist eine Möglichkeit, um REST-Clients einfacher zu implementieren. Es ist nicht auf Microservices begrenzt. Im Beispiel wird es nicht verwendet.
- *Eureka* dient zur Service Discovery.

Vor allem diese Technologien beeinflussen die Umsetzung des Beispiels.

Selber ausprobieren und experimentieren

Für die Einführung in Spring lohnt sich ein Blick auf die Spring Guides unter <https://spring.io/guides/>. Sie zeigen ganz praktisch, wie mithilfe von Spring beispielsweise REST-Service umgesetzt oder Messaging-Lösungen mit JMS implementiert werden können. Einen Einstieg in Spring Boot bietet <https://spring.io/guides/gs/spring-boot/>. Ein Durcharbeiten dieses Guides vermittelt die notwendigen Grundlagen, um die weiteren Beispiele in diesem Kapitel zu verstehen.

14.3 Build

Das Beispielprojekt wird mit dem Werkzeug Maven [16] gebaut. Unter [19] ist auch die Installation des Werkzeugs beschrieben. Mit `mvn package` im Verzeichnis `microservice/microservice-demo` werden alle abhängigen Bibliotheken aus dem Internet heruntergeladen und die

Anwendung kompiliert.

Die Konfiguration der Projekte für Maven ist in Dateien namens pom.xml abgelegt. Das Beispielprojekt hat ein Parent-POM im Verzeichnis microservice-demo. Es enthält die allgemeingültigen Einstellungen für alle Module und außerdem eine Liste der Module des Beispielprojekts. Jeder Microservice ist ein solches Modul und auch einige der Infrastruktur-Server. Die einzelnen Module haben eigene pom.xml, die unter anderem den Namen des Moduls enthalten. Außerdem enthalten sie die Abhängigkeiten – also Java-Bibliotheken, die sie nutzen.

Listing 14–2 Ausschnitt aus pom.xml mit Abhängigkeiten

```
...
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
...
</dependencies>
...
```

[Listing 14–2](#) zeigt einen Ausschnitt aus einem pom.xml, das die Abhängigkeiten des Moduls auflistet. Abhängig davon, welche Features von Spring Cloud das Projekt nutzt, müssen in diesem Bereich des pom.xml noch zusätzliche Einträge vorgenommen werden – üblicherweise mit der groupId org.springframework.cloud.

Das Ergebnis des Build-Prozesses ist pro Modul jeweils eine JAR-Datei, die den kompilierten Code, die Konfiguration und alle benötigten Bibliotheken enthält. Java kann solche JAR-Dateien direkt starten. Obwohl die Microservices über HTTP zugreifbar sind, müssen sie nicht auf einem Application- oder Webserver deployt werden. Auch dieser Teil der Infrastruktur ist in der JAR-Datei enthalten.

Weil die Projekte mit Maven gebaut werden, können sie auch in alle gängigen Java-Entwicklungsumgebungen importiert und dort weiterentwickelt werden. Entwicklungsumgebungen vereinfachen die Änderungen am Code deutlich.

Selber ausprobieren und experimentieren

Lade das Beispiel von [\[1\]](#) herunter. Installiere Maven [\[19\]](#). Führe im Unterverzeichnis microservices-demo den Befehl mvn package aus. So wird das komplette Projekt gebaut.

Unter [\[17\]](#) steht ein Beispielprojekt für das Continuous-Delivery-Buch [\[18\]](#) zur Verfügung. Es enthält im Unterverzeichnis ci-setup ein Setup für einen Continuous Integration Server (Jenkins) mit statischer Codeanalyse (Sonarqube) und Artifactory für die Verwaltung der Binärartefakte. Integriere das Microservices-Projekt in diese Infrastruktur, sodass bei jeder Änderung ein neuer Build ausgelöst wird. [Abschnitt 14.5](#) erläutert Vagrant noch näher. Dieses Werkzeug wird für den Aufbau dieses Continuous

Integration Servers verwendet. Es vereinfacht den Aufbau von Testumgebungen deutlich.

14.4 Deployment mit Docker

Das Deployment der Microservices ist sehr einfach:

- Es muss auf dem Server Java installiert werden.
- Auf dem Server muss die JAR-Datei vorhanden sein, die als Ergebnis des Builds geliefert worden ist.
- Für weitere Konfiguration kann eine eigene Konfigurationsdatei `application.properties` angelegt werden. Sie wird von Spring Boot automatisch ausgelesen und kann zur weiteren Anpassung genutzt werden. In der JAR-Datei ist ein `application.properties` mit Vorgabewerten enthalten.
- Schließlich muss ein Java-Prozess die Anwendung aus der JAR-Datei starten.

Jeder der Microservices startet in einem eigenen Docker-Container. Wie in [Abschnitt 12.6](#) erläutert, setzt Docker auf Linux-Container. So kann der Microservice Prozesse in anderen Docker-Containern nicht sehen und hat ein komplett eigenes Dateisystem. Basis für dieses Dateisystem ist das Docker Image. Aber alle Docker-Container teilen sich einen Kernel, was Ressourcen spart. Gegenüber einem Betriebssystemprozess hat ein Docker-Container praktisch keinen zusätzlichen Overhead.

***Listing 14–3** Dockerfile für einen Microservice aus dem Beispiel*

```
FROM java
CMD /usr/bin/java -Xmx400m -Xms400m \
    -jar /microservice-demo/microservice-demo-
catalog/target/microservice-demo-catalog-0.0.1-SNAPSHOT.jar
EXPOSE 8080
```

Eine Datei namens `Dockerfile` definiert, wie ein Docker-Container zusammengestellt wird. [Listing 14–3](#) zeigt ein Dockerfile für einen der Microservices aus dem Beispiel:

- `FROM` legt fest, welches Basisimage der Docker-Container nutzt. Ein Dockerfile für das Image `java` ist ebenfalls im Beispielprojekt enthalten. Es erzeugt ein minimales Docker Image, in dem nur eine JVM installiert ist.
- `CMD` definiert das Kommando, das beim Start des Docker-Containers ausgeführt wird. In diesem Beispiel ist es eine einfache Kommandozeile. Sie startet eine Java-Anwendung aus der JAR-Datei, die der Build erzeugt hat.
- Docker-Container können über Netzwerk-Ports nach außen kommunizieren. `EXPOSE` legt fest, welche Ports von außen zugreifbar sind – im Beispiel der Port 8080, auf dem die Anwendung HTTP-Anfragen entgegennimmt.

Die JAR-Dateien mit dem Anwendungscode sind übrigens in dem Beispiel nicht im Docker Image enthalten. Das Verzeichnis `/microservice-demo` ist ein Verzeichnis, das nicht zu dem Docker-Container gehört. Es liegt auf dem Host, auf dem die Docker-Container laufen. Es ist auch möglich, die Dateien in das Docker Image zu kopieren. Anschließend

kann das entstandene Image auf einen Repository Server kopiert und von dort heruntergeladen werden. Dann enthält der Docker-Container alle notwendigen Dateien, um den Microservice auszuführen. Ein Deployment in Produktion muss nur noch die Docker Images auf einem Productionsserver starten.

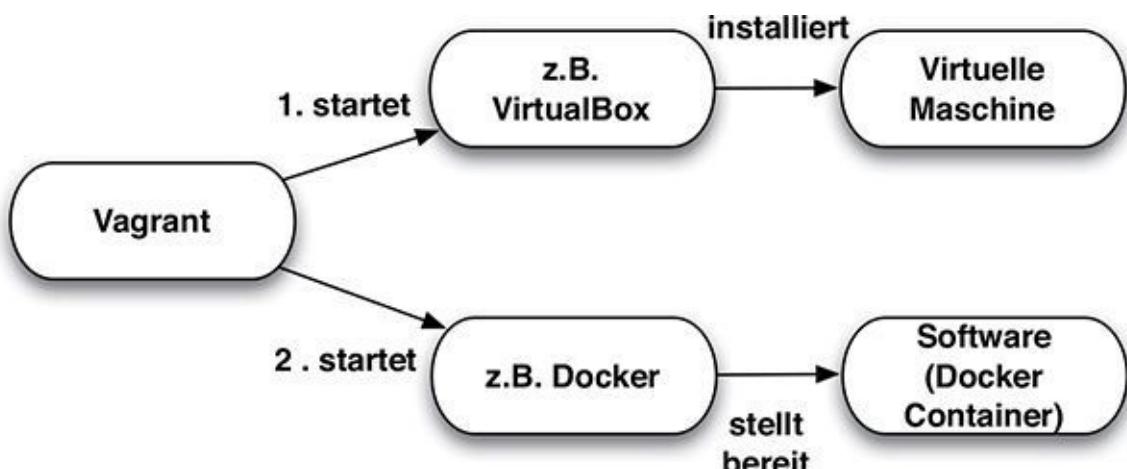
14.5 Vagrant

Docker läuft nur unter Linux, da es Linux-Container verwendet. Für andere Betriebssysteme gibt es allerdings Lösungen, die eine virtuelle Linux-Maschine starten und so die Nutzung von Docker erlauben. Das ist weitgehend transparent, sodass die Nutzung praktisch mit der Nutzung unter Linux identisch ist. Aber außerdem müssen alle Docker-Container gebaut und gestartet werden.

Um die Installation und den Umgang mit Docker möglichst einfach zu gestalten, nutzt die Beispielanwendung Vagrant. [Abbildung 14–3](#) zeigt, wie Vagrant funktioniert:

- Zunächst startet Vagrant eine virtuelle Maschine mit einer Software wie beispielsweise VirtualBox.
- Anschließend installiert Vagrant auf dieser virtuellen Maschine Software. Im Beispiel sind das Docker und die passenden Docker-Container. Vagrant erstellt die Docker-Container mithilfe der Dockerfiles und startet sie.

Abb. 14–3 Funktionsweise von Vagrant



Für die Konfiguration von Vagrant ist nur eine einzige Datei verantwortlich, nämlich das Vagrantfile. [Listing 14–4](#) zeigt das Vagrantfile der Beispielanwendung:

- config.vm.box wählt ein Basisimage aus – konkret eine Ubuntu-14.04-Linux-Installation (Trusty Tahr).
- Das Verzeichnis mit den Ergebnissen des Maven-Build blendet config.vm.synced_folder in die virtuelle Maschine ein. So können die Docker-Container die Ergebnisse des Builds direkt verwenden.
- Die Ports der virtuellen Maschine können an Ports des Rechners gebunden werden, auf dem die virtuelle Maschine läuft. Dazu dienen die config.vm.network-Einstellungen. So werden Anwendungen in der Vagrant VM so zugreifbar, als würden sie auf dem Rechner direkt laufen.

- config.vm.provision leitet den Teil der Konfiguration ein, der die Installation der Software in der virtuellen Maschine regelt. Als Werkzeug dient Docker, das ebenfalls automatisch in der virtuellen Maschine installiert wird.
- Schließlich erstellt d.build_image die Docker Images mithilfe der Dockerfiles. Das ist zunächst das Basisimage java. Dazu kommen Images für die drei Microservices: customer-app, catalog-app und order-app. Zu der Infrastruktur gehören die Images für die Server der Netflix-Technologien: eureka für Service Discovery, turbine für das Monitoring und zuul zum Routen der Benutzeranfragen.
- Mit d.run startet Vagrant die einzelnen Images. Dieser Schritt wird nicht nur bei der Provisionierung der virtuellen Maschine, sondern auch bei einem Neustart ausgeführt (run: "always"). Die Option -v blendet das Verzeichnis /microservice-demo in die Docker-Container ein, sodass die Docker-Container direkt den kompilierten Code ausführen können. -p bindet einen Port des Docker-Containers an einen Port der virtuellen Maschine. Die Links sorgen dafür, dass der Docker-Container eureka unter dem Hostnamen eureka in den anderen Docker-Containern zugreifbar ist.

Listing 14–4 Vagrantfile aus der Beispielanwendung

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.synced_folder "../microservice-demo",
    "/microservice-demo", create: true
  config.vm.network "forwarded_port", guest: 8080, host: 18080
  config.vm.network "forwarded_port", guest: 8761, host: 18761
  config.vm.network "forwarded_port", guest: 8989, host: 18989

  config.vm.provision "docker" do |d|
    d.build_image "--tag=java /vagrant/java"
    d.build_image "--tag=eureka /vagrant/eureka"
    d.build_image "--tag=customer-app /vagrant/customer-app"
    d.build_image "--tag=catalog-app /vagrant/catalog-app"
    d.build_image "--tag=order-app /vagrant/order-app"
    d.build_image "--tag=turbine /vagrant/turbine"
    d.build_image "--tag=zuul /vagrant/zuul"
  end

