Eberhard Wolff

Das Microservices-Praxisbuch

Grundlagen, Konzepte und Rezepte

Inklusive Link zur **kostenlosen** englischen E-Book-Version

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 Bücher geschrieben – u.a. über Continuous Delivery – und als Sprecher auf internationalen Konferenzen vorgetragen. Sein technologischer Schwerpunkt liegt auf modernen Architekturansätzen – Cloud, Continuous Delivery, DevOps, Microservices oder NoSQL spielen oft eine Rolle.

Eberhard Wolff

Das Microservices-Praxisbuch

Grundlagen, Konzepte und Rezepte

dpunkt.verlag

Eberhard Wolff
eberhard.wolff@gmail.com

Lektorat: René Schönfeldt
Projektmanagement: Miriam Metsch
Copy-Editing: Petra Kienle, Fürstenfeldbruck
Satz: III-satz, www.drei-satz.de
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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

ISBN:
Buch 978-3-86490-526-1
PDF 978-3-96088-461-3
ePub 978-3-96088-462-0
mobi 978-3-96088-463-7

1. Auflage 2018

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1

Inhaltsübersicht

Inhaltsverzeichnis

Anhang

Einleitung

Microservices sind einer der wichtigsten Software-Architektur-Trends, grundlegende Werke über Microservices gibt es schon. Unter anderem auch das Microservices-Buch (*http://microservices-buch.de*)[1] vom Autor dieses Werks. Warum noch ein weiteres Buch über Microservices?

Es ist eine Sache, eine Architektur zu definieren. Sie umzusetzen, ist eine ganz andere Sache. Dieses Buch stellt technologische Ansätze für die Umsetzung von Microservices vor und zeigt die jeweiligen Vor- und Nachteile.

Dabei geht es um Technologien für ein Microservices-System als Ganzes. Jeder Microservice kann anders implementiert werden. Daher sind die technologischen Entscheidungen für die Frameworks innerhalb der Microservices nicht so wichtig wie die Entscheidungen für das gesamte System. Die Entscheidung für ein Framework kann in jedem Microservice revidiert werden. Technologien für das Gesamtsystem sind kaum änderbar.

Grundlagen

Um Microservices zu verstehen, ist eine Einführung in die Architektur, ihre Vor- und Nachteile und Spielweisen unerlässlich. Die Grundlagen sind in dem Buch soweit erläutert, wie sie für das Verständnis der praktischen Umsetzungen notwendig sind.

Konzepte

Microservices benötigen Lösungen für verschiedene Herausforderungen. Dazu zählen Konzepte zur Integration (*Frontend-Integration, synchrone und asynchrone* Microservices) und zum Betrieb (*Monitoring, Log-Analyse, Tracing*). Microservices-Plattformen wie *PaaS* oder *Kubernetes* stellen vollständige Lösungen für den Betrieb von Microservices dar.

1. Eberhard Wolff: Microservices: Grundlagen flexibler Softwarearchitekturen, dpunkt.verlag, 2015, ISBN 978-3-86490-313-7

Rezepte

Das Buch nutzt Rezepte als Metapher für die Technologien, mit denen die Konzepte umgesetzt werden können. Jeder Ansatz hat viel mit einem Rezept gemeinsam:

- Jedes Rezept ist *praktisch* beschrieben, einschließlich einer technischen Implementierung als Beispiel. Bei den Beispielen liegt der Fokus auf *Einfachheit*. Jedes Beispiel kann leicht nachvollzogen, erweitert und modifiziert werden.
- Das Buch bietet dem Leser eine *Vielzahl von Rezepten*. Der Leser muss aus diesen Rezepten für sein Projekt *eine Auswahl* treffen, so wie ein Koch es für sein Menü tut. Das Buch zeigt verschiedene Optionen. In der Praxis muss fast jedes Projekt anders angegangen werden. Dazu bieten die Rezepte die Basis.
- Zu jedem Rezept gibt es *Rezept-Variationen*. Schließlich kann ein Rezept auf viele verschiedene Arten und Weisen umgesetzt werden. Das gilt auch für die Technologien in diesem Buch. Manchmal sind die Variationen so einfach, dass sie direkt als *Experiment* in dem ablauffähigen Beispiel umgesetzt werden können.

Für jedes Rezept gibt es ein *ablauffähiges Beispiel* mit der konkreten Technologie. Die Beispiele sind einzeln ablauffähig und bauen nicht aufeinander auf. So kann der Leser sich mit den für ihn interessantesten Rezepten und Beispielen beschäftigen, ohne sich dabei mit anderen Beispielen befassen zu müssen.