  config.vm.provision "docker", run: "always" do |d|
    d.run "eureka",
      args: "-p 8761:8761 -v /microservice-demo:/microservice-demo"
    d.run "customer-app",
      args: "-v /microservice-demo:/microservice-demo --link eureka:eureka"
    d.run "catalog-app",
      args: "-v /microservice-demo:/microservice-demo --link eureka:eureka"
    d.run "order-app",
      args: "-v /microservice-demo:/microservice-demo --link eureka:eureka"
    d.run "zuul",
      args: "-p 8080:8080 -v /microservice-demo:/microservice-demo --link eureka:eureka"
    d.run "turbine",
      args: "-p 8989:8989 -v /microservice-demo:/microservice-demo --link eureka:eureka"
  end
end
```

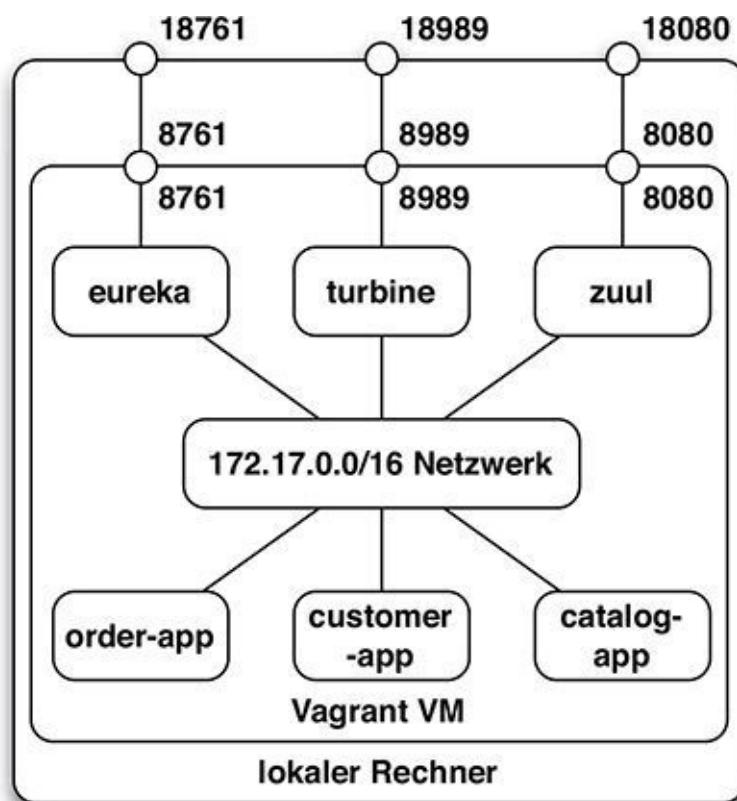
Listing 14–4 zeigt, wie die einzelnen Microservices in der Beispielanwendung über ein Netzwerk miteinander

Netzwerk in der Beispielanwendung

kommunizieren. Alle Docker-Container sind in dem Netzwerk mit IP-Nummern aus dem 172.17.0.0/16-Netzwerk erreichbar. Dieses Netzwerk legt Docker automatisch an und verbindet alle Docker-Container mit diesem Netzwerk. In dem Netzwerk sind alle Ports verfügbar, die mit EXPOSE in den Dockerfiles definiert worden sind. Auch die Vagrant VM ist an dieses Netzwerk angeschlossen. Durch die Docker-Links (siehe Abb. 14-4) kennen alle Docker-Container den Eureka-Container und können ihn unter dem Hostnamen eureka erreichen. Die anderen Microservices müssen über Eureka gesucht werden. Die weitere Kommunikation erfolgt dann über die IP-Adresse.

Zusätzlich hat die -p-Option für die Docker-Container in den d.run-Einträgen in Abbildung 14-4 Ports an die Vagrant VM gebunden. Diese Container sind über diese Ports der Vagrant VM erreichbar. Um sie auch von dem Rechner aus zu erreichen, auf dem die Vagrant VM läuft, gibt es ein Port-Mapping, das die Ports an den lokalen Rechner bindet. Dazu dienen die config.vm.network-Einträge im Vagrantfile. Beispielsweise kann der Port 8080 des Docker-Containers »zuul« unter dem Port 8080 in der Vagrant VM erreicht werden. Dieser Port ist auf dem lokalen Rechner unter dem Port 18080 erreichbar. Die URL <http://localhost:18080/> kontaktiert genau diesen Docker-Container.

Abb. 14-4 Netzwerk und Ports in der Beispielanwendung



Selber ausprobieren und experimentieren

Der Vorteil der Beispielanwendung ist, dass sie ohne großen Aufwand zum Laufen gebracht werden kann. Eine laufende Beispielanwendung ist die Basis für die anderen Experimente in diesem Kapitel.

Ein Hinweis: Das Vagrantfile definiert, wie viel Speicher und CPUs der virtuellen Maschine zugewiesen werden. Dazu dienen die Einstellungen v.memory und v.cpus, die das Listing nicht zeigt. Abhängig von dem benutzten Rechner kann es sein, dass die

Werte erhöht werden können, weil viel RAM oder CPUs vorhanden sind. Wenn das möglich ist, sollte man es auch tun, um die Anwendung zu beschleunigen.

Die Installation von Vagrant beschreibt <http://docs.vagrantup.com/v2/installation/index.html>. Vagrant benötigt eine Virtualisierungslösung wie VirtualBox. Die Installation von VirtualBox beschreibt <https://www.virtual-box.org/wiki/Downloads>. Beide Werkzeuge sind kostenlos.

Das Starten des Beispiels ist nur möglich, wenn der Code kompiliert worden ist. Die Anleitung dazu findet sich im Experiment von [Abschnitt 14.3](#). Danach kann man in das Verzeichnis docker-vagrant wechseln und das Beispiel mit dem Befehl `vagrant up` starten.

Um mit den verschiedenen Docker-Containern zu interagieren, muss man sich auf der virtuellen Maschine einloggen. Dazu dient der Befehl `vagrant ssh`, den man im Unterverzeichnis `docker-vagrant` ausführen muss. Dazu muss auf dem Rechner ein ssh-Client installiert sein. Unter Linux und Mac OS X ist häufig schon ein Client vorhanden. Unter Windows bringt die Installation von git einen ssh-Client mit – siehe dazu <http://git-scm.com/download/win>. Danach sollte `vagrant ssh` funktionieren.

Docker hat verschiedene nützliche Kommandos:

- `docker ps` liefert einen Überblick über die laufenden Docker-Container.
- **Logs** zeigt das Kommando `docker log <Name des Docker-Containers>`. `docker log -f < Name des Docker-Containers >` liefert ständig die gerade aktuellen Log-Informationen aus dem Container.
- `docker kill <Name des Docker-Containers>` beendet einen Docker-Container.
- Mit `docker rm <Name des Docker-Containers>` werden alle Daten gelöscht. Das ist nur möglich, wenn der Container zuvor beendet worden ist.

Nach dem Start der Anwendung kann man die Logs-Files der einzelnen Docker-Container anschauen.

Ein mögliches Experiment: Man kann einen Docker-Container beenden (`docker kill`) und auch die Daten des Containers löschen (`docker rm`). Die Kommandos müssen in der Vagrant VM eingegeben werden. Mit `vagrant provision` werden die fehlenden Docker-Container neu gestartet. Dieses Kommando muss auf dem Host eingegeben werden, auf dem Vagrant läuft. Wenn man eine Änderung an einem Docker-Container vornehmen will, kann man ihn einfach löschen, den Code neu kompilieren und mit `vagrant provision` das System wieder herstellen.

Weitere Vagrant-Kommandos:

- `vagrant halt` beendet die virtuelle Maschine. Sie kann mit `vagrant up` wieder gestartet werden.
- `vagrant destroy` zerstört die VM und alle gespeicherten Daten.

Weitere Experimente:

- Aktuell speichern die Docker-Container die Daten nicht ab. Nach einem Neustart sind die Daten verloren. Die genutzte HSQLDB-Datenbank kann Daten auch in eine Datei speichern. Dazu muss eine passende HSQLDB URL genutzt werden,

siehe http://hsqldb.org/doc/guide/dbproperties-chapt.html#dpc_connection_url. Spring Boot kann die JDBC URL aus den application.properties auslesen, siehe <http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html#boot-features-connect-to-production-database>. Nun kann der Container neu gestartet werden – was aber, wenn der Docker-Container neu erzeugt werden muss? Docker kann Daten auch außerhalb des eigentlichen Containers speichern, siehe <https://docs.docker.com/userguide/docker-volumes/>. Solche Möglichkeiten wären eine gute Basis für weitere Experimente. Außerdem kann statt HSQLDB eine andere Datenbank genutzt werden wie beispielsweise MySQL. Dazu muss ein eigener Docker-Container installiert werden, der die Datenbank enthält. Neben der Anpassung der JDBC URL muss außerdem ein JDBC-Treiber in das Projekt aufgenommen werden.

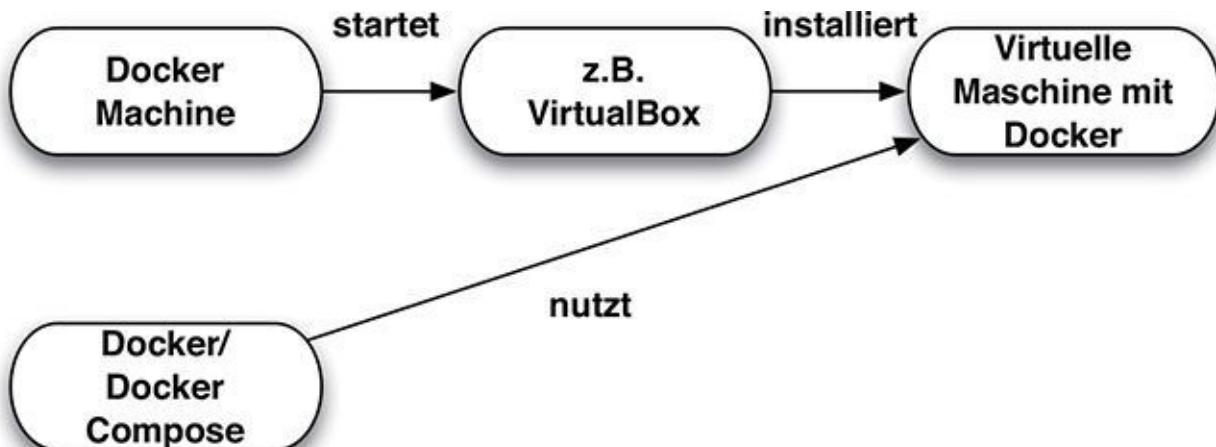
- <https://www.docker.com/tryit/> ist ein einfaches interaktives Tutorial, um Docker besser kennenzulernen. Das Durcharbeiten dauert nur wenige Minuten. Es zeigt die Nutzung von Docker ohne Dockerfiles und ist daher eine gute Ergänzung. Software muss nicht installiert werden, da das Tutorial vollständig im Browser läuft.
- Wie wird das Java Docker Image zusammengebaut? Das Dockerfile ist komplexer als die hier diskutierten. <https://docs.docker.com/reference/builder/> zeigt, welche Befehle es in den Dockerfiles gibt. Versuche, den Aufbau des Dockerfiles zu verstehen.

14.6 Docker Machine

Vagrant dient zur Installation von Umgebungen auf einem EntwicklerLaptop. Neben Docker kann Vagrant auch einfach Shell-Skripte für das Deployment nutzen. Für Produktionsumgebungen ist diese Lösung aber ungeeignet. Docker Machine [20] ist auf Docker spezialisiert. Es unterstützt viel mehr Virtualisierungslösungen und auch einige Cloud-Anbieter.

Abbildung 14–5 zeigt, wie Docker Machine eine Docker-Umgebung aufbauen kann: Zunächst wird auf Basis einer Virtualisierungslösung wie VirtualBox eine virtuelle Maschine installiert. Sie basiert auf boot2docker, einem sehr leichtgewichtigen Linux, das als Ablaufumgebung für Docker-Container gedacht ist. Darauf installiert Docker Machine eine aktuelle Version von Docker. Ein Kommando wie `docker-machine create —driver virtualbox dev` erzeugt beispielsweise eine neue Umgebung mit dem Namen `dev`, die auf einem VirtualBox-Rechner läuft.

Abb. 14–5 Docker Machine



Das Docker-Werkzeug kann nun mit diesem Rechner kommunizieren. Die Docker-Kommandozeilenwerkzeuge nutzen eine REST-Schnittstelle, um mit dem Docker-Server zu kommunizieren. Also muss das Kommandozeilenwerkzeug nur noch so konfiguriert werden, dass es mit dem Server passend kommuniziert. Unter Linux oder Mac OS X reicht der Befehl eval "\$(docker-machine env dev)", der Docker passend konfiguriert. Für Windows PowerShell wäre es docker-machine.exe env —shell powershell dev und für Windows cmd docker-machine.exe env —shell cmd dev.

Docker Machine erlaubt es also sehr einfach, eine oder mehrere Docker-Umgebungen zu installieren. Alle diese Umgebungen kann Docker Machine verwalten und das Docker-Kommandozeilenwerkzeug kann auf sie zugreifen. Da Docker Machine auch Technologien wie die Amazon Cloud oder VMware vSphere unterstützt, kann es zum Aufbau von Produktionsumgebungen dienen.

Selber ausprobieren und experimentieren

Die Beispielanwendung kann auch auf einer Umgebung laufen, die mit Docker Machine aufgesetzt wird.

Die Installation von Docker Machine beschreibt <https://docs.docker.com/machine/#installation>. Docker Machine benötigt eine Virtualisierungslösung wie VirtualBox. Die Installation von VirtualBox beschreibt <https://www.virtual-box.org/wiki/Downloads>.

Mit docker-machine create —virtualbox-memory "4096" —driver virtualbox dev kann nun eine Docker-Umgebung auf einer Virtual Box erzeugt werden. Ohne weitere Konfiguration ist die Speichereinstellung 1 GB, was für eine größere Anzahl Java Virtual Machines nicht ausreicht.

docker-machine ohne Parameter gibt einen Hilfetext aus und docker-machine create zeigt die Optionen für das Erzeugen einer neuen Umgebung. <https://docs.docker.com/machine/#using-docker-machine-with-a-cloud-provider> zeigt, wie Docker Machine auch mit einer Cloud genutzt werden kann. Also kann die Beispielanwendung sehr einfach auch auf einer Cloud-Umgebung gestartet werden.

Am Ende der eigenen Experimente dient docker-machine rm zum Löschen der Umgebung.

14.7 Docker Compose

Ein Microservice-System besteht typischerweise aus mehreren Docker-Containern. Sie müssen gemeinsam gebaut und in Produktion gebracht werden.

Dazu dient Docker Compose [21]. Es ermöglicht die Definition von Docker-Containern, die jeweils einen Service beherbergen. Als Format kommt YAML zum Einsatz.

Listing 14–5 Docker-Compose-Konfiguration für die Beispielanwendung

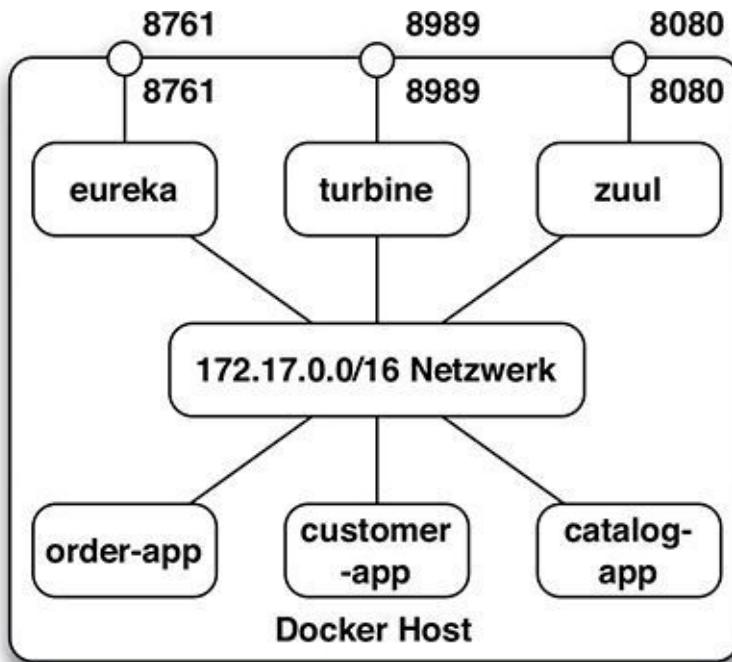
```
eureka:  
  build: ../microservice-demo/microservice-demo-eureka-server  
  ports:  
    - "8761:8761"  
customer:  
  build: ../microservice-demo/microservice-demo-customer  
  links:  
    - eureka  
catalog:  
  build: ../microservice-demo/microservice-demo-catalog  
  links:  
    - eureka  
order:  
  build: ../microservice-demo/microservice-demo-order  
  links:  
    - eureka  
zuul:  
  build: ../microservice-demo/microservice-demo-zuul-server  
  links:  
    - eureka  
  ports:  
    - "8080:8080"  
turbine:  
  build: ../microservice-demo/microservice-demo-turbine-server  
  links:  
    - eureka  
  ports:  
    - "8989:8989"
```

[Listing 14–5](#) zeigt die Konfiguration der Beispielanwendung. Sie besteht aus den verschiedenen Services. `build` referenziert jeweils das Verzeichnis mit dem Dockerfile. Das Dockerfile wird genutzt, um das Image für den Service zu erzeugen. `links` zeigt an, welche anderen Docker-Container der jeweilige Container im Zugriff haben soll. Alle Container erreichen unter dem Namen `eureka` den Eureka-Container.

Im Gegensatz zu der Vagrant-Konfiguration gibt es kein Java-Basis-Container, also einen Container, der nur eine Java-Installation enthält. Der Grund ist, dass Docker Compose nur Container unterstützt, die auch tatsächlich einen Service anbieten. Daher wird dieses Basisimage nun aus dem Internet geladen.

Außerdem werden bei den Docker-Compose-Containern die JAR-Dateien in die Docker Images kopiert, sodass die Images ausreichen, um die Microservices zu starten.

Abb. 14–6 Netzwerk bei Docker Compose



Es ergibt sich ein System, das dem Vagrant-System sehr ähnlich ist (Abb. 14–6). Die Docker-Container sind über ein eigenes Netzwerk verbunden. Nach außen ist nur Zuul für die Verarbeitung von Requests und Eureka für das Dashboard verfügbar. Sie laufen direkt auf einem Host, auf den dann zugegriffen werden kann.

Mit dem Aufruf `docker-compose build` wird das System auf Basis der Docker-Compose-Konfiguration erstellt. Es werden also die passenden Images erstellt. `docker-compose up` startet das System dann. Docker Compose arbeitet mit denselben Einstellungen wie das Docker-Kommandozeilenwerkzeug. Es kann auch mit Docker Machine zusammenarbeiten. Es ist also transparent, ob das System auf einer lokalen VM oder irgendwo in der Cloud erzeugt wird.

Selber ausprobieren und experimentieren

Die Beispielanwendung hat eine passende Docker-Compose-Konfiguration. Nach dem Aufbau einer Umgebung mit Docker Machine können mit Docker Compose die Docker-Container aufgebaut werden. Das `LIESMICH.md` im Verzeichnis `docker` beschreibt das dazu notwendige Vorgehen.

Wirf einen Blick auf das `docker-compose scale`-Kommando. Es dient zum Skalieren der Umgebung.

Ebenso können Services neu gestartet, die Logs analysiert und schließlich gestoppt werden. Nach dem Start der Anwendung kannst du diese Funktionalitäten ausprobieren.

Mesos (<http://mesos.apache.org/>) zusammen mit Mesosphere (<http://mesosphere.com/>), Kubernetes (<http://kubernetes.io/>) oder CoreOS (<http://coreos.com/>) bieten ähnliche Möglichkeiten wie Docker Compose und Docker Machine, aber sie sind für Server und Server Cluster gedacht. Die Docker-Compose- und Docker-Machine-Konfigurationen können eine gute Basis für den Betrieb der Anwendung auf diesen Plattformen sein.

14.8 Service Discovery

Abschnitt 8.9 hat die allgemeinen Prinzipien für Service Discovery eingeführt. Die Beispielanwendung nutzt Eureka [9] für Service Discovery.

Eureka ist ein REST-basierter Server, bei dem sich Services registrieren können, sodass andere Services ihre Daten abfragen können. Im Wesentlichen kann jeder Service unter seinem Namen eine URL hinterlegen. Unter dieser URL können andere Services ihm dann REST-Nachrichten schicken.

Eureka unterstützt Replikation auf mehrere Server und Caches auf dem Client. Dadurch ist es gegen Ausfälle einzelner Eureka-Server gut abgesichert und kann Anfragen schnell beantworten. Änderungen an den Daten müssen über die Server repliziert werden. Daher kann es einige Zeit dauern, bis sie tatsächlich überall umgesetzt sind. In dieser Zeit sind die Daten inkonsistent: Jeder Server hat einen anderen Datenbestand.

Außerdem unterstützt Eureka Amazon Web Services, weil Netflix es in dieser Umgebung nutzt. Beispielsweise kann Eureka recht einfach mit der Amazon-Skalierung kombiniert werden.

Eureka überwacht die registrierten Services und entfernt sie aus der Liste der Server, wenn der Eureka-Server sie nicht mehr erreichen kann.

Eureka ist die Basis vieler anderer Services aus dem Netflix-Stack und von Spring Cloud. Durch eine einheitliche Service Discovery sind andere Aspekte wie beispielsweise Routing sehr einfach umsetzbar.

Damit eine Spring-Boot-Anwendung sich bei einem Eureka-Server registriert und andere Microservices finden kann, muss die Anwendung mit `@EnableDiscoveryClient` oder `@EnableEurekaClient` annotiert werden. Außerdem muss eine Abhängigkeit zu `spring-cloud-starter-eureka` in das `pom.xml` eingebaut werden. Die Anwendung registriert sich automatisch bei dem Eureka-Server und kann auf andere Microservices zugreifen. Der Zugriff auf andere Microservices erfolgt in der Beispielanwendung zusammen mit einem Load Balancer und wird in Abschnitt 14.11 näher erläutert.

Eine Konfiguration der Anwendung ist notwendig, um beispielsweise den zu benutzenden Eureka-Server zu definieren. Dazu dient die Datei `application.properties` (Listing 14–6). Sie liest Spring Boot automatisch aus, um die Anwendung zu konfigurieren. Dieser Mechanismus kann auch dazu genutzt werden, um eigenen Code konfigurierbar zu machen. Im vorliegenden Fall dienen die Werte dazu, den Eureka-Client zu konfigurieren:

- Die erste Zeile definiert den Eureka-Server. In der Beispielanwendung wird der Docker-Link verwendet, der unter dem Host-Namen »eureka« den Eureka-Server zur Verfügung stellt.
- Mit dem `leaseRenewalIntervalInSeconds` wird geregelt, wie oft ein Datenabgleich zwischen dem Client und dem Server stattfindet. Da die Daten auf jedem Client lokal in einem Cache gehalten werden, muss ein neuer Service zunächst seinen eigenen Cache aufbauen, ihn auf den Server replizieren und dann werden die Daten auf die Clients

repliziert. In einer Testumgebung ist es wichtig, Änderungen an dem System schnell zu sehen, sodass die Beispielanwendung fünf Sekunden statt dem Vorgabewert von 30 Sekunden nutzt. In Produktion mit vielen Clients sollte der Wert erhöht werden. Sonst verschlingt der Abgleich der Informationen viele Ressourcen, obwohl die Informationen im Wesentlichen unverändert bleiben.

- `spring.application.name` dient als Name für den Service bei der Registrierung gegenüber Eureka. Bei der Registrierung wird der Name in Großbuchstaben umgewandelt. Dieser Service wäre also unter dem Namen “CUSTOMER” bei Eureka bekannt.
- Von jedem Service kann es mehrere Instanzen geben, um Ausfallsicherheit und Lastverteilung zu ermöglichen. Die `instanceId` muss für jede Instanz eines Service eindeutig sein. Daher hat sie noch eine Zufallszahl, die zu einer Eindeutigkeit führt.
- `preferIpAddress` sorgt dafür, dass sich die Microservices mit ihrer IP-Adresse registrieren und nicht mit ihrem Hostnamen. In einer Docker-Umgebung sind die Hostnamen leider nicht von den anderen Hosts aus einfach auflösbar. Dieses Problem wird durch die Nutzung der IP-Adressen umgangen.

Listing 14–6 Ausschnitt aus application.properties mit Eureka-Konfiguration

```
eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/  
eureka.instance.leaseRenewalIntervalInSeconds=5  
spring.application.name=catalog  
eureka.instance.metadataMap.instanceId=catalog:${random.value}  
eureka.instance.preferIpAddress=true
```

Der Eureka-Server ([Listing 14–7](#)) ist eine einfache Spring-Boot-Anwendung, die durch `@EnableEurekaServer` zu einem Eureka-Server wird. Außerdem muss der Server eine Abhängigkeit zu `spring-cloud-starter-eureka-server` haben.

Listing 14–7 Eureka-Server