So liefert das Buch einen *Einstieg*, um einen *Überblick* über die Technologien zu bekommen und einen Technologie-Stack auszuwählen. Danach kann der Leser sich anhand der im Buch enthaltenen Links weiter in die relevanten Technologien vertiefen.

Aufbau des Buchs

Dieses Buch besteht aus drei Teilen.

Teil I – Architektur-Grundlagen

Teil I gibt eine Einführung in die Architektur-Grundlagen, die mit Kapitel 1 beginnt.

- Kapitel 1 klärt den Begriff »Microservice« und Kapitel 3 erläutert Self-contained Systems als besonders praktikablen Ansatz für Microservices.
- In einem Microservices-System gibt es die Ebenen der Mikro- und Makro-Architektur, die globale und lokale Entscheidungen darstellen (Kapitel 2).
- Oft sollen alte Systeme in Microservices migriert werden (Kapitel 4).

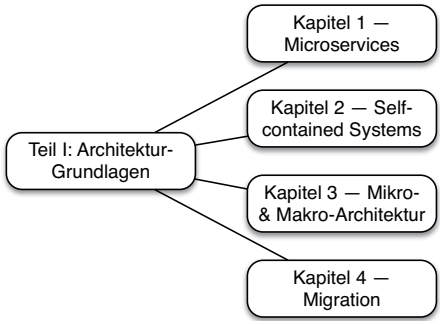

Abb. 1 *Überblick über Teil I*

Teil II – Technologie-Stacks

Technologie-Stacks stehen im Mittelpunkt von Teil II, der mit Kapitel 5 beginnt.

- *Docker* ist die Basis vieler Microservices-Architekturen (Kapitel 5). Es erleichtert das Ausrollen von Software und den Betrieb der Services.
- Die *technische Mikro-Architektur* (Kapitel 6) beschreibt Technologien, die zur Implementierung eines Microservice genutzt werden können.
- Eine Möglichkeit zur Integration ist das Konzept zur *Integration am Web-Frontend* (Kapitel 7). Die Frontend-Integration führt zu einer losen Kopplung der Microservices und einer hohen Flexibilität.
  - Das Rezept aus Kapitel 8 setzt für die Web-Frontend-Integration auf *Links* und auf *JavaScript* für das dynamische Nachladen von Inhalten. Dieser Ansatz ist einfach realisierbar und nutzt gängige Web-Technologien.
  - Auf dem Server kann die Integration mit *ESI (Edge Side Includes)* erfolgen (Kapitel 9). ESI ist in Caches implementiert, sodass das System eine höhere Performance und Zuverlässigkeit erreichen kann.
- Das Konzept der *asynchronen Kommunikation* steht im Mittelpunkt von Kapitel 10. Asynchrone Kommunikation verbessert die Zuverlässigkeit und entkoppelt das Systems.
  - Eine asynchrone Technologie ist *Apache Kafka* (Kapitel 11), mit der Messages verschickt werden können. Kafka speichert die Nachrichten dauerhaft ab und erlaubt so die Rekonstruktion des Zustands eines Microservices aus den Nachrichten.
  - Die Alternative für asynchrone Kommunikation ist *Atom* (Kapitel 12), ein HTTP- und REST-basiertes Protokoll. Atom nutzt eine REST-Infrastruktur und kann daher sehr einfach umgesetzt werden.
- Kapitel 13 stellt vor, wie das Konzept *synchroner Microservices* umgesetzt werden kann. Die synchrone Kommunikation zwischen Microservices wird in der Praxis sehr häufig genutzt, obwohl dieser Ansatz bei Antwortzeiten und Zuverlässigkeit Herausforderungen bereithalten kann.

- Der *Netflix-Stack* (Kapitel 14) bietet Eureka für Service Discovery, Ribbon für Load Balancing, Hystrix für Resilience und Zuul für Routing. Netflix wird vor allem in der Java Community breit genutzt.
- *Consul* (Kapitel 15) ist eine Alternative für Service Discovery. Consul hat sehr viele Features und kann mit einer breiten Palette an Technologien genutzt werden.

Kapitel 16 erläutert das Konzept der *Microservices-Plattformen*, die den Betrieb und die Kommunikation der Microservices unterstützen.

- *Kubernetes* (Kapitel 17) ist eine Microservices-Plattform, die Docker Container ausführen kann, aber auch Service Discovery und Load Balancing löst. Der Microservice bleibt von dieser Infrastruktur unabhängig.
- *PaaS (Platform as a Service)* ist eine weitere Microservices-Plattform (Kapitel 18), die am Beispiel Cloud Foundry erläutert wird. Cloud Foundry ist sehr flexibel und kann auch im eigenen Rechenzentrum betrieben werden.

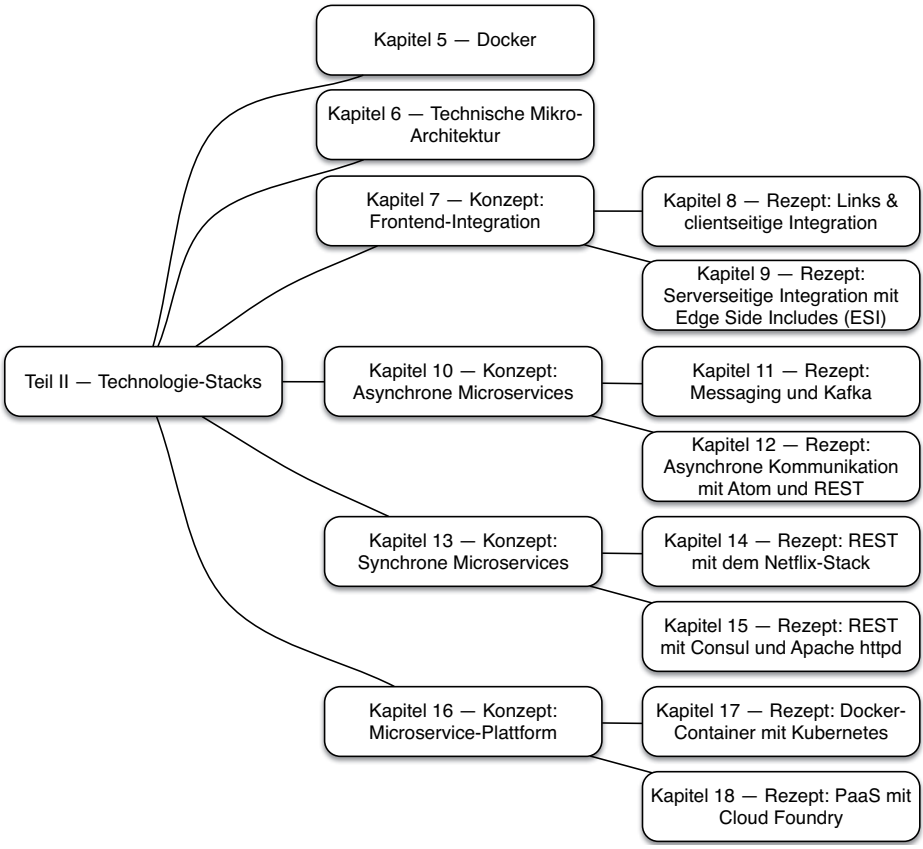

Abb. 2 *Überblick über Teil II*

Teil III – Betrieb

Den *Betrieb* einer Vielzahl von Microservices sicherzustellen, ist eine große Herausforderung. Teil III (ab Kapitel 19) diskutiert mögliche Rezepte zur Lösung.

- Das Kapitel 19 erläutert *Grundlagen*, und warum der Betrieb von Microservices so schwierig ist.
- Im Kapitel 20 geht es um *Monitoring* und das Werkzeug Prometheus. Prometheus unterstützt multidimensionale Datenstrukturen und kann die Monitoring-Werte auch von vielen Microservice-Instanzen analysieren.
- Die *Analyse von Log-Daten* steht im Mittelpunkt von Kapitel 21. Als Werkzeug zeigt das Kapitel den Elastic Stack. Dieser Stack ist sehr weit verbreitet und stellt eine gute Basis für die Analyse auch großer Log-Datenmengen dar.
- *Tracing* verfolgt Aufrufe zwischen Microservices (Kapitel 22). Dazu kommt Zipkin zum Einsatz. Zipkin unterstützt verschiedene Plattformen und stellt einen De-facto-Standard für Tracing dar.