```
@EnableEurekaServer  
@EnableAutoConfiguration  
public class EurekaApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(EurekaApplication.class, args);  
    }  
}
```

Der Eureka-Server bietet ein Dashboard an, das die registrierten Services zeigt. In der Beispielanwendung steht dieser unter `http://localhost:18761/` (Vagrant) oder auf dem Docker Host unter dem Port 8761 zur Verfügung. [Abbildung 14–7](#) zeigt einen Screenshot des Eureka-Dashboards für die Beispielanwendung. Zu erkennen sind die drei Microservices und der Zuul-Proxy, den der nächste Abschnitt behandelt.

Abb. 14–7 Eureka-Dashboard

The screenshot shows the Spring Eureka dashboard at localhost:18761. The top navigation bar includes links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main section is titled 'System Status' and displays various configuration parameters:

Environment	Current time	2015-04-03T08:20:56 +0000
Data center	Uptime	00:04
	Lease expiration enabled	true
	Renews threshold	7
	Renews (last min)	10

The 'DS Replicas' section shows a single entry for 'localhost'.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 172.17.0.25:catalog:a5fb7f7dc1dfbb6cb83c55c198ccb637
CUSTOMER	n/a (1)	(1)	UP (1) - 172.17.0.24:customer:a0a7d00a563263391263ae9994720148
ORDER	n/a (1)	(1)	UP (1) - 172.17.0.26:order:903933c9d8fcfd6d56578051df2e7ef4e
ZUUL	n/a (1)	(1)	UP (1) - 017f72e4c4a3

14.9 Kommunikation

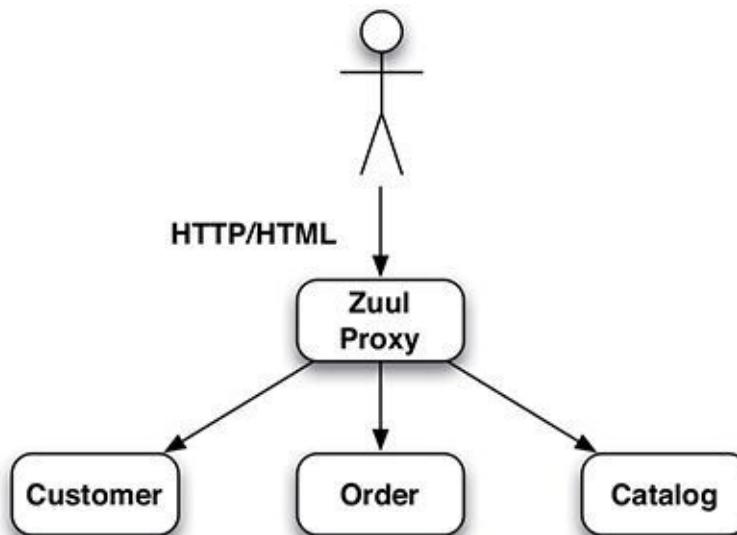
Kapitel 9 hat erläutert, wie Microservices miteinander kommunizieren und integriert werden können. Die Beispielanwendung nutzt REST für die interne Kommunikation. Die REST-Endpunkte können zwar auch von außen angesprochen werden, aber wichtiger ist die Web-Schnittstelle, die das System den Nutzern anbietet. Die REST-Umsetzung nutzt HATEOAS. Die Liste aller Bestellungen enthält beispielsweise Links zu den einzelnen Bestellungen. Das setzt Spring Data REST automatisch so um. Aber Links zu dem Kunden und den Waren in der Bestellung fehlen.

Die Nutzung von HATEOAS kann noch weiter gehen: Zu jedem Order kann das JSON ein Link zu einem HTML-Dokument enthalten – und umgekehrt. Dadurch kann ein JSON-REST-basierter Service eine HTML-Seite erzeugen, um Daten anzuzeigen oder zu modifizieren. Beispielsweise kann solcher HTML-Code die Darstellung einer Ware in einer Bestellung realisieren. Da der HTML-Code für die Ware vom Catalog-Team kommt, kann das Catalog-Team Änderungen an der Darstellung selber durchführen – auch wenn die Waren in einem anderen Modul angezeigt werden. Hier kann REST auch helfen: Eigentlich sind HTML und JSON nur Repräsentationen derselben Ressource, die durch eine URL adressiert werden kann. Über Content Negotiation kann die richtige Repräsentation der Ressource als JSON oder HTML ausgewählt werden (siehe Abschnitt 9.2).

Eingehende Anfragen leitet der Zuul-Proxy [7] an die jeweiligen Microservices weiter. Der Zuul-Proxy ist ein eigener Java-Prozess. So ist nach außen nur eine URL bekannt, aber intern werden die Aufrufe auf verschiedenen Microservices umgesetzt. Dadurch kann das System intern die Struktur der Microservices ändern und nach außen immer noch eine URL anbieten. Außerdem kann Zuul auch eigene Webressourcen bereitstellen. Im Beispiel liefert Zuul die erste HTML-Seite, die ein Nutzer sieht.

Zuul: Routing

Abb. 14–8 Zuul-Proxy in der Beispielanwendung



Zuul muss wissen, welche Anfragen auf welchen Microservice weitergeleitet werden sollen. Ohne weitere Konfiguration wird ein Aufruf an eine URL, die mit /customer beginnt, an den Microservice mit dem Namen CUSTOMER bei Eureka weitergeleitet. Dadurch wird die interne Benennung der Microservices nach draußen sichtbar. Aber das Routing kann umkonfiguriert werden. Außerdem können Zuul-Filter die Anfragen ändern, um generelle Aspekte im System umzusetzen. So gibt es eine Integration mit Spring Cloud Security, um die Sicherheits-Token an die Microservices weiterzugeben. Solche Filter können auch genutzt werden, um bestimmte Requests an andere Server weiterzuleiten. Beispielsweise können Anfragen an Server mit mehr Analysemöglichkeiten weitergeleitet werden, um Fehlersituationen zu untersuchen. Oder es kann eine Funktionalität eines Microservice durch einen anderen Microservice abgelöst werden.

Die Implementierung des Zuul-Proxys mit Spring Cloud ist sehr einfach und entspricht im Wesentlichen dem Eureka-Server aus Listing 14–7. Statt @EnableEurekaServer aktiviert @EnableZuulProxy den Zuul-Proxy. Als zusätzliche Abhängigkeit muss in der Anwendung spring-cloud-starter-zuul z. B. in der Maven-Build-Konfiguration eingetragen sein, das dann die restlichen Abhängigkeiten zu Zuul in die Anwendung integriert.

Die Alternative zu einem Zuul-Proxy ist ein Zuul-Server. Er hat kein eingebautes Routing, sondern nutzt nur Filter. Ein Zuul-Server wird mit @EnableZuulServer aktiviert.

Selber ausprobieren und experimentieren

- Erweitere die Anwendung so, dass eine Bestellung auch Links zu dem Kunden und den Waren enthält und so HATEOAS besser umsetzt.

- Ergänze die JSON-Dokumente für die Kunden, Waren und Bestellungen um Links auf die Formulare.
- Ändere die Darstellung der Order so, dass für die Ware HTML aus dem Catalog-Service genutzt wird. Dazu muss in der Order-Komponente passender JavaScript-Code eingefügt werden, der HTML-Code aus dem Catalog nachlädt.
- Setze einen eigenen Zuul-Filter um (siehe [8]). Der Filter kann beispielsweise einfach nur die Requests ausgeben.
- Führe ein zusätzliches Routing ein zu einer externen URL. Beispielsweise könnte /google auf <http://google.com/> umlenken. Siehe dazu die Spring-Cloud-Referenz-Dokumentation.
- Füge eine Authentifizierung und Autorisierung mit Spring Cloud Security ein. Siehe <http://cloud.spring.io/spring-cloud-security/>.

14.10 Resilience

Resilience bedeutet, dass Microservices mit dem Ausfall anderer Microservices umgehen können und trotzdem weiter laufen. [Abschnitt 10.5](#) hat dieses Thema dargestellt.

Das Beispiel setzt Resilience mit Hystrix [10] um. Diese Bibliothek dient dazu, Aufrufe so abzusichern, dass es auch bei einem Ausfall eines Systems zu keinem Problem kommt. Wenn ein mit Hystrix abgesichertes System aufgerufen wird, geschieht das in einem anderen Thread als der eigentliche Aufruf. Dieser Thread kommt aus einem eigenen Thread Pool. Dadurch ist es relativ einfach, einen Timeout bei einem Aufruf umzusetzen.

Hystrix implementiert einen Circuit Breaker. Wenn Circuit Breaker Aufrufe zu einem Fehler führen, öffnet sich der Circuit Breaker nach einer bestimmten Zahl Fehler. Weitere Aufrufe werden nicht mehr an das aufgerufene System weitergeleitet, sondern führen sofort zu einem Fehler. Nach einem bestimmten Zeitfenster schließt sich der Circuit Breaker wieder, sodass die Aufrufe an das eigentliche System weitergeleitet werden. Das genaue Verhalten kann konfiguriert werden [11]. Dabei kann der Anteil der Aufrufe festgelegt werden, die in einem konfigurierten Zeitfenster zu einem Fehler führen müssen, damit sich der Circuit Breaker öffnet. Außerdem kann das Zeitfenster definiert werden, in dem der Circuit Breaker offen ist und keine Anfragen an das System weiterschickt.

Spring Cloud nutzt für die Konfiguration von Hystrix Hystrix mit Annotationen Annotationen aus dem Projekt `hystrix-javanica`. Es ist ein Teil von `hystrix-contrib` [12]. Die annotierten Methoden werden entsprechend geschützt. Ohne diesen Ansatz müssten Hystrix Commands geschrieben werden, die durch die Nutzung der Annotationen vollständig entfallen.

Um Hystrix in einer Spring-Cloud-Anwendung nutzen zu können, muss die Anwendung mit `@EnableCircuitBreaker` bzw. `@EnableHystrix` annotiert werden. Außerdem muss das Projekt eine Abhängigkeit zu `spring-cloud-starter-hystrix` haben.

[Listing 14–8](#) zeigt einen Ausschnitt aus der Klasse `CatalogClient` des Order-Projekts aus

dem Beispielprojekt. Die Methode `findAll()` ist mit `@HystrixCommand` annotiert. Dadurch werden die Verarbeitung in einem anderen Thread und der Circuit Breaker aktiviert. Der Circuit Breaker kann konfiguriert werden – im Beispiel wird die Anzahl der Aufrufe, die auf einen Fehler laufen müssen, bevor der Circuit Breaker sich öffnet, auf 2 gesetzt. Das Beispiel definiert außerdem eine `fallbackMethod`. Hystrix ruft sie auf, wenn die eigentliche Methode einen Fehler erzeugt. Die Logik in `findAll()` speichert das letzte Ergebnis in einem Cache, der dann von der `fallbackMethod` zurückgegeben wird, ohne das eigentliche System aufzurufen. So kann bei Ausfall des Microservice immer noch eine Antwort zurückgegeben werden, die aber möglicherweise nicht mehr aktuell ist.

Listing 14–8 Beispiel für eine mit Hystrix abgesicherte Methode

```
@HystrixCommand(fallbackMethod = "getItemsCache",
    commandProperties = {
        @HystrixProperty
        (name = "circuitBreaker.requestVolumeThreshold",
        value = "2") })
public Collection<Item> findAll() {
    this.itemsCache = ...
    ...
    return pagedResources.getContent();
}

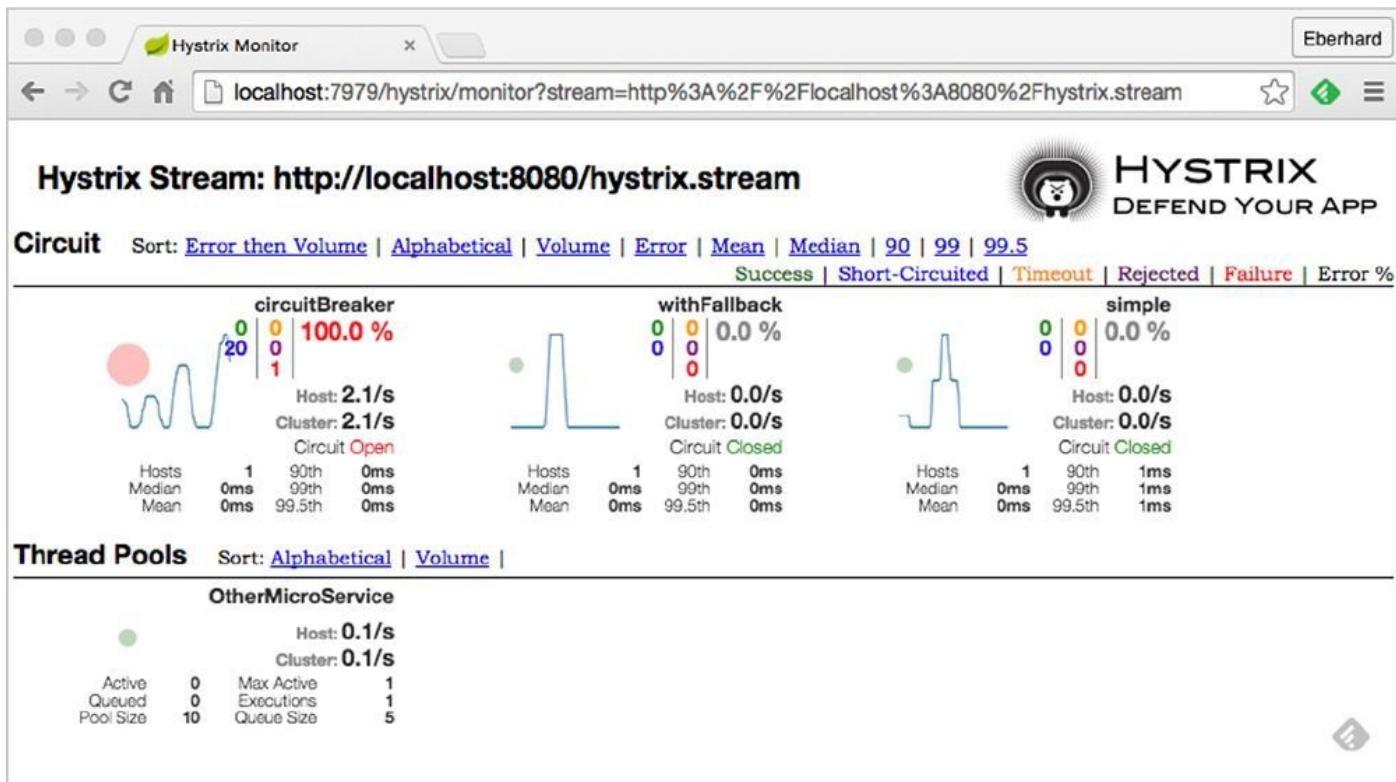
private Collection<Item> getItemsCache() {
    return itemsCache;
}
```

Ob ein Circuit Breaker gerade offen oder geschlossen ist, gibt einen Eindruck davon, wie gut das System gerade läuft. Hystrix bietet dazu eigene Monitoring-Daten an.

*Monitoring mit dem Hystrix-
Dashboard*

Ein Hystrix-System liefert solche Daten als Strom von JSON-Dokumenten über HTTP. Das Hystrix-Dashboard kann die Daten in einer Weboberfläche visualisieren. Das Dashboard stellt alle Circuit Breaker mit der Anzahl der Requests und dem Zustand (offen/geschlossen) dar ([Abb. 14–9](#)). Außerdem zeigt es den Zustand der Thread Pools.

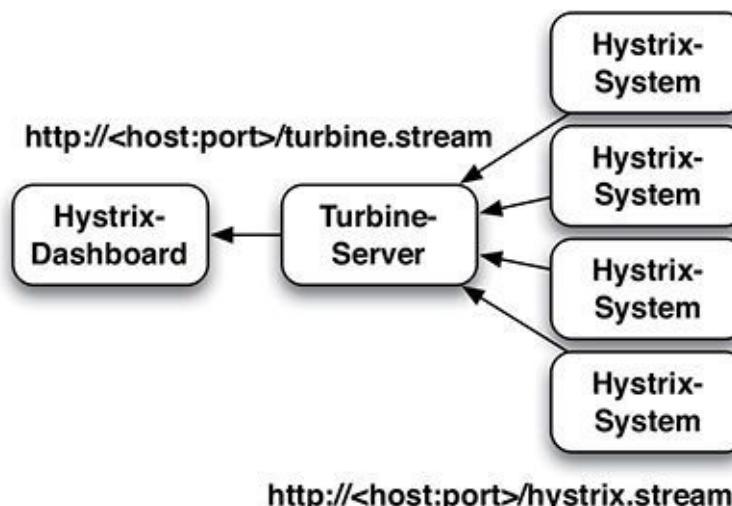
Abb. 14–9 Beispiel für ein Hystrix-Dashboard



Eine Spring-Boot-Application muss die Annotation `@EnableHystrix-Dashboard` tragen und eine Abhängigkeit zu `spring-cloud-starter-hystrix-dashboard` haben, um ein Hystrix-Dashboard darzustellen.

In einer komplexen Microservices-Umgebung ist es kaum sinnvoll, dass jeder Microservice die Informationen über den Zustand seiner Hystrix Circuit Breaker darstellt. Es sollte der Zustand aller Circuit Breaker im gesamten System auf einer einzigen Webseite zu finden sein. Um die Daten der verschiedenen Hystrix-Systeme auf einem Dashboard anzuzeigen, gibt es ein eigenes Projekt: Turbine. Abbildung 14–10 zeigt das Vorgehen von Turbine: Die einzelnen Streams des Hystrix-Monitorings stehen unter URLs wie `http://<host:port>/hystrix.stream` zur Verfügung. Der Turbine-Server fragt sie ab und stellt sie konsolidiert unter der URL `http://<host:port>/turbine.stream` zur Verfügung. Diese URL kann das Dashboard nutzen, um die Informationen aller Circuit Breaker der verschiedenen Microservices anzuzeigen.

Abb. 14–10 Hystrix-Daten mit Turbine konsolidieren



Turbine läuft in einem eigenen Prozess. Mit Spring Boot ist der Turbine-Server eine

einfache Anwendung, die mit `@EnableTurbine` und `@EnableEurekaClient` annotiert ist. Im Beispiel trägt sie zusätzlich die Annotation `@EnableHystrixDashboard`, sodass sie auch das Hystrix-Dashboard anzeigt.

Offen ist, welche Daten der Turbine-Server überhaupt konsolidiert. Das legt die Konfiguration der Anwendung fest. [Listing 14–9](#) zeigt die Konfiguration des Turbine-Servers aus dem Beispielprojekt. Sie entspricht den `application.properties`-Dateien, ist aber in YAML geschrieben. Die Konfiguration legt den Wert »ORDER« für `turbine.aggregator.clusterConfig` fest. Das ist der Name der Anwendung in Eureka. `turbine.aggregator.appConfig` ist der Name des Datenstroms im Turbine-Server. Im Hystrix-Dashboard muss eine URL wie `http://172.17.0.10:8989/turbine.stream?cluster=ORDER` verwendet werden, um den Datenstrom auszulesen. Die URL nutzt die IP-Adresse des Turbine-Servers, die aus dem Eureka-Dashboard ausgelesen werden kann. Das Dashboard greift auf den Turbine-Server über das Netzwerk zwischen den Docker-Containern zu.

Listing 14–9 Konfiguration application.yml

```
turbine:  
  aggregator:  
    clusterConfig: ORDER  
    appConfig: order
```

Selber ausprobieren und experimentieren

- Lege mit der Beispielanwendung einige Bestellungen an. Beende mit `docker kill` den Catalog-Docker-Container. Die Nutzung ist mit Hystrix abgesichert. Was passiert?
- Was passiert, wenn der Customer-Docker-Container ebenfalls beendet wird? Die Nutzung dieses Microservice ist nicht mit Hystrix abgesichert.
- Sichere die Nutzung des Customer-Docker-Containers ebenfalls mit Hystrix ab. Ändere dazu die Klasse `CustomerClient` aus dem Order-Projekt. `CatalogClient` kann ein Vorbild sein.
- Ändere die Konfiguration von Hystrix für den Catalog-Microservice. Siehe [11] für die Konfigurationsoptionen und [Listing 14–8](#) (Catalog-Client aus dem Order-Projekt) für die Nutzung der Hystrix-Annotationen. Beispielsweise sind andere Zeitfenster für das Öffnen oder Schließen des Circuit Breakers denkbar.

14.11 Load Balancing

Für das Load Balancing nutzt die Beispielanwendung Ribbon [13]. Viele Load Balancer sind proxy-basiert. In diesem Modell schicken die Clients alle Anfragen an einen Load Balancer. Der Load Balancer läuft als eigener Server und schickt die Nachrichten dann an einen Webserver weiter – oft abhängig von der aktuellen Auslastung der Webserver.

Ribbon setzt Client Side Load Balancing um: Der Client hat alle Informationen, um den richtigen Server anzusprechen. Der Client spricht den Server direkt an und verteilt die Last selbstständig über verschiedene Server. In der Architektur gibt es keinen Flaschenhals, da es keinen zentralen Server gibt, über den alle Aufrufe laufen. Zusammen

mit der Replikation der Daten von Eureka ist Ribbon recht widerstandsfähig: Solange der Client noch läuft, kann er noch Anfragen absetzen. Der Ausfall eines Proxy Load Balancers würde alle Anfragen an den Server zum Stillstand bringen.

Eine dynamische Skalierung ist in diesem System sehr einfach: Eine neue Instanz wird gestartet, trägt sich bei Eureka ein und dann leiten die Ribbon-Clients Last auf die Instanz um.

Wie schon im Abschnitt über Eureka ([Abschnitt 14.8](#)) diskutiert, können die Daten über die Server inkonsistent sein. Weil die Daten nicht aktuell sind, könnten Server angesprochen werden, die eigentlich aus dem Load Balancing schon ausgenommen sein sollten.

Spring Cloud vereinfacht auch die Nutzung von [Ribbon mit Spring Cloud](#) Ribbon. Die Anwendung muss mit `@RibbonClient` annotiert sein. Dabei kann auch ein Name der Anwendung definiert werden. Außerdem muss die Anwendung eine Abhängigkeit zu `spring-cloud-starter-ribbon` haben. Dann kann mit Code wie aus [Listing 14–10](#) eine Instanz eines Microservice ausgelesen werden. Dazu nutzt der Code den Eureka-Namen des Microservice.

Listing 14–10 Ermitteln eines Servers mit Ribbon Load Balancing

```
ServiceInstance instance = loadBalancer.choose("CATALOG");
String url = "http://" + instance.getHost() + ":" +
+ instance.getPort() + "/catalog/";
```

Die Nutzung kann auch weitgehend transparent sein. Dazu zeigt [Listing 14–11](#) die Nutzung des `RestTemplates` mit Ribbon. Das ist eine Spring-Klasse, mit der REST-Server angesprochen werden können. In dem Listing wird das `RestTemplate` von Spring Cloud in das Objekt injiziert, weil es mit `@Autowired` annotiert ist. Der Aufruf in `callMicroservice()` sieht aus, als würde er einen Server namens `stores` ansprechen. In Wirklichkeit wird aber dieser Name genutzt, um bei Eureka einen Server zu suchen, und an den Server wird der REST-Aufruf geschickt. Das geschieht über Ribbon, sodass die Last auch verteilt wird.

Listing 14–11 Nutzung von Ribbon mit dem RestTemplate

```
@RibbonClient(name = "ribbonApp")
...
public class RibbonApp {

    @Autowired
    private RestTemplate restTemplate;

    public void callMicroservice() {
        Store store = restTemplate.
            getForObject("http://stores/store/1", Store.class);
    }
}
```

Selber ausprobieren und experimentieren

Der Order-Microservice verteilt die Last auf mehrere Instanzen des Customer- und Catalog-Microservice – wenn mehrere Instanzen vorhanden sind. Ohne weitere Maßnahmen läuft nur jeweils eine Instanz. Der Order-Microservice gibt im Log aus,

welchen Catalog- oder Customer-Microservice er anspricht. Setze eine Bestellung ab und beobachte, welche Services angesprochen werden.

- Starte dann einen weiteren Catalog Microservice. Dazu dient der Befehl:

```
docker run -v /microservice-demo:/microservice-demo —  
link eureka:eureka catalog-app
```

- Überprüfe, ob der Container läuft, und beobachte die Log-Ausgaben.
- Lege einen neuen Datensatz mit einer neuen Ware an. Wird sie immer in der Auswahl der Waren angezeigt? Hinweis: Die Datenbank läuft im Prozess des Microservice – also hat jede Microservice-Instanz auch ihre eigene Datenbank.
- Das Experiment in [Abschnitt 14.4](#) zeigt übrigens die wesentlichen Befehle für den Umgang mit Docker.

14.12 Integration anderer Technologien

Spring Cloud und der gesamte Netflix-Stack basieren auf Java. Daher scheint die Nutzung der Infrastruktur von anderen Programmiersprachen oder Plattformen nicht möglich zu sein. Es gibt eine Lösung: Der eigentlichen Anwendung wird ein Sidecar zur Seite gestellt. Das Sidecar ist in Java geschrieben und nutzt die Java Libraries, um sich in die Netflix-Infrastruktur zu integrieren. Das Sidecar besorgt beispielsweise die Registrierung in Eureka und das Heraussuchen anderer Microservices. Netflix selber bietet dazu das Prana-Projekt [14]. Die Spring-Cloud-Lösung ist in der Dokumentation [15] erläutert. Das Sidecar läuft dann in einem eigenen Prozess und dient dem eigentlichen Microservice als Schnittstelle zu der Microservice-Infrastruktur. So können auch andere Programmiersprachen und Plattformen sehr einfach in eine Netflix- oder Spring-Cloud-Umgebung integriert werden.

14.13 Tests

Für die Entwickler der Microservices gibt es Testanwendungen. Sie sind so geschrieben, dass sie ohne Microservice-Infrastruktur auskommen und ohne weitere Microservices – im Gegensatz zu den Produktionsanwendungen. So können die Entwickler ohne komplexe Infrastruktur die Microservices laufen lassen.

Die Klasse OrderTestApp im Order-Projekt enthält eine solche Testanwendung. Die Anwendungen haben eine eigene Konfigurationsdatei `application-test.properties` mit speziellen Einstellungen im Verzeichnis `src/test/resources`. Die Einstellungen vermeiden, dass die Anwendungen sich bei der Service Discovery Eureka registrieren. Außerdem erhalten sie andere URLs für die abhängigen Microservices. Diese Konfiguration nutzt die Testanwendung, weil sie ein passendes Spring-Profil verwendet. Alle JUnit-Tests nutzen diese Einstellungen auch, sodass sie ohne abhängige Services laufen können.

Die URL für die abhängigen Microservices bei der Stubs Testanwendung und den JUnit-Tests zeigen auf Stubs.

Das sind vereinfachte Microservices, die nur einen Teil der Funktionalitäten anbieten. Sie laufen im selben Java-Prozess wie die eigentlichen Microservices oder JUnit-Tests. Letztendlich muss für die Entwicklung eines Microservice nur ein einziger Java-Prozess gestartet werden, wie man es auch von der normalen Java-Entwicklung gewohnt ist. Die Stubs könnten anders implementiert werden – beispielsweise mit einer anderen Programmiersprache oder gar einem Webserver, der bestimmte statische Dokumente zurückgibt, die Testdaten repräsentieren (siehe [Abschnitt 11.6](#)).

Die Stubs erleichtern die Entwicklung. Wenn jeder Entwickler eine vollständige Umgebung mit allen Microservices zur Entwicklung nutzen soll, benötigt das viele Hardware-Ressourcen und viel Aufwand, um die Umgebung ständig aktuell zu halten. Die Stubs vermeiden dieses Problem, da keine abhängigen Microservices für die Entwicklung notwendig sind. Durch die Stubs ist der Aufwand zum Starten eines Microservice kaum größer als der Aufwand für eine normale Java-Anwendung.

In einem realen Projekt können die Teams Stubs parallel zur Entwicklung der eigentlichen Microservices implementieren. Das Customer-Team kann neben dem eigentlichen Service auch einen Stub für den Customer-Microservice implementieren, der von den anderen Microservices für die Entwicklung genutzt werden kann. Dadurch ist sichergestellt, dass der Stub dem Microservice weitgehend entspricht und bei Änderungen nachgezogen wird. Der Stub kann in einem eigenen Projekt gepflegt werden, das die anderen Teams dann nutzen können.

Es muss sichergestellt sein, dass die Stubs sich so verhalten wie die Microservices, die sie simulieren. Außerdem muss ein Microservice die Erwartungen an die Schnittstelle eines anderen Microservice definieren. Dazu dienen Consumer-Driven Contract Tests (siehe [Abschnitt 11.7](#)). Sie werden von dem Team geschrieben, das die Microservices verwendet. Im Beispiel wäre es das Team, das für den Order-Microservice verantwortlich ist. Im Order-Microservice finden sich die Consumer-Driven Contract Tests in den Klassen `CatalogConsumerDrivenContractTest` und `CustomerConsumerDrivenContractTest`. Dort laufen sie, um die Stubs auf Korrektheit zu testen.

Noch wichtiger als die korrekte Funktion der Stubs ist die korrekte Funktion der Microservices. Daher sind die Consumer-Driven Contract Tests auch im Customer- und im Catalog-Projekt enthalten. Dort laufen sie gegen die tatsächlich implementierten Microservices. So ist sichergestellt, dass sowohl die Stubs als auch die echten Microservices dieser Spezifikation entsprechen. Wenn die Schnittstelle geändert werden soll, können diese Tests zur Absicherung verwendet werden. Es ist die Aufgabe der benutzten Microservices – Customer und Catalog im Beispiel –, diese Tests zu erfüllen. So können die Anforderungen des Order-Microservice an den Customer- und Catalog-Microservice formal festgehalten und überprüft werden. Die Consumer-Driven Contract Tests sind letztendlich eine formale Definition der abgestimmten Schnittstelle.

In dem Beispiel sind die Consumer-Driven Contract Tests Teil des Customer- und Catalog-Projekts, um die Einhaltung der Schnittstelle zu überprüfen. Außerdem sind sie Teil des Order-Projekts, um die korrekte Funktion der Stubs zu überprüfen. In einem realen Projekt sollte das Kopieren der Tests unterbunden werden. Die Consumer-Driven Contract Tests können zusammen mit den Microservices in einem Projekt liegen. Dann

Consumer-Driven Contract Test

müssen alle Teams auf die Microservice-Projekte Zugriff haben, um die Tests ändern zu können. Oder sie liegen in den Projekten der Teams, die den Microservice nutzen. Dann muss der getestete Microservice die Tests aus den anderen Projekten zusammensammeln und ausführen.

In einem realen Projekt ist es auch nicht zwingend, Stubs durch Consumer-Driven Contract Tests abzusichern. Ziel der Stubs ist ja gerade, eine einfachere Implementierung anzubieten als die eigentlichen Microservices – daher werden die Funktionalitäten auch anders sein und Consumer-Driven Contract Tests brechen.

Selber ausprobieren und experimentieren

- Füge ein Feld in die Daten von Catalog oder Customer ein. Funktioniert das System weiterhin? Warum?
- Lösche ein Feld in der Implementierung des Servers für Catalog oder Customer. Wo fällt das Problem auf? Warum?
- Ersetze die selbstgeschriebenen Stubs mit Stubs, die ein Werkzeug aus [Abschnitt 11.6](#) nutzen.
- Ersetze die Consumer-Driven Contract Tests mit Tests, die ein Werkzeug aus [Abschnitt 11.7](#) nutzen.

Erfahrungen mit JVM-basierten Microservices in der Amazon Cloud von Sascha Möllering, zanox AG

In den letzten Monaten hat zanox eine leichtgewichtige Microservices-Architektur in Amazon Web Services (AWS) implementiert, die in mehreren AWS-Regionen parallel läuft. Die Regionen unterteilen die Amazon-Cloud in Bereiche wie US-East oder EU-West mit jeweils eigenen Rechenzentren. Sie arbeiten völlig unabhängig voneinander und tauschen keinerlei Daten direkt aus. Unterschiedliche AWS-Regionen werden genutzt, da Latenz für diese Art von Applikationen sehr wichtig ist und durch ein Routing auf Basis von Latzenzen (»latency based routing«) minimiert wird. Grundsätzliches Ziel war es darüber hinaus, die Architektur event-driven zu konzipieren. Weiterhin sollten die einzelnen Services nicht direkt miteinander kommunizieren, sondern durch Message-Queues bzw. Bus-Systeme voneinander getrennt werden. Zentraler Synchronisierungspunkt der unterschiedlichen Regionen ist ein Apache Kafka-Cluster als Message-Bus im zanox-Rechenzentrum. Jeder Service ist als zustandslose Applikation implementiert. Der Zustand wird in externen Systemen wie den Bussystemen, Amazon ElastiCache (basierend auf der NoSQL-Datenbank Redis), der Data-Stream-Processing-Lösung Amazon Kinesis und der NoSQL-Datenbank Amazon DynamoDB abgelegt. Basis für die Implementierung der einzelnen Services ist die JVM. Bei der Wahl des Frameworks haben wir uns für Vert.x und den Embedded Webserver Jetty entschieden. Alle Applikationen wurden von uns self-contained entwickelt, sodass am Ende des Build-Prozesses ein Fat-JAR erzeugt wird, das sehr leicht über java -jar gestartet werden kann.

Es müssen keine weiteren Komponenten oder gar ein Application-Server installiert

werden. Vert.x kommt als Basisframework für den HTTP-Teil der Architektur zum Einsatz. Innerhalb der Applikation wird aus Gründen der Performance nahezu vollständig asynchron gearbeitet. Für die übrigen Komponenten nutzen wir Jetty als Framework: Diese agieren entweder als Kafka/Kinesis-Consumer oder aktualisieren den Cache auf Redis-Basis für die HTTP-Schicht. Alle angesprochenen Applikationen werden in Docker-Containern ausgeliefert. Damit kann ein einheitlicher Deployment-Mechanismus unabhängig von der eingesetzten Technologie verwendet werden. Um die Services in unterschiedlichen Regionen autark ausliefern zu können, wurde in jeder Region eine eigene Docker Registry aufgesetzt, die die Docker Images in einem S3 Bucket abspeichert. S3 ist ein Amazon-Service, der die Bereitstellung auch großer Dateien unterstützt.

Wer Cloud Services nutzen möchte, sollte für sich die Frage konsequent beantworten, ob er die Managed Services des Cloud-Providers nutzen oder die Infrastruktur selbst bauen und betreiben möchte. zanox hat sich dafür entschieden, die Managed Services des Cloud-Providers zu nutzen, da das Bauen und Verwalten eigener Infrastrukturbausteine keinerlei Business Value bringt. Reine Infrastruktur sind die EC2-Rechner aus dem Amazon-Portfolio. IAM hingegen bietet umfangreiche Sicherheitsmechanismen. In den deployten Services wird das AWS Java SDK genutzt, das es in Kombination mit IAM-Rollen für EC2 [1] erlaubt, Applikationen zu erstellen, die auf die Managed Services von AWS zugreifen können, ohne explizite Credentials zu nutzen. Einer EC2-Instanz wird beim initialen Bootstrapping eine IAM-Rolle mit den notwendigen Permissions zugewiesen. Über den Meta-Data-Service [2] werden dem AWS SDK die notwendigen Credentials übergeben. Die Applikation kann dann auf die Managed Services zugreifen, die in der Rolle definiert sind. Somit kann beispielsweise eine Applikation implementiert werden, die Metriken an das Monitoring-System Amazon CloudWatch und Events an die Data-Stream-Processing-Lösung Amazon Kinesis sendet, ohne explizite Credentials mit der Applikation ausrollen zu müssen.

Alle Anwendungen werden mit REST-Schnittstellen für Heartbeats und Healthchecks ausgestattet, damit sowohl die Applikation selbst als auch die für die korrekte Funktionalität notwendige Infrastruktur jederzeit geprüft werden können: Jede Applikation überprüft mit den Healthchecks die Infrastrukturkomponenten, die von ihr genutzt werden. Die Skalierung der Applikationen wird ausschließlich über Elastic Load Balancing (ELB) und AutoScaling [3] implementiert, damit eine feingranulare Skalierung der Applikationen über die konkrete Auslastung erreicht werden kann. AutoScaling startet bei Bedarf weitere EC2-Instanzen. ELB verteilt die Last zwischen den Instanzen. Der ELB-Service von AWS ist nicht nur für Webanwendungen geeignet, die mit dem HTTP-Protokoll arbeiten, sondern für alle Anwendungstypen. Ein Healthcheck kann auch auf Basis des TCP-Protokolls ohne HTTP implementiert werden, was noch einfacher als ein HTTP-Healthcheck ist.

Dennoch hat sich das Entwicklungsteam dafür entschieden, die ELB-Healthchecks über HTTP für alle Services zu implementieren, damit sich alle Services vollkommen gleich verhalten, unabhängig von der implementierten Logik, den genutzten Frameworks und der Sprache. Es kann durchaus sein, dass in Zukunft auch Applikationen in AWS deployt werden, die nicht auf der JVM laufen und

beispielsweise Go oder Python als Programmiersprachen nutzen. Für den ELB Healthcheck verwendet zanox die Heartbeat-URL der Applikation. Das heißt, erst wenn die EC2-Instanz mit der Applikation sauber gestartet wurde und der Heartbeat erfolgreich abgefragt werden kann, wird Traffic auf die Applikation geleitet bzw. werden eventuell notwendige Skalierungsoperationen der Infrastruktur vorgenommen.

Für das Monitoring der Applikationen bietet sich Amazon CloudWatch an, da mithilfe von CloudWatch-Alarmen Skalierungs-Events für die Auto Scaling Policies definiert werden können, d. h., die Infrastruktur auf Basis von Metriken automatisch skaliert. Dafür können beispielsweise EC2-Basismetriken wie CPU verwendet werden. Alternativ ist es aber auch möglich, nach CloudWatch eigene Metriken zu senden. Für diesen Zweck wird in diesem Projekt ein Fork des Projektes jmxtrans-agent [4] verwendet, das die Cloud-Watch-API nutzt, um JMX-Metriken in das Monitoring-System zu senden. JMX (Java Management Extension) ist der Standard für Monitoring und Metriken in der Java-Welt. Darüber hinaus werden aus der Applikation selbst (also aus der Geschäftslogik) Metriken mithilfe der Bibliothek Coda Hale Metrics [7] und einem Modul für die CloudWatch-Integration von Blacklocus [5] geschickt. Für das Logging wird ein etwas anderer Ansatz gewählt: In einer Cloud-Umgebung kann nie sichergestellt werden, ob nicht eine Server-Instanz plötzlich terminiert wird. Das führt normalerweise dazu, dass Daten, die üblicherweise persistent auf einem Server liegen, auf einmal verschwunden sein können. Log-Dateien sind ein Beispiel dafür. Aus diesem Grund läuft neben der eigentlichen Applikation auf dem Server noch zusätzlich der logstash-forwarder [6], um die Log-Einträge an unseren ELK-Service zu senden, der im eigenen Rechenzentrum läuft. Dieser Stack besteht aus Elasticsearch zur Speicherung, Logstash zum Parsen der Log-Daten und Kibana zur Analyse mithilfe einer UI. ELK ist ein Akronym für Elasticsearch, Logstash und Kibana. Darüber hinaus wird für jeden Request bzw. jedes Event in unserer HTTP-Schicht eine UUID errechnet, damit die Zuordnung von Log-Eintrag zu Event auch noch gemacht werden kann, wenn die EC2-Instanzen längst nicht mehr existieren.

Fazit

Das Pattern der Microservices-Architekturen passt sehr gut zum dynamischen Ansatz der Amazon Cloud, wenn die Architektur sauber entworfen und implementiert worden ist. Der klare Vorteil gegenüber der Implementierung im eigenen Rechenzentrum besteht in der Flexibilität der Infrastruktur. Dadurch kann eine nahezu beliebig skalierbare Architektur implementiert werden, die zudem sehr kosteneffizient ist.

- [1] <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html>
- [2] <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html>
- [3] <https://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/as-add-elb-healthcheck.html>
- [4] <https://github.com/SaschaMoellering/jmxtrans-agent>
- [5] <https://github.com/blacklocus/metrics-cloudwatch>

[6] <https://github.com/elastic/logstash-forwarder>

[7] <https://dropwizard.github.io/metrics/>

14.14 Fazit

Die im Beispiel verwendeten Technologien sind eine sehr gute Basis für die Umsetzung einer Microservices-Architektur, wenn sie mit Java arbeiten soll. Im Wesentlichen basiert das Beispiel auf dem Netflix-Stack, der sich schon seit Jahren in einer der größten Websites bewährt.

So zeigt das Beispiel das Zusammenspiel verschiedener Technologien für Service Discovery, Load Balancing und Resilience – wie auch einen Ansatz zum Testen der Microservices und für den Betrieb in Docker-Containern. Das Beispiel ist nicht darauf ausgelegt, direkt so in Produktion genutzt zu werden, sondern es soll vor allem sehr einfach zum Laufen gebracht werden können. Dazu sind einige Kompromisse notwendig. Aber so kann das Beispiel sehr gut als Basis für eigene Experimente und Ideen dienen.

Außerdem zeigt das Beispiel das Deployment der Anwendung mit Docker, das eine gute Basis für Microservices darstellt.

Wesentliche Punkte

- Spring, Spring Boot, Spring Cloud und der Netflix-Stack bieten gut integrierten Stack für die Entwicklung von Microservices mit Java. Die Technologien lösen alle typischen Herausforderungen von Microservices.
- Das Deployment mit Docker ist einfach realisierbar und mit Docker Machine und Docker Compose auch für die Cloud nutzbar.
- Testen von Microservices mit Consumer-Driven Contract Tests und Stubs ist auch ohne spezielle Werkzeuge möglich.

Selber ausprobieren und experimentieren

Für den Betrieb eines Microservice-Systems ist Log-Analyse aller Log-Dateien wichtig. Unter [17] steht ein Beispielprojekt für das Continuous-Delivery-Buch [18] zur Verfügung. Es enthält im Unterverzeichnis `log-analysis` ein Setup für die Log-Analyse mit dem ELK-Stack (Elasticsearch, Logstash und Kibana). Nutze diesen Ansatz, um auch das Microservice-Beispiel mit Log-Analyse zu versehen.

Genauso hat das Beispielprojekt für das Continuous-Delivery-Buch im Unterverzeichnis `graphite` eine Installation von Graphite für das Monitoring. Adaptiere diese für das Microservice-Beispiel.

Schreibe einen der Services in einer anderen Programmiersprache um. Nutze die Consumer-Driven Contract Tests (siehe [Abschnitt 14.13](#) und [11.7](#)), um die Implementierung abzusichern. Verwende ein Sidecar für die Integration in den Technologie-Stack (siehe [Abschnitt 14.12](#)).

14.15 Links & Literatur

- [1] <https://github.com/ewolff/microservice>
- [2] <http://projects.spring.io/spring-framework/>
- [3] <http://projects.spring.io/spring-data-rest/>
- [4] <http://projects.spring.io/spring-boot/>
- [5] <https://jaxenter.de/application-server-sind-tot-296>
- [6] <http://projects.spring.io/spring-cloud/>
- [7] <https://github.com/Netflix/zuul>
- [8] <https://github.com/Netflix/zuul/wiki/Writing-Filters>
- [9] <https://github.com/Netflix/zuul>
- [10] <https://github.com/Netflix/Hystrix/>
- [11] <https://github.com/Netflix/Hystrix/wiki/Configuration>
- [12] <https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib>
- [13] <https://github.com/Netflix/ribbon/wiki>
- [14] <http://github.com/Netflix/Prana/>
- [15] <http://projects.spring.io/spring-cloud/docs/1.0.0/spring-cloud.html>
- [16] <http://maven.apache.org/>
- [17] <https://github.com/ewolff/user-registration>
- [18] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, dpunkt.verlag, 2014, ISBN 978-3864902086
- [19] <https://maven.apache.org/download.cgi>
- [20] <https://docs.docker.com/machine/>
- [21] <http://docs.docker.com/compose/>

15 Technologien für Nanoservices

Der Begriff Microservice ist nicht einheitlich definiert. Einige verstehen unter Microservices extrem kleine Services – also eher zehn bis hundert Zeilen Code (LoC). Dieses Buch nennt den Ansatz Nanoservices. Die Abgrenzung von Microservices gegenüber Nanoservices steht im Mittelpunkt dieses Kapitels. Wesentliche Voraussetzung für das Umsetzen kleiner Services ist eine geeignete Technologie. Wenn die Technologie beispielsweise mehrere Services zu einem Betriebssystemprozess zusammenfasst, kann das den Ressourcenverbrauch pro Service und auch das Ausrollen der Services in Produktion vereinfachen. Dadurch sinkt der Aufwand pro Service. So können kleine Nanoservices und eine große Anzahl Nanoservices unterstützt werden.

[Abschnitt 15.1](#) diskutiert die Vorteile von Nanoservices und gibt so eine Motivation, warum Nanoservices sinnvoll sein können. [Abschnitt 15.2](#) definiert Nanoservices und grenzt sie von Microservices ab. In [Abschnitt 15.3](#) steht Amazon Lambda im Mittelpunkt: eine auf JavaScript oder Java basierende Cloud-Technologie, bei der keine Maschinen oder Application Server angemietet werden, sondern jeder Funktionsaufruf abgerechnet wird. OSGi ([Abschnitt 15.4](#)) dient zur Modularisierung von Java-Anwendungen und hat auch einen Service-Begriff. Eine andere Java-Technologie für Nanoservices ist Java EE ([Abschnitt 15.5](#)), wenn man sie richtig nutzt. Vert.x ([Abschnitt 15.6](#)) läuft auch auf der JVM, aber unterstützt neben Java eine breite Vielfalt an Programmiersprachen. Mit Erlang steht in [Abschnitt 15.7](#) eine Programmiersprache im Mittelpunkt, die es schon lange gibt und oft zur Umsetzung sehr anspruchsvoller Anwendungen dient. Die Architektur von Erlang erlaubt die Umsetzung von Nanoservices. Seneca ([Abschnitt 15.8](#)) hat einen ähnlichen Ansatz wie Erlang, basiert aber auf JavaScript und ist speziell für die Entwicklung von Nanoservices entwickelt.

15.1 Warum Nanoservices?

Nanoservices sind kein Widerspruch zu den bereits eingeführten Maßen für die Größe von Microservices: Sie liegen unter der in [Abschnitt 4.1](#) definierten maximalen Größe, die von der Größe des Teams abhängt und davon, wie groß ein Modul maximal sein kann, um es noch verstehen zu können. Durch geeignete Technologien können die technischen Grenzen für die minimale Größe eines Microservice aus [Abschnitt 4.1](#) weiter reduziert werden.

Sehr kleine Module sind leichter zu verstehen und damit leichter zu warten und weiterzuentwickeln. Auch können kleinere Microservices einfacher durch neue Implementierungen ersetzt werden. Also ist ein System, das aus möglichst kleinen Microservices besteht, einfacher weiterentwickelbar.

Es gibt Systeme, die mit sehr kleinen Microservices erfolgreich sind. In der Praxis sind zu groß gewählte Module meistens die Quelle für Probleme, die einer erfolgreichen Weiterentwicklung des Systems im Wege stehen. Jede Funktionalität könnte in einem eigenen Microservice implementiert werden – jede Klasse oder Funktion kann ein eigener

Microservice werden. [Abschnitt 10.2](#) hat gezeigt, dass es bei CQRS sinnvoll sein kann, einen Microservice zu implementieren, der nur Daten eines bestimmten Typs liest. Das Schreiben derselben Daten kann schon in einem anderen Microservice implementiert sein.

Warum also nicht winzig kleine Microservices? [Abschnitt 4.1](#) hat Faktoren identifiziert, die zu kleinen Microservices nicht praktikabel machen:

- Der Aufwand für die Infrastruktur wächst. Wenn jeder Microservice ein eigener Prozess ist und Infrastruktur wie einen Application Server und eine Integration in das Monitoring benötigt, ist der Aufwand für Hunderte oder gar Tausende von Microservices viel zu groß. Deswegen benötigen Nanoservices einen technologischen Ansatz, bei dem der Aufwand für die Infrastruktur für einen einzelnen Service möglichst gering ist. Ebenso ist ein niedriger Ressourcenverbrauch wünschenswert. Jeder einzelne Service soll möglichst wenig Speicher und CPU nutzen.
- Bei sehr kleinen Services fällt viel Kommunikation über das Netzwerk an, was die Performance des Systems negativ beeinflusst. Für Nanoservices sollte die Kommunikation zwischen den Services nicht über das Netzwerk stattfinden. Das kann dazu führen, dass die Wahlfreiheit bei den Technologien eingeschränkt wird. Wenn die Nanoservices in einem einzigen Prozess laufen, müssen sie meistens dieselbe Technologie verwenden. Ebenso beeinflusst ein solcher Ansatz die Robustheit des Systems. Wenn mehrere Services im selben Prozess laufen, ist es viel schwieriger, sie gegeneinander zu isolieren. Ein Nanoservice kann so viele Ressourcen verbrauchen, dass andere Nanoservices nicht mehr fehlerfrei funktionieren. Wenn beide Nanoservices im selben Prozess laufen, kann das Betriebssystem in solchen Situationen nicht mehr eingreifen. Ebenso kann der Absturz eines Nanoservice andere Nanoservices mit sich reißen.

Die technischen Kompromisse können Eigenschaften der Nanoservices negativ beeinflussen. Auf jeden Fall muss die wesentliche Eigenschaft von Microservices erhalten bleiben – nämlich das unabhängige Deployment der einzelnen Services.

Letztendlich geht es darum, Technologien zu identifizieren, mit denen der Overhead für einen einzelnen Nanoservice möglichst gering ist und gleichzeitig möglichst viele Vorteile der Microservices erhalten bleiben.

Konkret sind folgende Eigenschaften interessant:

- Der Aufwand für die Infrastruktur muss gering sein. Das umfasst Monitoring und Deployment. Ein neuer Nanoservice muss möglichst ohne Aufwand in Produktion gebracht werden können und dann auch gleich im Monitoring angezeigt werden.
- Der Ressourcenverbrauch beispielsweise in Bezug auf Speicher soll möglichst gering sein. So kann auch mit wenig Hardware eine große Menge an Nanoservices unterstützt werden. Das verbilligt nicht nur die Produktionsumgebung, sondern erleichtert auch den Aufbau von Testumgebungen.
- Kommunikation sollte ohne Netzwerk möglich sein. Das kommt nicht nur der Latenz und der Performance zugute, sondern erhöht auch die Zuverlässigkeit der Kommunikation zwischen den Nanoservices.
- Die Nanoservices können bei der Isolation gegeneinander Kompromisse eingehen.

Sie sollten gegeneinander isoliert sein, sodass ein Nanoservice nicht die anderen Services zum Ausfallen bringen kann. Sonst besteht die Gefahr, dass ein Nanoservice das gesamte System ausfallen lassen kann. Eine perfekte Isolation kann aber weniger wichtig sein als der geringere Aufwand für die Infrastruktur, der geringere Ressourcenverbrauch und die anderen Vorteile von Nanoservices.

- Nanoservices können die Wahl der Programmiersprachen und Frameworks einschränken. Microservices hingegen erlauben eine freie Wahl der Technologie.

Nanoservices ermöglichen die Nutzung von Microservice-Ansätzen in Bereichen, in denen Microservices kaum nutzbar sind. So ist es beispielsweise denkbar, eine Desktop-Anwendung in Nanoservices aufzuteilen. OSGi ([Abschnitt 15.4](#)) wird beispielsweise für Desktop und sogar für embedded Anwendungen genutzt. Eine Desktop-Anwendung, die aus Microservices besteht, ist hingegen wahrscheinlich zu schwierig zu deployen, um sie als Desktop-Anwendung auszuliefern.

15.2 Definition Nanoservice

Ein Nanoservice unterscheidet sich von einem Microservice dadurch, dass er in bestimmten Bereichen gegenüber einem Microservice Kompromisse eingeht – beispielsweise in Bezug auf die Isolation, weil mehrere Nanoservices auf einer virtuellen Maschine oder in einem Prozess laufen, oder in Bezug auf die Technologiefreiheit, weil die Nanoservices eine gemeinsame Plattform oder Programmiersprache erzwingen. Diese Einschränkungen ermöglichen kleinere Services. Die Infrastruktur kann so effizient sein, dass eine viel größere Anzahl an Services möglich ist, sodass der einzelne Service kleiner sein kann. Ein Nanoservice kann nur einige wenige Zeilen Code groß sein.

Auf keinen Fall darf die Technologie ein gemeinsames Deployment von Nanoservices erzwingen, denn das unabhängige Deployment ist die zentrale Eigenschaft von Microservices und Nanoservices. Erst das unabhängige Deployment ermöglichen die wesentlichen Vorteile von Microservices: die unabhängige Arbeit in den Teams oder die starke Modularisierung und in der Folge die nachhaltige Entwicklung.

Es ergibt sich also folgende Definition:

- Nanoservices gehen bei einigen Eigenschaften von Microservices *Kompromisse* ein. Das kann die Isolation oder die Technologiefreiheit sein – auf jeden Fall müssen die Nanoservices aber unabhängig deploybar sein.
- Die Kompromisse erlauben es, eine *größere Anzahl* Services und damit *kleinere Services* zu unterstützen. Nanoservices können nur einige wenige Programmzeilen umfassen.
- Dazu nutzen die Nanoservices sehr *effiziente Ablaufumgebungen*. Sie machen sich die Einschränkungen der Nanoservices zunutze, um so mehr und kleinere Services zu ermöglichen.

Nanoservices hängen also sehr stark von der verwendeten Technologie ab. Die Technologie erlaubt bestimmte Kompromisse in den Nanoservices und damit eine bestimmte Größe eines Nanoservice. Daher orientiert sich dieses Kapitel an verschiedenen

Technologien, um die möglichen Spielarten der Nanoservices genauer zu erläutern.

Ziel der Nanoservices ist es, die Vorteile von Microservices zu verstärken. Noch kleinere Deployment-Einheiten senken das Risiko von Deployments weiter, vereinfachen sie noch mehr und erreichen eine noch bessere Verständlichkeit und Austauschbarkeit von einzelnen Nanoservices. Auch die fachliche Architektur ändert sich: Ein BOUNDED CONTEXT, der aus einem oder einigen wenigen Microservices bestehen kann, wird aus einer Vielzahl Nanoservices bestehen, die jeweils sehr eng begrenzte Funktionalitäten implementieren.

Der Übergang zwischen Microservices und Nanoservices ist fließend: Werden zwei Microservices in derselben virtuellen Maschine deployt, erhöht das schon die Effizienz und ist gleichzeitig ein Kompromiss bei der Isolation. Beide Microservices teilen sich nun eine Betriebssystem-Instanz und eine virtuelle Maschine. Wenn ein Microservice die Ressourcen der virtuellen Maschine aufbraucht, fallen die anderen Microservices auf der virtuellen Maschine auch aus. Also ist die Isolation der Microservices schlechter.

Übrigens ist der Begriff »Nanoservice« nicht sehr gebräuchlich. Dieses Buch verwendet ihn, um klarzumachen, dass es Modularisierungen gibt, die Microservices ähnlich sind, aber sich im Detail unterscheiden und noch kleinere Services ermöglichen. Um diese Technologien mit ihren Kompromissen klar von »echten« Microservices abzugrenzen, ist der Begriff »Nanoservice« hilfreich.

15.3 Amazon Lambda

Amazon Lambda [1] ist ein Service in der Amazon Cloud. Er ist weltweit in allen Amazon-Rechenzentren verfügbar – auch in Europa und Deutschland.

Amazon Lambda kann einzelne Funktionen ausführen, die in JavaScript geschrieben sind und auf dem JavaScript Framework Node.js oder Java 8 mit OpenJDK basieren. Der Code dieser Funktionen hat keine Abhängigkeiten zu Amazon Lambda. Zugriffe auf das Betriebssystem sind möglich. Auf den Rechnern sind das Amazon Web Services SDK sowie ImageMagick für Bildmanipulationen vorhanden. Diese Funktionalitäten können Amazon-Lambda-Anwendungen ausnutzen. Ebenso können weitere Node.js-Bibliotheken nachinstalliert werden.

Amazon-Lambda-Funktionen müssen recht schnell starten, denn es kann passieren, dass sie für jeden Request einzeln gestartet werden. Deswegen dürfen die Funktionen auch keinen Zustand halten.

Dementsprechend fallen keine Kosten an, wenn keine Requests an die Funktion gestellt werden. Sonst wird jeder Aufruf abgerechnet, wobei die erste Million Aufrufe kostenlos ist und eine weitere Million aktuell 0,20 \$ kostet.

Die Lambda-Funktionen können direkt mit einem [Lambda-Funktionen aufrufen](#) Kommandozeilentool aufgerufen werden. Die Verarbeitung erfolgt asynchron. Die Funktionen können über verschiedene Amazon-Funktionalitäten Ergebnisse zurückliefern. Dazu enthält die Amazon-Cloud Messaging-Lösungen wie SNS (Simple Notification Service) oder SQS (Simple Queuing Service).

Folgende Ereignisse können einen Aufruf einer Lambda-Funktion bewirken:

- In S3 (Simple Storage Service) können große Dateien abgelegt und heruntergeladen werden. Solche Aktionen lösen Ereignisse aus, auf die eine Amazon-Lambda-Funktion reagieren kann.
- Mit Amazon Kinesis können Datenströme verwaltet und verteilt werden. Diese Technologie ist auf die Echtzeitverarbeitung großer Datenmengen ausgelegt. Lambda-Funktionen können als Reaktion auf neue Daten in diesen Strömen aufgerufen werden.
- DynamoDB ist eine Datenbank in der Amazon Cloud. Sie kann bei Änderungen an dem Datenbestand der Datenbank Lambda-Funktionen aufrufen, sodass Lambda-Funktionen praktisch Datenbank-Trigger werden.

Lambda erlaubt ohne Probleme das unabhängige Deployment verschiedener Funktionen. Sie können auch jeweils ihre eigenen Bibliotheken mitbringen.

Bewertung für Nanoservices

Der Aufwand für die Infrastruktur ist bei dieser Technologie minimal: Eine neue Version einer Amazon-Lambda-Funktion kann mit einem Kommandozeilenwerkzeug einfach deployt werden. Auch die Überwachung ist einfach, weil die Funktionen sofort in Cloud Watch integriert sind. Cloud Watch bietet Amazon an, um Metriken aus Cloud-Anwendungen zu erheben und Log-Dateien zu konsolidieren und zu überwachen. Ebenso können auf Basis dieser Daten Alarne definiert werden, die dann per SMS oder E-Mail weitergeleitet werden können. Da alle Amazon Services mit einer API angesprochen werden können, ist es möglich, das Monitoring oder Deployment zu automatisieren und in eigene Infrastrukturen zu integrieren.

Amazon Lambda erzwingt eine Kommunikation über die verschiedenen Amazon-Services. Es ist nicht ohne Weiteres möglich, eine Amazon-Lambda-Funktion über REST anzusprechen, da nur S3, Kinesis und DynamoDB Amazon-Lambda-Funktionen aufrufen können. Ebenso erzwingt Amazon Lambda, dass Node.js oder Java genutzt wird. Das schränkt die Technologiefreiheit entscheidend ein.

Aber Amazon Lambda bietet eine hervorragende Isolation der Funktionen. Das ist auch notwendig. Schließlich steht die Plattform verschiedenen Nutzern zur Verfügung. Es ist nicht akzeptabel, wenn die Lambda-Funktion eines Nutzers die Lambda Funktionen der anderen Nutzer negativ beeinflusst.

Mit Amazon Lambda können extrem kleine Services umgesetzt werden. Der Overhead für einen einzelnen Service ist sehr gering. Ein unabhängiges Deployment ist ebenfalls möglich. Als kleinste Deployment-Einheit kann eine JavaScript- oder Java-Funktion dienen – kleiner geht es kaum. Und selbst für eine Vielzahl an Java- oder JavaScript-Funktionen ist der Aufwand für das Deployment recht gering.

Fazit

Amazon Lambda ist nur ein Teil des Amazon-Ökosystems. Daher kann es um Technologien wie Amazon Elastic Beanstalk ergänzt werden. Dort können Microservices laufen, die größer sein und in anderen Sprachen geschrieben sein können. Ebenso ist eine Kombination mit EC2 (Elastic Computing Cloud) möglich. EC2 bietet virtuelle

Maschinen, auf denen beliebige Software installiert werden kann. Und es gibt eine breite Auswahl an Datenbanken und anderen Services, die mit geringem Aufwand genutzt werden können. Amazon Lambda definiert sich als eine Ergänzung dieses Cloud-Baukastens. Schließlich ist einer der entscheidenden Vorteile der Amazon Cloud, dass bereits nahezu jede erdenkliche Infrastruktur vorhanden ist. Entwickler können sich so auf die Entwicklung spezieller Funktionalitäten konzentrieren, während die meisten Standardkomponenten nur angemietet werden müssen.

Selber ausprobieren und experimentieren

Unter [8] findet sich ein sehr ausführliches Tutorial, das den Umgang mit Amazon Lambda praktisch verdeutlicht. Es bleibt nicht bei den einfachen Szenarien stehen, sondern zeigt auch die Nutzung komplexer Mechanismen wie die Verwendung verschiedener Node.js-Bibliotheken oder das Reagieren auf verschiedene Events in dem Amazon-System. Amazon bietet Neukunden kostenlose Kontingente der meisten Services an, aber bei Lambda hat jeder Kunde ein so umfangreiches Freikontingent, dass es für Tests und ein näheres Kennenlernen der Technologie auf jeden Fall ausreichend ist.

15.4 OSGi

OSGi [3] ist ein Standard, von dem es verschiedene Implementierungen gibt [4]. Eingebettete Systeme nutzen oft OSGi. Auch die Entwicklungsumgebung Eclipse basiert auf OSGi und viele Java-Desktop-Anwendungen nutzen das Eclipse-Framework. Der OSGi-Standard definiert eine Modularisierung innerhalb der JVM (Java Virtual Machine). Java kennt zwar eine Aufteilung des Codes in Klassen, aber es gibt kein Modulkonzept für größere Einheiten.

OSGi ergänzt Java um ein solches Modulsystem. Dazu führt OSGi Bundles in die Java-Welt ein. Bundles bauen auf Javas JAR-Dateien auf, die den Code zu mehreren Klassen enthalten. Bundles haben einige zusätzliche Einträge in der Datei META-INF/MANIFEST.MF, die jede JAR-Dateien enthalten sollte. Mit diesen Einträgen kann geregelt werden, welche Klassen und Interfaces das Bundle exportiert. Andere Bundles können diese Klassen und Interfaces importieren. So ergänzt OSGi Java um ein recht ausgefeiltes Modulkonzept, ohne völlig neue Konzepte zu erfinden.

Das OSGi-Modulsystem

Listing 15–1 Ein OSGi MANIFEST.MF

```
Bundle-Name: Ein Service
Bundle-SymbolicName: com.ewolff.service
Bundle-Description: Ein kleiner Service
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: com.ewolff.service.Activator
Export-Package: com.ewolff.service.interfaces;version="1.0.0"
Import-Package: com.ewolff.otherservice.interfaces;version="1.3.0"
```

Listing 15–1 zeigt ein Beispiel. Neben Beschreibungen und Namen des Bundles wird der

Bundle Activator definiert. Diese Java-Klasse wird beim Start des Bundles ausgeführt und kann das Bundle initialisieren. Schließlich gibt Export-Package an, welche Packages von diesem Bundle bereitgestellt werden. Alle Klassen und Interfaces aus diesen Packages stehen anderen Bundles zur Verfügung. Import-Package dient dazu, Packages von einem anderen Bundle zu importieren. Die Packages können auch versioniert werden.

Neben Interfaces und Klassen können Bundles auch Services exportieren. Dazu reicht kein Eintrag in MANIFEST.MF, sondern es muss Code geschrieben werden. Services sind letztendlich nur Java-Objekte. Andere Bundles können die Services importieren und nutzen. Auch die Nutzung der Services geschieht im Code.

Bundles können zur Laufzeit installiert, gestartet, gestoppt und deinstalliert werden. Wenn ein Bundles Klassen oder Interfaces exportiert und ein anderes Bundle diese nutzt, ist ein Update nicht mehr ganz so einfach. Alle Bundles, die Klassen oder Interfaces des alten Bundles nutzen und nun das neu installierte Bundles nutzen wollen, müssen neu gestartet werden.

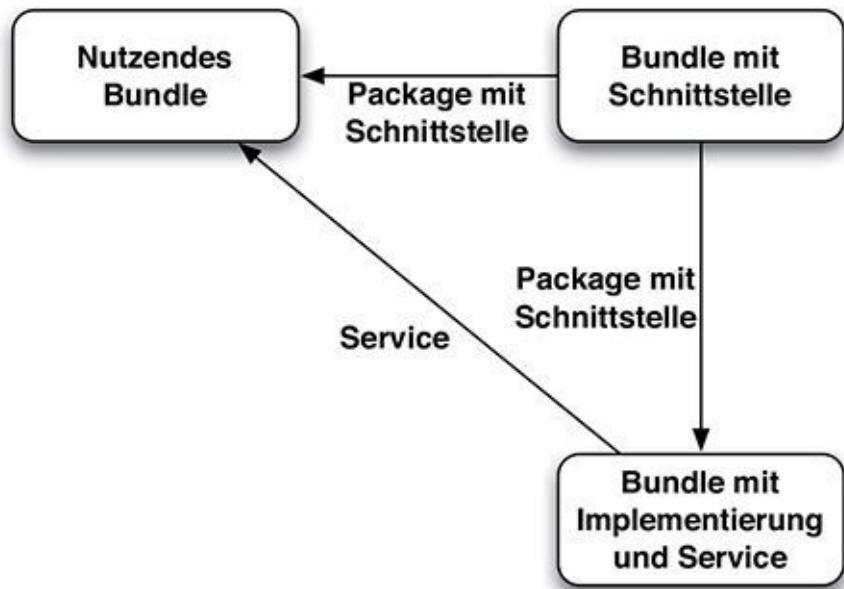
Die gemeinsame Nutzung von Code ist für den Microservice-Ansatz bei Weitem nicht so wichtig wie das Nutzen von Services. Dennoch muss mindestens die Schnittstelle der Services auch anderen Bundles angeboten werden.

In der Praxis hat sich ein Vorgehen etabliert, bei dem ein Bundle nur die Schnittstellen des Service als Klassen und Interfaces exportiert. Ein anderes Bundle enthält die Implementierung des Service. Die Klassen der Implementierung werden nicht exportiert. Die Service-Implementierung wird als OSGi Service exportiert. Zur Nutzung des Service muss ein Bundle die Schnittstelle aus dem einen Bundle importieren und den Service aus dem anderen Bundle (siehe Abb. 15–1).

OSGi erlaubt es, dass Services neu gestartet werden. Mit dem gezeigten Vorgehen kann die Implementierung des Service ausgetauscht werden, ohne dass andere Bundles neu gestartet werden müssen. Diese Bundles verwenden nur die Interfaces und Klassen der Schnittstelle, die sich nicht ändert, sodass ein Neustart nicht mehr zwingend ist. Der Zugriff auf Services kann so implementiert werden, dass die neue Version des Service auch tatsächlich genutzt wird.

Mit den OSGi Blueprints [5] oder OSGi Declarative Services [6] können die Details beim Umgang mit dem OSGi-Service-Modell abstrahiert werden. Das vereinfacht den Umgang mit OSGi. Beispielsweise machen diese Technologien es viel einfacher, mit dem Neustart eines Service oder dessen vorübergehendem Ausfall bei einem Neustart umzugehen.

Abb. 15–1 OSGi-Service, Implementierung und Schnittstelle



Ein unabhängiges Deployment von Services ist möglich, aber auch aufwendig, da Schnittstelle und Service in unterschiedliche Bundles aufgeteilt werden müssen. Dieses Modell erlaubt nur Änderungen an der Implementierung. Änderungen an der Schnittstelle sind komplexer in der Umsetzung. Dann müssen die benutzenden Bundles neu gestartet werden, weil sie die Schnittstelle neu laden müssen.

In der Realität werden aus diesen Gründen OSGi-Systeme oft komplett neu installiert, statt einzelne Bundles zu ändern. Beispielsweise zieht ein Eclipse-Update oft einen Neustart nach sich. Eine komplette Neuinstallation erleichtert auch die Reproduktion der Umgebung. Wenn ein OSGi-System dynamisch geändert wird, ist es irgendwann in einem Zustand, den keiner mehr wiederherstellen kann. Das Ändern einzelner Bundles ist aber wesentliche Voraussetzung, um den Nanoservice-Gedanken mit OSGi umzusetzen. Unabhängiges Deployment ist die wesentliche Eigenschaft eines Nanoservice.

OSGi hat auf Java-Projekte bezüglich der Architektur eine positive Auswirkung. Die Bundles sind meistens recht klein, sodass sie einzeln gut verständlich sind. Außerdem zwingt die Aufteilung in Bundles Entwickler und Architekten dazu, über die Beziehungen zwischen den Bundles nachzudenken und diese in den Konfigurationen der Bundles zu hinterlegen. Andere Abhängigkeiten sind im System nicht möglich. So ergibt sich meistens eine sehr saubere Architektur mit klaren und gewollten Abhängigkeiten.

Bewertung für Nanoservices

Allerdings bietet OSGi keine Technologiefreiheit: Es basiert auf der JVM und kann daher nur mit Java oder JVM-basierten Sprachen genutzt werden. Es ist zum Beispiel so gut wie ausgeschlossen, dass ein OSGi-Bundle eine eigene Datenbank mitbringt, weil Datenbanken meistens nicht in Java geschrieben sind. Für solche Fälle müssten eigene Lösungen neben der OSGi-Infrastruktur gefunden werden.

Bei einigen Java-Technologien ist eine Integration mit OSGi schwierig, weil das Laden der Java-Klassen ohne OSGi anders funktioniert. Und viele populäre Java-Application-Server unterstützen OSGi für eigene Anwendungen nicht, sodass in solchen Umgebungen das Ändern von Code zur Laufzeit nur unzureichend unterstützt wird. Die Infrastruktur muss speziell auf OSGi angepasst werden.

Auch sind die Bundles nicht optimal isoliert: Wenn ein Bundle sehr viel CPU nutzt oder

die JVM zum Absturz bringt, beeinträchtigt das die anderen Bundles in derselben JVM. Abstürze sind z. B. durch ein Speicherleck möglich, bei dem wegen eines Fehlers immer mehr Speicher allokiert wird, bis das System zusammenbricht. Solche Fehler können leicht durch ein Versehen entstehen.

Dafür können die Bundles dank OSGi lokal miteinander kommunizieren. Verteilte Kommunikation ist mit verschiedenen Protokollen auch möglich. Und die Bundles teilen sich eine JVM, was beispielsweise den Speicherverbrauch senkt.

Lösungen zum Monitoring sind ebenfalls bei den verschiedenen OSGi-Implementierungen vorhanden.

OSGi hat vor allem Einschränkungen bei der Technologiefreiheit. OSGi begrenzt das Projekt auf Java-Technologien. Das unabhängige Deployment der Bundles ist in der Praxis nur schwer umsetzbar. Insbesondere Schnittstellenänderungen werden nur schlecht unterstützt. Außerdem sind Bundles nicht besonders gut gegeneinander isoliert. Dafür können die Bundles sehr einfach mit lokalen Aufrufen miteinander interagieren.

Selber ausprobieren und experimentieren

- Mach dich mit OSGi beispielsweise mit dem Tutorial [7] vertraut.
- Erstelle für einen Ausschnitt eines dir bekannten Systems ein Konzept zur Aufteilung in Bundles und Services.
- Wenn du das System mit OSGi umsetzen würdest: Welche zusätzlichen Technologien (Datenbanken etc.) müsstest du unterbringen? Wie würdest du damit umgehen?

15.5 Java EE

Java EE [18] ist ein Standard aus dem Java-Bereich. Er umfasst verschiedene APIs. Dazu gehören beispielsweise JSF (Java ServerFaces), Servlet und JSP (Java Server Pages) für Webanwendungen, JPA (Java Persistence API) für Persistenz oder JTA für Transaktionen. Außerdem definiert Java EE ein Deployment-Modell. Webanwendungen können in WAR-Dateien (Web ARchive) verpackt werden, JAR-Dateien (Java ARchive) können Logikkomponenten wie Enterprise Java Beans (EJBs) enthalten und schließlich können EARs (Enterprise ARchives) eine Sammlung von JARs und WARs enthalten. Alle diese Komponenten werden in einen Application Server deployt. Der Application Server implementiert die Java-EE-APIs und bietet beispielsweise Unterstützung für HTTP, die dafür notwendige Verwaltung von Threads und Netzwerkverbindungen und schließlich auch Unterstützung für den Zugriff auf Datenbanken.

Dieser Abschnitt beschäftigt sich mit WARs und dem Deployment-Modell von Java EE Application Servern. Kapitel 14 hat bereits ausführlich ein Java-System gezeigt, das ohne Application Server auskommt. Stattdessen startet es auf der Java Virtual Machine (JVM) direkt eine Java-Anwendung. Die Anwendung ist in einer JAR-Datei eingepackt und enthält die vollständige Infrastruktur. Dieses Deployment heißt auch Fat-JAR-

Deployment, weil in einem JAR die Anwendung einschließlich der vollständigen Infrastruktur enthalten ist. Das Beispiel aus [Kapitel 14](#) nutzt Spring Boot, das auch einige Java-EE-APIs wie JAX-RS für REST unterstützt. Dropwizard [19] bietet ein solches JAR-Modell auch an. Es ist eigentlich für JAX RS basierte REST-Webservices gedacht, kann aber auch andere Anwendungen unterstützen. Wildfly Swarm [20] ist eine Variante des Java-EE-Servers Wildfly, die auch ein solches Deployment-Modell unterstützt.

Ein Fat-JAR-Deployment verbraucht zu viele Ressourcen für Nanoservices. Aber in einem Java-EE-Application-Server können mehrere WARs deployt und so Ressourcen eingespart werden. Auf jedes WAR kann unter einer eigenen URL zugegriffen werden. Und jedes WAR kann einzeln deployt werden. So ist es möglich, jeden Nanoservice einzeln in Produktion zu bringen.

Die Trennung zwischen WARs ist aber nicht perfekt:

- Speicher und CPU nutzen alle Nanoservices gemeinsam. Wenn ein Nanoservice viel CPU oder Speicher nutzt, kann das die anderen Nanoservices behindern. Der Absturz eines Nanoservice reißt alle anderen Nanoservices mit sich.
- Das erneute Deployment eines WAR führt in der Praxis zu Speicherlecks, wenn nicht die komplette Anwendung aus dem Speicher entfernt werden kann. Daher ist das unabhängige Deployment einzelner Nanoservices in der Praxis nicht so ohne Weiteres möglich.
- Im Gegensatz zu OSGi sind die ClassLoader der WARs vollständig getrennt. Es gibt keine Möglichkeit, auf den Code der anderen Nanoservices zuzugreifen.
- Wegen der Trennung des Codes können WARs nur über HTTP oder REST miteinander kommunizieren. Lokale Methodenaufrufe sind nicht möglich.

Da sich mehrere Nanoservices einen Application Server und eine JVM teilen, ist diese Lösung effizienter als ein Fat JAR Deployment jedes Microservice in einer eigenen JVM wie in [Kapitel 14](#). Die Nanoservices nutzen einen gemeinsamen Heap und verbrauchen so weniger Speicher. Aber die Skalierung funktioniert nur, indem mehr Application Server gestartet werden. Jeder der Application Server enthält alle Nanoservices. Alle Nanoservices müssen gemeinsam skaliert werden. Das Skalieren einzelner Nanoservices ist nicht möglich.

Die Technologiewahl ist auf JVM-Technologien begrenzt. Außerdem sind alle Technologien ausgeschlossen, die nicht mit dem Servlet-Modell arbeiten. Das schließt beispielsweise Vert.x ([Abschnitt 15.6](#)) oder Play aus.

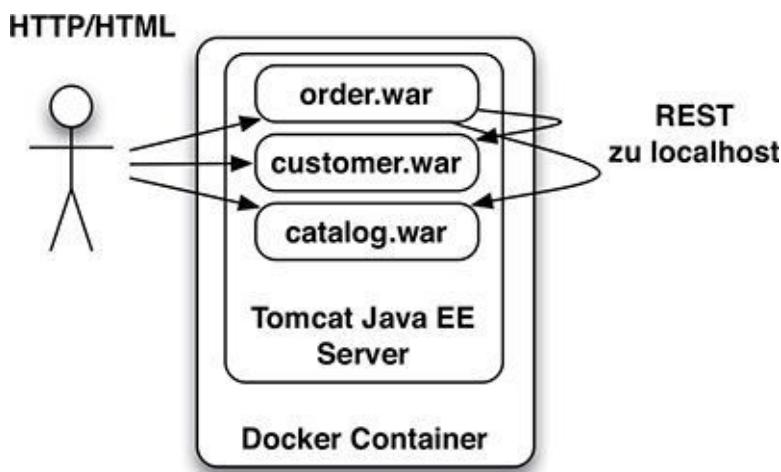
Für Microservices kann Java EE auch eine Option sein: *Microservices mit Java EE?* Theoretisch wäre es denkbar, jeden Microservice mit einem eigenen Application Server auszustatten. Gegenüber dem Fat-JAR-Deployment muss dann neben der Anwendung auch ein Application Server installiert und konfiguriert werden. Die Version des Application Servers muss zu der Version der Anwendung passen und die Konfiguration ebenfalls. Dieser zusätzlichen Komplexität steht kein Vorteil gegenüber. Da das Deployment und das Monitoring der Application Server nur für Java-Anwendungen funktionieren, können diese Features in einer Microservices-Architektur

nur genutzt werden, wenn die Technologiefreiheit auf Java-Technologien begrenzt wird. Generell gilt, dass Application Server kaum einen Vorteil bieten – gerade für Microservices [21].

Die Anwendung aus [Kapitel 14](#) gibt es auch mit dem Java-EE-Deployment-Modell [22]. [Abbildung 15–2](#) zeigt einen Überblick über das Beispiel: Es gibt drei WARs, die jeweils Order, Customer und Catalog enthalten. Sie kommunizieren untereinander durch REST. Wenn Customer ausfällt, würde Order auf dem Host auch ausfallen. Order kommuniziert nämlich nur mit dieser einen Customer-Instanz. Für eine bessere Verfügbarkeit müsste der Zugriff auf andere Customer-Instanzen umgeleitet werden.

Ein Kunde kann die UI der Nanoservices von außen mit HTML/ HTTP nutzen. Der Code ist gegenüber der Lösung aus [Kapitel 14](#) nur geringfügig modifiziert. Die Netflix-Bibliotheken sind entfernt worden. Dafür ist die Anwendung um eine Unterstützung von Servlet-Containern ergänzt worden.

Abb. 15–2 Beispielanwendung mit Java-EE-Microservices



Selber ausprobieren und experimentieren

Unter [22] findet sich die Anwendung als Java-EE-Nanoservices.

Die Anwendung nutzt die Netflix-Technologien nicht. Konkret:

- Hystrix würde Resilience bieten (siehe [Abschnitt 14.10](#)). Ist es sinnvoll, Hystrix in die Anwendung zu integrieren? Wie sind die Nanoservices gegeneinander isoliert? Hilft Hystrix immer? Siehe auch [Abschnitt 10.5](#) zu Stabilität und Resilience. Wie können diese Patterns in dieser Anwendung realisiert werden?
- Eureka ist bei Service Discovery hilfreich. Wie würde es in die Java-EE-Microservices passen?
- Wie könnten andere Services-Discovery-Technologien integriert werden (siehe [Abschnitt 8.9](#))?
- Ribbon für Load Balancing zwischen REST-Services wäre ebenfalls integrierbar. Welche Vorteile würde das bringen? Wäre es auch ohne Eureka möglich, Ribbon zu nutzen?

15.6 Vert.x

Vert.x [23] ist ein Framework mit zahlreichen interessanten Ansätzen. Es läuft zwar auf der JVM (Java Virtual Machine), aber unterstützt viele unterschiedliche Programmiersprachen – u. a. Java, Scala, Clojure, Groovy, Ceylon –, aber auch JavaScript, Ruby oder Python. Ein Vert.x-System ist aus Verticles aufgebaut. Sie empfangen Events und können Nachrichten zurückschicken.

[Listing 15–2](#) zeigt ein einfaches Vert.x Verticle, das lediglich die eingehenden Nachrichten zurückschickt. Der Code erzeugt einen Server. Wenn sich ein Client mit dem Server verbindet, wird ein Callback aufgerufen und der Server erzeugt eine Pump. Die Pump dient dazu, Daten von einer Quelle zu einem Ziel zu übertragen. Im Beispiel sind Ziel und Quelle identisch.

Die Anwendung wird nur aktiv, wenn ein Client sich verbindet und der Callback aufgerufen wird. Auch die Pump wird nur aktiv, wenn neue Daten vorliegen. Solche Events werden vom Event Loop verarbeitet, der die Verticles aufruft. Die Verticles müssen die Events dann verarbeiten. Ein Event Loop ist ein Thread. Üblicherweise wird pro CPU-Kern ein Event Loop gestartet, sodass die Event Loops parallel bearbeitet werden. Ein Event Loop und damit ein Thread bzw. ein CPU-Kern können beliebig viele Netzwerkverbindungen verwalten. Events von allen Verbindungen können in einem einzigen Event Loop bearbeitet werden. Daher ist Vert.x auch für Anwendungen geeignet, die sehr viele Verbindungen handhaben müssen.

[Listing 15–2](#) Einfaches Java-Vert.x-Echo-Verticle

```
public class EchoServer extends Verticle {

    public void start() {
        vertx.createNetServer().connectHandler(
            new Handler<NetSocket>() {
                public void handle(final NetSocket socket) {
                    Pump.createPump(socket, socket).start();
                }
            }).listen(1234);
    }
}
```

Wie erwähnt unterstützt Vert.x verschiedene Programmiersprachen. [Listing 15–3](#) zeigt denselben Echo-Verticle in JavaScript. Der Code entspricht den JavaScript-Konventionen und nutzt beispielsweise eine JavaScript-Funktion für das Callback. Vert.x hat einen Layer, mit dem die Basisfunktionalität so adaptiert wird, dass sie wie eine native Bibliothek für die jeweilige Programmiersprache wirkt.

[Listing 15–3](#) Einfaches JavaScript-Vert.x-Echo-Verticle

```
var vertx = require('vertx')

vertx.createNetServer().connectHandler(function(sock) {
    new vertx.Pump(sock, sock).start();
}).listen(1234);
```

Vert.x Modules können mehrere Verticles in verschiedenen Sprachen enthalten. Verticles und Modules können untereinander mit einem Event-Bus kommunizieren. Die

Nachrichten auf dem Event-Bus nutzen JSON als Datenformat. Der Event-Bus kann auf mehrere Server verteilt werden. So unterstützt Vert.x Verteilung und kann Hochverfügbarkeit umsetzen, indem Module auf anderen Servern gestartet werden. Außerdem sind die Verticles und Modules lose gekoppelt, da sie nur Nachrichten austauschen. Vert.x bietet auch eine Unterstützung für andere Messaging-Systeme und kann auch mit HTTP und REST kommunizieren. Eine Integration von Vert.x-Systemen in Microservice-Systeme ist recht einfach möglich.

Module können einzeln deployt und auch wieder entfernt werden. Da die Module durch Events miteinander kommunizieren, ist es ohne Weiteres möglich, dass Module zur Laufzeit durch neue Module ersetzt werden. Sie müssen nur dieselben Nachrichten verarbeiten. Ein Modul kann einen Nanoservice implementieren. Module können in neuen Knoten gestartet werden, sodass der Ausfall einer JVM kompensiert werden kann.

Vert.x unterstützt auch Fat JARs, bei dem die Anwendung alle benötigten Bibliotheken mitbringt. Das ist für Microservices nützlich, weil die Anwendung so alle Abhängigkeiten mitbringt. Für Nanoservices ist dieser Ansatz nicht besonders nützlich.

Vert.x kann durch das unabhängige Deployment der Modules und die lose Kopplung durch den Event-Bus mehreren Nanoservices eine Heimat innerhalb einer JVM bieten. Allerdings würde ein Absturz der JVM, ein Speicherleck oder ein Blockieren des Event Loops alle Modules und Verticles in der JVM beeinflussen. Dafür unterstützt Vert.x viele unterschiedliche Programmiersprachen – trotz der Beschränkung auf die JVM. Das ist nicht nur eine theoretische Möglichkeit, sondern Vert.x hat als Ziel, in allen unterstützten Sprachen möglichst natürlich nutzbar zu sein. Vert.x setzt voraus, dass die gesamte Anwendung nichtblockierend geschrieben ist. Es gibt allerdings die Möglichkeit, blockierende Aufgaben in Worker Verticles auszuführen. Sie nutzen getrennte Thread Pools, sodass sie die nichtblockierenden Verticles nicht beeinflussen.

Fazit

Selber ausprobieren und experimentieren

Einen Einstieg in die Entwicklung mit Vert.x bietet direkt die Homepage [23]. Sie zeigt, wie ein Webserver mit verschiedenen Programmiersprachen implementiert und ausgeführt werden kann. Die Entwicklung von Modulen zeigt am Beispiel Java und Maven [24]. Komplexe Beispiele in verschiedenen Programmiersprachen finden sich unter [25].

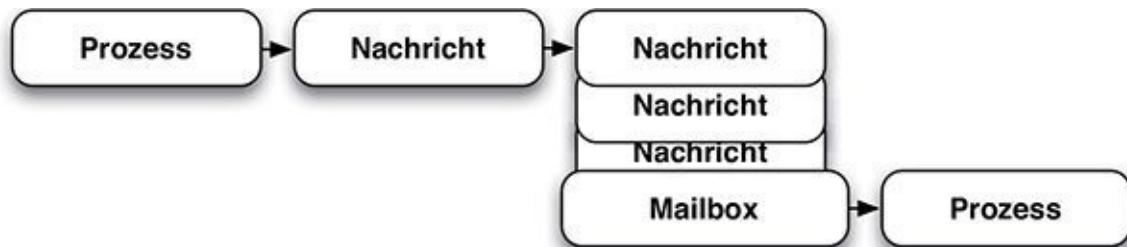
15.7 Erlang

Erlang [9] ist eine funktionale Programmiersprache, die vor allem zusammen mit dem OTP-Framework (Open Telecom Platform) genutzt wird. Ursprünglich ist Erlang für den Bereich der Telekommunikation entwickelt worden. Anwendungen in diesem Bereich müssen sehr zuverlässig sein. Mittlerweile wird Erlang in allem Bereichen genutzt, in denen seine Stärken besonders vorteilhaft sind. Erlang nutzt als Ablaufumgebung ähnlich wie Java eine virtuelle Maschine. Sie heißt BEAM (Bogdan/Björn's Erlang Abstract Machine).

Erlangs Stärken sind vor allem die Ausfallsicherheit und die Möglichkeit, Systeme jahrelang laufen zu lassen. Das ist nur möglich durch dynamische Updates der Software. Gleichzeitig hat Erlang ein leichtgewichtiges Parallelitätskonzept. Dazu nutzt Erlang das Konzept der Prozesse. Sie haben nichts mit Betriebssystemprozessen zu tun und sind noch leichtgewichtiger als Betriebssystem-Threads. In einem Erlang-System können Millionen von Prozessen laufen, die gegeneinander isoliert sind.

Ein weiterer Faktor für die Isolation ist die asynchrone Kommunikation. Die Prozesse eines Erlang-Systems kommunizieren über Nachrichten miteinander. Nachrichten landen in der Mailbox eines Prozesses. In einem Prozess wird immer nur eine Nachricht verarbeitet. Dadurch ist der Umgang mit Parallelität sehr einfach: Zwischen den Prozessen gibt es parallele Nachrichten, in einem Prozess wird immer nur eine Nachricht bearbeitet. Zu dem Modell passt auch der funktionale Ansatz der Sprache, der möglichst ohne Zustand auskommt. Dieser Ansatz entspricht den Verticles in Vert.x und ihrer Kommunikation über den Event-Bus.

Abb. 15–3 Kommunikation zwischen Erlang-Prozessen



[Listing 15–4](#) zeigt einen einfachen Erlang-Server, der die empfangene Nachricht zurückschickt. Er ist in einem eigenen Modul definiert. Das Modul exportiert die Funktion `loop`, die keine Parameter hat. Diese Funktion empfängt eine Nachricht `Msg` von einem Knoten `From` und schickt dann an diesen Knoten dieselbe Nachricht zurück. Zum Verschicken der Nachricht dient der Operator `»!«`. Danach wird die Funktion wieder aufgerufen und wartet auf die nächste Nachricht. Genau derselbe Code kann auch genutzt werden, um von einem anderen Rechner über das Netzwerk aus aufgerufen zu werden. Lokale Nachrichten und Nachrichten über das Netzwerk werden mit denselben Mechanismen verarbeitet.

Listing 15–4 Ein Erlang-Echo-Server

```

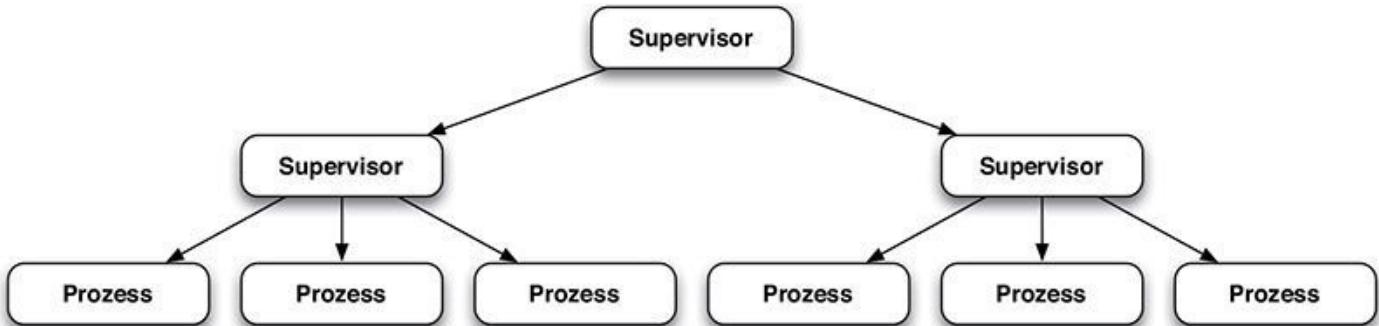
-module(server).
-export([loop/0]).

loop() ->
    receive
        {From, Msg} ->
            From ! Msg,
            loop()
    end.
  
```

Durch das Verschicken von Nachrichten sind Erlang-Systeme besonders robust. Erlang setzt auf »Let It Crash«. Ein einzelner Prozess kann einfach neu gestartet werden, wenn es ein Problem gibt. Dafür ist der Supervisor zuständig: ein Prozess, der nur dazu da ist, andere Prozesse zu überwachen und gegebenenfalls neu zu starten. Auch der Supervisor wird überwacht und bei einem Problem neu gestartet. Dadurch entsteht in Erlang ein Baum, durch den letztendlich das System auf den Ausfall jedes Prozesses vorbereitet ist

(siehe Abb. 15–4).

Abb. 15–4 Überwachung in Erlang



Weil das Erlang-Prozess-Modell so leichtgewichtig ist, ist der Neustart eines Prozesses sehr schnell erledigt. Wenn der Zustand in anderen Komponenten ausgelagert ist, gehen auch keine Informationen verloren. Der Rest des Systems ist von dem Absturz des Prozesses nicht beeinflusst: Da die Kommunikation asynchron ist, können die anderen Prozesse mit der höheren Latenz wegen des Neustarts umgehen. In der Praxis hat sich dieser Ansatz sehr gut bewährt. Erlang-Systeme sind sehr robust und dennoch einfach zu entwickeln.

Dieser Ansatz basiert auf dem Actor-Modell [10]: Actors kommunizieren über asynchrone Nachrichten miteinander. Als Ergebnis können sie selber Nachrichten schicken, neue Actors starten oder ihr Verhalten für die nächsten Nachrichten ändern. Erlangs Prozesse entsprechen den Actors.

Ebenso gibt es einfache Möglichkeiten, Erlang-Systeme zu überwachen. Erlang selber hat dazu Funktionen eingebaut, die Speicherauslastung oder den Zustand der Mailboxen überwachen können. OTP bietet dazu den Operations and Maintenance Support an (OAM), der beispielsweise auch in SNMP-Systeme integriert werden kann.

Da Erlang typische Problem bei der Implementierung von Microservices wie Ausfallsicherheit gut löst, bietet es sich für die Implementierung von Microservices an [11]. Ein Microservice wäre dann ein in Erlang geschriebenes System, das intern aus mehreren Prozessen besteht.

Aber die Services können auch kleiner werden: Jeder Prozess in einem Erlang-System könnte als Nanoservice aufgefasst werden. Er kann unabhängig von den anderen deployt werden, und das auch im laufenden Betrieb. Darüber hinaus ist Erlang dazu in der Lage, auch Betriebssystemprozesse zu integrieren. Sie werden dann auch in die Supervisor-Hierarchie eingebaut und bei einem Absturz gegebenenfalls neu gestartet.

Für Erlang kann also ein einzelner Prozess als Bewertung für Nanoservices Nanoservice verstanden werden. Der Aufwand für die Infrastruktur ist dann recht gering: Monitoring ist mit Erlang-Bordmitteln möglich, ebenso das Deployment. Da die Prozesse sich eine BEAM-Instanz teilen, ist der Overhead für einen einzelnen Prozess nicht besonders hoch. Ebenso ist es möglich, dass die Prozesse Nachrichten austauschen, ohne dass sie dabei über das Netzwerk kommunizieren müssen. Eine Isolation der Prozesse ist ebenfalls umgesetzt.

Schließlich können sogar Systeme in anderen Sprachen integriert werden. Dazu wird

ein Betriebssystemprozess, der in einer beliebigen Sprache implementiert sein kann, unter die Kontrolle von Erlang gestellt. Der Betriebssystemprozess kann zum Beispiel durch »Let It Crash« abgesichert werden. Dadurch sind praktisch alle anderen Technologien in Erlang integrierbar – selbst wenn sie in einem eigenen Prozess laufen.

Auf der anderen Seite ist Erlang nicht besonders weit verbreitet. Der konsequent funktionale Ansatz ist auch gewöhnungsbedürftig. Und schließlich ist die Syntax von Erlang für viele Entwickler nicht besonders intuitiv.

Selber ausprobieren und experimentieren

Ein einfaches Beispiel ist das Projekt unter [17]. Es basiert auf dem Code aus diesem Abschnitt und zeigt, wie die Kommunikation zwischen Knoten möglich ist. [12] ist ein sehr schönes Tutorial für Erlang, das auch auf Deployment und Betrieb eingehet. Mit den Informationen aus dem Tutorial kann das Beispiel aus [12] um Supervisor ergänzt werden. Als alternative Sprache aus dem Erlang-Ökosystem gibt es Elixir [13]. Es hat eine andere Syntax, profitiert aber auch von den Konzepten von OTP. Elixir ist wesentlich einfacher zu lernen als Erlang und bietet sich für einen Einsteig an. Schließlich gibt es viele andere Implementierungen des Actor Models [10]. Hier lohnt sich ein näherer Blick, um zu entscheiden, ob solche Implementierungen ebenfalls für die Umsetzung von Microservices oder Nanoservices sinnvoll nutzbar sind und welche Vorteile dabei entstehen. Interessant könnte beispielsweise Akka aus dem Scala-/Java-Bereich sein.

15.8 Seneca

Seneca [2] basiert auf Node.js und nutzt damit JavaScript auf dem Server. Node.js hat ein Programmiermodell, bei dem ein Prozess viele Aufgaben parallel abarbeiten kann. Um das zu ermöglichen, gibt es einen Event-Loop, in dem die Events verarbeitet werden. Wenn von einer Netzwerkverbindung aus eine Nachricht in das System kommt, wartet es zunächst, bis der Event Loop frei wird. Dann verarbeitet der Event Loop die Nachricht. Die Verarbeitung muss schnell gehen, weil er sonst blockiert wird und alle anderen Nachrichten lange auf eine Bearbeitung warten werden. Im Event Loop darf daher beispielsweise auf keinen Fall auf die Antwort anderer Server gewartet werden – das würde das System zu lange blockieren. Die Interaktion mit anderen Systemen muss so umgesetzt werden, dass die Interaktion nur angestoßen wird. Wenn die Antwort des anderen Systems ankommt, wird sie durch den Event Loop bearbeitet. Dann ruft der Event Loop ein Callback auf, das beim Anstoßen der Interaktion registriert worden ist. Dieses Modell ähnelt dem Ansatz von Vert.x und Erlang.

Seneca führt in Node.js einen Mechanismus ein, mit dem Commands verarbeitet werden können. Es werden Patterns von Commands definiert, bei denen ein bestimmter Code ausgeführt werden soll.

Die Kommunikation mit solchen Commands ist auch über das Netzwerk sehr einfach möglich. Listing 15–5 zeigt einen Server, der `seneca.add()` aufruft. Damit werden ein neues Pattern und Code zur Behandlung eines Events mit diesem Pattern definiert. Auf das

Command mit dem Bestandteil cmd: "echo" reagiert eine Funktion, die aus dem Command den Wert für value ausliest. Das nutzt die Funktion als Parameter für den Aufruf des Callbacks. Mit seneca.listen() wird nun der Server gestartet, der auf Commands aus dem Netzwerk horcht.

Listing 15–5 Seneca-Server

```
var seneca = require("seneca")()

seneca.add( {cmd: "echo"}, function(args,callback){
  callback(null,{value:args.value})
})

seneca.listen()
```

Der Client aus [Listing 15–6](#) schickt wegen seneca.client() alle Commands, die lokal nicht bearbeiten können, über das Netzwerk an den Server. Mit seneca.act() wird ein Command erzeugt. Es enthält cmd: "echo" – deswegen wird die Funktion des Servers aufgerufen. Als value wird "echo this" genutzt. Der Server gibt diesen String an die übergebene Funktion zurück – und so wird er schließlich an der Konsole ausgegeben. Der Beispielcode findet sich unter [\[16\]](#).

Listing 15–6 Seneca-Client

```
var seneca=require("seneca")()

seneca.client()

seneca.act('cmd: "echo",value:"echo this", function(err,result){
  console.log( result.value )
})
```

Es ist also sehr einfach, ein verteiltes System mit Seneca umzusetzen. Allerdings nutzen die Services kein Standardprotokoll wie REST für die Kommunikation. Aber auch REST-Systeme können mit Seneca umgesetzt werden. Außerdem basiert das Seneca-Protokoll auf JSON und kann daher auch von anderen Sprachen aus genutzt werden.

Ein Nanoservice kann eine Funktion sein, die mit Seneca auf Aufrufe aus dem Netzwerk reagiert – und damit sehr klein. Wie bereits dargestellt, ist ein Node.js-System, wie es mit Seneca umgesetzt wird, anfällig, wenn eine Funktion den Event-Loop zum Erliegen bringen kann. Also ist die Isolation nicht besonders gut.

Für das Monitoring der Seneca-Anwendung gibt es eine Admin-Konsole, die zumindest ein einfaches Monitoring anbietet. Sie ist aber jeweils nur für einen Node-Prozess verfügbar. Ein Monitoring über alle Knoten hinweg muss auf anderem Wege erfolgen.

Ein unabhängiges Deployment einer einzelnen Seneca-Funktion ist nur dann möglich, wenn es für die Seneca-Funktion einen einzelnen Node-Prozess gibt. Das schränkt das unabhängige Deployment sehr ein, denn der Aufwand eines Node-Prozesses ist kaum für eine einzige Funktion akzeptabel. Ebenso ist es nicht einfach möglich, andere Technologien in ein Seneca-System zu integrieren. Letztendlich muss das gesamte Seneca-System in JavaScript umgesetzt werden.

Seneca ist speziell für die Umsetzung von Microservices mit JavaScript entwickelt worden. Es

Bewertung für Nanoservices

erlaubt tatsächlich eine sehr einfache Implementierung von Services, die auch über das Netzwerk angesprochen werden können. Die grundlegende Architektur ist Erlang nicht unähnlich: Bei beiden Ansätzen schicken sich Services Messages bzw. Commands, auf die Funktionen reagieren. Bezuglich des unabhängigen Deployments einzelner Services, der Isolation der Services gegeneinander und der Integration anderer Technologien ist Erlang jedoch deutlich überlegen. Außerdem hat Erlang eine wesentlich längere Historie und wird schon lange in verschiedenen, auch sehr anspruchsvollen Anwendungen verwendet.

Selber ausprobieren und experimentieren

Der erste Schritt, sich mit Seneca vertraut zu machen, kann das Code-Beispiel aus [16] sein. Ein grundlegendes Tutorial für Seneca findet sich unter [14]. Ebenfalls lohnt sich ein Blick auf die Beispiele [15]. Das Nanoservice-Beispiel kann sicher zu einer umfangreicheren Anwendung ausgebaut oder auf mehr Node-Prozesse verteilt werden.

15.9 Fazit

Die Technologien in diesem Kapitel zeigen, wie Microservices auch ganz anders implementiert werden können. Weil der Unterschied so groß ist, erscheint der Begriff »Nanoservice« gerechtfertigt. Die Nanoservices sind dann nicht mehr zwangsläufig ein eigener Prozess, der nur über das Netzwerk angesprochen werden kann, sondern laufen alle gemeinsam in einem Prozess und sprechen mit lokalen Kommunikationsmechanismen miteinander. Dadurch erlauben Nanoservices nicht nur viel kleinere Services, sondern auch den Einsatz von Microservice-Ansätzen in Bereichen wie embedded oder Desktop-Anwendungen.

Einen Überblick über die Vor- und Nachteile der Technologien bietet Tabelle 15–1. Besonders interessant ist in diesem Zusammenhang sicher Erlang, weil es auch die Integration anderer Technologien ermöglicht und die einzelnen Nanoservices gut gegeneinander isolieren kann, sodass ein Problem in einem Nanoservice die anderen Services nicht zum Ausfall bringt. Dabei ist Erlang schon lange die Basis sehr vieler wichtiger Systeme, sodass die Technologie über jeglichen Zweifel erhaben ist.

Seneca hat einen ähnlichen Ansatz, kann aber bei der Isolation und der Integration anderer Technologien als JavaScript nicht mithalten. Vert.x hat einen ähnlichen Ansatz auf der JVM, unterstützt sehr viele Sprachen, geht aber bei der Isolation nicht ganz so weit wie Erlang. Java EE erlaubt keine Kommunikation ohne Netzwerk. Das getrennte Deployment in Java EE ist schwierig. In der Praxis kommt es beim Deployment von WARs immer wieder zu Speicherlecks. OSGi erlaubt im Gegensatz zu Java EE die gemeinsame Nutzung von Code zwischen Nanoservices. OSGi nutzt außerdem für die Kommunikation der Services Methoden-Aufrufe und keine Commands bzw. Messages, wie Erlang und Seneca das tun. Command oder Messages haben den Vorteil, dass sie etwas flexibler sind. Bestandteile, die von einem Service nicht genutzt werden können, sind kein Problem – sie können einfach ignoriert werden.

Tab. 15–1 Bewertung der Technologien bezüglich der Eignung für Nanoservices

	Amazon Lambda	OSGi	Java EE	Vert.x	Erlang	Seneca
Aufwand Infrastruktur pro Service	++	+	+	+	++	++
Ressourcenverbrauch	++	++	++	++	++	++
Kommunikation ohne Netzwerk	-	++	—	+	++	-
Isolation der Services gegeneinander	++	—	—	-	++	-
Nutzung verschiedener Technologien	-	—	—	+	+	—

Amazon Lambda ist besonders interessant, weil es in den Amazon-Technologiebaukasten integriert ist. Das macht den Umgang mit der Infrastruktur sehr einfach. Die Infrastruktur kann bei sehr kleinen Nanoservices ein Problem sein, weil eben viel mehr Infrastrukturen für die vielen Services angeboten werden müssen. Bei Amazon ist ein Datenbankserver aber nur ein API-Aufruf oder ein Klick weg – oder man nutzt statt eines Servers gleich eine API, um die Daten zu speichern. Server werden unsichtbar – und das ist auch bei Amazon Lambda so. Es gibt keine Infrastruktur für einen einzelnen Service, sondern nur Code, der ausgeführt wird und andere Services nutzen kann. Wegen der vorbereiteten Infrastruktur ist auch das Monitoring keine echte Herausforderung.

Wichtigste Punkte

- Nanoservices unterteilen Systeme in noch kleinere Services. Dafür gehen sie Kompromisse z. B. bei Technologiefreiheit oder Isolation ein.
- Für Nanoservices sind effiziente Infrastrukturen notwendig, die mit mehr und kleinen Nanoservices einfach zureckkommen.

15.10 Links und Literatur

- [1] <http://aws.amazon.com/lambda>
- [2] <http://senecajs.org/>
- [3] <http://www.osgi.org/>
- [4] http://en.wikipedia.org/wiki/OSGi#Current_framework_implementations
- [5] <http://wiki.osgi.org/wiki/Blueprint>

- [6] http://wiki.osgi.org/wiki/Declarative_Services
- [7] <http://www.vogella.com/tutorials/OSGi/article.html>
- [8] <http://aws.amazon.com/de/lambda/getting-started/>
- [9] <http://www.erlang.org/>
- [10] http://en.wikipedia.org/wiki/Actor_model
- [11] <https://www.innoq.com/en/talks/2015/01/talk-microservices-erlang-otp/>
- [12] <http://learnyousomeerlang.com/>
- [13] <http://elixir-lang.org/>
- [14] <http://senecajs.org/getting-started.html>
- [15] <https://github.com/rjroger/seneca-examples/>
- [16] <https://github.com/ewolff/seneca-example/>
- [17] <https://github.com/ewolff/erlang-example/>
- [18] <http://www.oracle.com/technetwork/java/javaee/overview/index.html>
- [19] <https://dropwizard.github.io/dropwizard/>
- [20] <http://github.com/wildfly-swarm/>
- [21] <http://jaxenter.com/java-application-servers-dead-1-111928.html>
- [22] <https://github.com/ewolff/javaee-example/>
- [23] <http://vertx.io/>
- [24] http://vertx.io/maven_dev.html
- [25] <https://github.com/vert-x/vertx-examples>

16 Wie mit Microservices loslegen?

Dieses Kapitel stellt zum Abschluss des Buchs dar, wie der Start mit Microservices aussehen kann. [Abschnitt 16.1](#) zeigt die verschiedenen Vorteile von Microservices noch einmal auf, um zu verdeutlichen, dass es nicht nur einen einzelnen Grund für Microservices gibt. [Abschnitt 16.2](#) beschreibt verschiedene Wege zum Einführen von Microservices – abhängig von dem Einsatzkontext und den erwarteten Vorteilen. Der [Abschnitt 16.3](#) geht schließlich der Frage nach, ob Microservices mehr als nur ein Hype sind.

16.1 Warum Microservices?

Microservices haben eine Vielzahl von Vorteilen (siehe auch [Kapitel 5](#)). Dazu gehören beispielsweise:

- Agilität ist durch Microservices für große Projekte einfacher umsetzbar, weil Teams unabhängig arbeiten können.
- Microservices können dabei helfen, Legacy-Anwendungen zu ergänzen und abzulösen.
- Eine nachhaltige Entwicklung ist möglich, da Microservices-Architekturen weniger anfällig für Architekturverfall sind und einzelne Microservices abgelöst werden können. Dadurch kann die Wartbarkeit langfristig hoch gehalten werden.
- Darüber hinaus gibt es weitere technische Gründe wie Stabilität und Skalierbarkeit.

Diese Vorteile und die Vorteile aus [Kapitel 5](#) zu priorisieren, sollte ein erster Schritt zur Adaption einer Microservices-Architektur sein. Genauso müssen die Herausforderungen aus [Kapitel 6](#) bewertet und gegebenenfalls Problemlösungsstrategien für die Herausforderungen erarbeitet werden.

Eine herausragende Stellung nehmen dabei Continuous Delivery und Infrastruktur ein. Wenn die Prozesse noch manuell sind, ist der Aufwand für den Betrieb einer größeren Anzahl Microservices so hoch, dass die Einführung von Microservices kaum machbar ist. Leider haben viele Organisationen gerade in den Bereichen Continuous Delivery und Infrastruktur noch erhebliche Schwächen. Dann sollte zusammen mit der Einführung von Microservices auch Continuous Delivery eingeführt werden. Da Microservices viel kleiner als Deployment-Monolithen sind, ist Continuous Delivery mit Microservices einfacher. So können sich beide Vorstöße gegenseitig befürworten.

Ebenso muss die organisatorische Ebene ([Kap. 13](#)) betrachtet werden. Wenn die Skalierung agiler Prozessoren ein wichtiger Grund für Microservices ist, sollten die agilen Prozesse auch funktionieren. Beispielsweise muss es einen Product Owner pro Team geben, der auch über Fachlichkeiten entscheidet, sowie eine agile Planung. Die Teams sollten auch schon weitgehend selbstständig sein – sonst nutzen sie am Ende die mögliche Unabhängigkeit der Microservices nicht aus.

Der Einstieg in Microservices kann mehr als nur ein Problem lösen. Die Motivation für Microservices unterscheidet sich bei jedem Projekt. Die große Anzahl an Vorteilen alleine kann auch ein guter Grund für Microservices sein. Die Strategie zur Einführung von Microservices muss auf die erwarteten Vorteile abgestimmt sein.

16.2 Wege zu Microservices

Auf dem Weg zu Microservices gibt es verschiedene Ansätze:

- Typisch ist der Start mit einem Monolithen, der schrittweise zu einer Menge von Microservices umgebaut wird. Ein treibender Grund ist oft das einfachere Deployment, aber auch die unabhängige Skalierung und die klarere Architektur können Gründe sein. Meistens werden dann einzelne Microservices schrittweise aus dem Monolithen herausgebrochen.
- Die Migration von einem Monolithen hin zu Microservices kann aber auch ganz anders ablaufen. Wenn beispielsweise Resilience der Grund für die Migration ist, können zunächst dafür Technologien wie Hystrix in den Monolithen eingebaut werden. Danach kann die Aufteilung in Microservices erfolgen.
- Viel weniger häufig ist der Start auf der grünen Wiese. In der Situation kann das Projekt mit einem Monolithen beginnen. Aber sinnvoller ist eine erste grobe fachliche Aufteilung, aus der sich erste Microservices ergeben. Dadurch wird eine Infrastruktur geschaffen, die mehr als einen Microservice unterstützt. Teams können so auch unabhängig an Features arbeiten. Eine Aufteilung in feingranulare Microservices ist hingegen kaum sinnvoll, weil sie vermutlich revidiert werden muss und die Änderungen dazu sehr komplex sind.

Microservices sind sehr gut mit vorhandenen Systemen kombinierbar, was die Einführung vereinfacht. Ein kleiner Microservice als Ergänzung zu einem Deployment-Monolithen ist schnell geschrieben. Wenn sich große Probleme ergeben, kann der Microservice auch recht schnell wieder aus dem System entfernt werden. Schrittweise können dann andere technische Elemente eingeführt werden.

Die einfache Kombination von Microservices mit Legacy-Systemen ist ein wesentlicher Grund dafür, dass der Einstieg so einfach ist und auch gleich Vorteile bringen kann.

16.3 Microservice: Hype oder Realität?

Microservices sind ohne Zweifel ein Ansatz, der gerade im Mittelpunkt des Interesses steht. Das ist an sich nicht schlecht – aber oft sind solche Ansätze auf den zweiten Blick doch nur ein Modetrend, der keine echten Probleme löst.

Das Interesse an Microservices ist mehr als nur ein solcher Modetrend:

- Wie schon in der Einleitung beschrieben, setzt Amazon auf ein solches Modell seit vielen Jahren. Viele Internet-Firmen setzen ebenfalls auf diesen Ansatz und auch schon seit langer Zeit. Microservices sind also nicht einfach ein neuer Trend, sondern werden schon lange im Verborgenen genutzt.

- Die Microservice-Pioniere haben sehr viel Geld in Infrastruktur investiert, die als Open Source zur Verfügung steht – Netflix ist ein solches Beispiel. Diese Unternehmen nutzen Microservices schon lange. Die Vorteile sind für sie so groß, dass sie dazu eigene Infrastrukturen geschaffen haben, während heutzutage diese Technologien als kostenloses Open Source zur Verfügung stehen. So ist der Einstieg mittlerweile viel einfacher.
- Der Trend zu Agilität und Cloud-Infrastrukturen findet in Microservices eine passende Architektur: Sie ermöglicht die Skalierung von Agilität und entspricht den Anforderungen der Cloud an robuste und skalierbare Architekturen.
- Ebenso unterstützen Microservices als kleine Deployment-Einheiten Continuous Delivery, auf das viele setzen, um die Qualität der Software zu erhöhen und Software schneller in Produktion zu bringen.
- Es gibt mehr als einen Grund für Microservices. Also sind Microservices ein Fortschritt in verschiedenen Bereichen. So hängt die Nutzung nicht von einem einzigen Grund ab, sondern ist viel stabiler.

Jeder hat vermutlich schon große, unübersichtliche Systeme gesehen. Vielleicht ist es einfach an der Zeit, kleinere Systeme zu entwickeln und die damit einhergehenden Vorteile zu realisieren. Gute Gründe für Monolithen gibt es jedenfalls scheinbar wenige – außer der niedrigeren technischen Komplexität.

16.4 Fazit

Der Schritt hin zu Microservices erscheint also aus verschiedenen Gründen sinnvoll:

- Es gibt eine Vielzahl von Vorteilen (siehe [Abschnitt 16.1](#) und [Kap. 5](#)).
- Der Weg zu Microservices ist evolutionär. Es ist nicht nötig, komplett neu zu starten. Im Gegenteil: Eine schrittweise Migration ist der übliche Ansatz ([Abschnitt 16.2](#)). Dabei können unterschiedliche Ansätze gewählt werden, um möglichst schnell von den Vorteilen zu profitieren.
- Der Start ist reversibel: Sind Microservices doch ungeeignet, baut man sie einfach zurück oder verfolgt den Weg nicht weiter.
- Microservices sind mehr als ein Hype ([Abschnitt 16.3](#)). Dazu sind sie zu lange in Benutzung und zu breit adaptiert.

Also sollte man mit Microservices mindestens experimentieren – und dazu lädt das Buch an vielen Stellen ein.

Selber ausprobieren und experimentieren

Beantworte folgende Fragen anhand einer dir bekannten Architektur/System:

- Welches sind die wichtigsten Vorteile von Microservices in dem Kontext?
- Wie könnte eine Migration zu Microservices aussehen? Mögliche Ansätze:
 - Neue Funktionalitäten im Microservice implementieren

- Bestimmte Eigenschaften (z. B. Robustheit oder schnelles Deployment) durch passende Technologien ermöglichen
- Wie könnte ein Projekt aussehen, das mit möglichst geringem Aufwand den Einstieg in Microservices ausprobiert?
 - Wann wäre das Projekt ein Erfolg und der Einstieg daher sinnvoll?

Index

A

Abhangigkeit 103
Acceptance Test Driven Design 220
ACID-Eigenschaften 35
Akka 214, 363
Akzeptanztests 220
Amazon 22
Amazon Lambda 349
AMQP 186
AngularJS 168, 171
Antwortzeit 152
API Keys 158
Archaius 314
Architekturmanagement 106
Asset-Server 169
ATDD 220
Authentifizierung 154
Autorisierung 154

B

BDD 220
Behavior Driven Design 220
Betrieb 241, 288, 300
BIND 145
Blue/Green-Deployment 259
BMC Remedy 254
Bounded Context 44, 46, 52, 309
Bulkhead 208, 211

C

CA Opscenter 254
Canary Releasing 260
CAP-Theorem 142
CDC 234
Circuit Breaker 208, 211, 331
Client Library 75, 295
Cloud 264
CMS 134, 136
Codezeilen 31
collectd 255
Collective Code Ownership 278
Command Query Responsibility Segregation 199
Command Query Separation 199
Consul 145, 149
Consumer-driven Contract 234
Consumer-Driven Contract Test 338
Content Management System 134, 136
Content Negotiation 329
Context Map 109
Continuous Delivery 5, 63, 241
Continuous Deployment 259
Continuous-Delivery-Pipeline 63, 223
Conway 39, 67
CoreOS 267
CORS 170
CQRS 199
CQS 199
Cross Origin Resource Sharing 170

D

Datenbank 187

Datenreplikation 188
Datensparsamkeit 159
DDD 44, 102, 198, 202
Deployment 218
Deployment-Monolith 3
Developer Anarchy 288
DevOps 288, 300
DNS 144
Docker 264, 316
Docker Compose 268
Docker Machine 268
Dockerfile 316
Domain Name System 144
Domain-Driven Design 44, 102, 198, 202
Don't Reperat Yourself 116
Drogenmissbrauch 283
DRY 116
Durchsatz 152

E

EDA 137
Edge Side Includes 174
Elasticsearch 246
Skalieren 247
ELK-Stack 247
Ember.js 168
Endurance-Tests 221
Enterprise Integration Patterns 129
Erlang 360
ESI 174
etcd 267
Eureka 145, 314, 326

Event Sourcing 201
Event-driven Architecture 137
explorativer Test 221
Ext JS 168

F

Fachbereich 294
Fachliche Architektur 101
Feign 314
Firewall 159

G

Gesetz von Conway 39, 67
Grafana 253
Graphite 253
Graylog 247
Größe 31

H

HAL 181
HATEOAS 180
hexagonale Architektur 203
HP Operations Manager 254
HSQLDB 310
Hypermedia as the Engine of Application State 180
Hystrix 211, 314, 331
Hystrix-Dashboard 332

I

IBM Tivoli 254
Icinga 253
Immutable Server 142, 257

Integrationstests [220](#)

Intrusion Detection [159](#)

J

Java EE [355](#)

Java Messaging Service [186](#)

JavaScript Object Notation [181](#)

JMS [186](#)

JSON [181](#)

JSON Web Encryption [156](#)

JSON Web Signature [156](#)

JSON Web Token [156](#)

JWE [156](#)

JWS [156](#)

JWT [156](#)

K

Kafka [186](#)

Kapazitätstest [221](#)

Kapselung [198](#)

Kerberos [157](#)

Kibana [246](#)

Kohäsion [103, 198](#)

Kompensationstransaktion [36](#)

Konsistenz [36](#)

Kopplung [103, 197](#)

Kubernetes [267](#)

L

Lambda [349](#)

Last-Tests [221](#)

Legacy-Anwendung [127](#)

Legacy-System [4](#), [61](#)

Lines of Code [31](#)

LoC [31](#), [345](#)

Logstash [246](#)

M

Makro-Architektur [281](#)

Manuelle Test [220](#)

Mesos [267](#)

Messaging [183](#)

Metrics [255](#)

Microservice

 Definition [2](#)

 Gründe [3](#)

 Herausforderungen [6](#)

 und UI [54](#)

Mikro-Architektur [281](#)

Mobile Client [176](#)

Mocks [232](#)

Moco [233](#)

Modularisierung [32](#)

Monitoring [249](#)

Monolith [3](#)

mountebank [232](#)

N

Nachhaltige Software-Entwicklung [61](#)

Nagios [253](#)

Nanoservice [348](#)

Nanoservices [35](#), [345](#)

Netflix [313](#)

Node.js [349](#), [363](#)

O

OAuth2 [154](#), [170](#)
Open Source [296](#)
Operations Manager [254](#)
Opscenter [254](#)
Orchestrierung [85](#), [88](#), [212](#)
OSGi [352](#)
OTP [360](#)

P

PaaS [268](#)
Packetbeat [108](#), [253](#)
Pact [236](#)
Pacto [236](#)
Performance-Tests [221](#)
Platform as a Service [268](#)
Play [214](#)
Portal [85](#), [175](#)
Portlet [175](#)
Ports and Adapters [204](#)
Postels Gesetz [192](#)
Prana [336](#)
Product Owner [293](#), [300](#)
Protocol Buffer [182](#)
Pull Request [280](#)

R

RabbitMQ [186](#)
Reactive [211](#), [213](#)
Refactoring [33](#), [78](#), [112](#)

Remedy 254
Remote Procedure Call 182
Replikation 188
Representational State Transfer 179
Resilience 136, 207
REST 179
RESTful HTTP 180
Ribbon 149, 314, 335
Rich Client 176
Riemann 253
Robustheit 65
Robustheitsprinzip 192
Roll Forward (Deployment) 259
Rollback (Deployment) 259
Routing 329
RPC 182
RxJava 214
RxJS 214

S

Same Origin Policy 170
SAML 157
Scala 214
Schnittstelle 190
SCS 55
Self-Contained System 55
Semantic Versioning 192
Server Side Includes 175
Service Discovery 130, 143, 148, 326
Service-Oriented Architecture 83
Seyren 253
Sharding 152

Shared-Nothing-Ansatz 75

Sicherheit 153, 160

Sidecar 140, 336

Single-Page-App 168

Skalierbarkeit 150

Skalierung 64

dynamisch 151

SOA 83

SOAP 88, 182

SPA 168

Splunk 248

Spring Boot 311

Spring Cloud 312

Spring Cloud Netflix 313

Spring Data REST 311

SSI 175

SSL 158

Stabilität 207

StatsD 255

Steady State 209

Strategic Design 102

Structure 101 106

Stub 337

stubby4j 233

Stubs 232

T

Technologische Wahlfreiheit 66

Template 243

Testgetriebene Entwicklung 219

Testpyramide 225

Thrift 183

Timeout 207, 211

Tivoli 254

TLS 158

Transaktion 35–36, 185, 198

Transaktionen 35

Turbine 314, 333

U

UI 54

UI-Test 220

Unit-Test 219

UNIX-Philosophie 2

V

Vagrant 317

Vault 159

Vertikale 55

Vert.x 214, 358

VirtualBox 317

virtuelle Maschine 263

W

Whitebox-Reuse 114

Wiederverwendung 114, 295

WireMock 233

WS-Security 182

WS-* 89, 183

X

XML 181

Z

ZeroMQ [186](#)

Zertifikat [158](#)

Zipkin [249](#)

Zustandslosigkeit [213](#)

Zuul [314, 329](#)

Zyklische Abhangigkeiten [104](#)