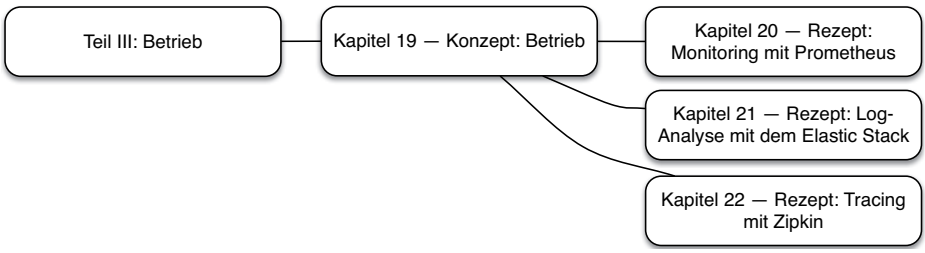

Abb. 3 *Überblick über Teil III*

Abschluss und Anhänge

Abschließend bietet das Kapitel 23 noch einen *Ausblick*.

Die Anhänge erklären die Installation der Software (Anhang A), die Benutzung des Build-Werkzeugs Maven (Anhang B) sowie Docker und Docker Compose (Anhang C), mit denen die Umgebungen für die Beispiele betrieben werden.

Zielgruppe

Das Buch erläutert Grundlagen und technische Aspekte von Microservices. Es ist für verschiedene Zielgruppen interessant:

- *Entwicklern* bietet Teil II eine Hilfe bei der Auswahl eines geeigneten Technologie-Stacks. Die Beispielprojekte dienen als Basis für das Einarbeiten in die Technologien. Die Microservices in den Beispielprojekten sind in Java mit dem Spring-Framework geschrieben, aber die Technologien in den Beispielen dienen zur Integration von Microservices, sodass weitere Microservices in anderen Sprachen ergänzt werden können. Teil III rundet das Buch in Rich-

tung Betrieb ab, der für Entwickler immer wichtiger wird, und Teil I erläutert die grundlegenden Architektur-Konzepte.

- *Architekten* vermittelt Teil I das grundlegende Wissen über Microservices. Teil II und Teil III zeigen praktische Rezepte und Technologien, um die Architekturen umzusetzen. Damit geht das Buch weiter als ein reines Microservices-Architektur-Buch.
- Für Experten aus den Bereichen *DevOps* und *Betrieb* stellen die Rezepte in Teil III eine Basis für eine Technologie-Bewertung von Betriebsaspekten wie Log-Analyse, Monitoring und Tracing von Microservices dar. Teil II zeigt Technologien für Deployment wie Docker, Kubernetes oder Cloud Foundry. Teil I beschreibt als Hintergrund die Konzepte hinter dem Microservices-Architektur-Ansatz.
- *Manager* bekommen in Teil I einen Überblick über die Vorteile des Architektur-Ansatzes und die besonderen Herausforderungen. Sie können bei Interesse an technischen Details Teil II und Teil III lesen.

Vorwissen

Das Buch setzt grundlegendes Wissen über Software-Architektur und Software-Entwicklung voraus. Die praktischen Beispiele sind so dokumentiert, dass sie mit wenig Vorwissen ausgeführt werden können. Das Buch fokussiert auf Technologien, die für Microservices in verschiedenen Programmiersprachen genutzt werden können. Die Beispiele sind in Java mit den Frameworks Spring Boot und Spring Cloud geschrieben, sodass für Änderungen an dem Code Java-Kentnisse notwendig sind.

Quick Start

Das Buch vermittelt vor allem Technologien. Zu jeder Technologie in jedem Kapitel gibt es ein Beispiel. Um schnell praktische Erfahrungen mit den Technologien zu sammeln und anhand der Beispiele nachzuvollziehen, gibt es einen Quick Start:

- Zunächst muss auf dem Rechner die notwendige Software *installiert* sein. Die Installation beschreibt Anhang A.
- Der Build der Beispiele erfolgt mit *Maven*. Den Umgang mit Maven erläutert Anhang B.
- Die Beispiele setzen alle auf *Docker* und *Docker Compose* auf. Das Anhang C beschreibt die wichtigsten Befehle für Docker und Docker Compose.

Sowohl für den Build mit Maven als auch für Docker und Docker Compose enthalten die Kapitel Anleitungen zum Troubeshooting.

Die Beispiele sind in folgenden Abschnitten erläutert:

Konzept	Rezept	Abschnitt
Frontend-Integration	Links & clientseitige Integration	8.2
Frontend-Integration	Edge Side Includes (ESI)	9.2
Asynchrone Microservices	Kafka	11.4
Asynchrone Microservices	REST & Atom	12.2
Synchrone Microservices	Netflix-Stack	14.1
Synchrone Microservices	Consul & Apache httpd	15.1
Microservices-Plattform	Kubernetes	17.3
Microservices-Plattform	Cloud Foundry	18.3
Betrieb	Monitoring mit Prometheus	20.4
Betrieb	Log-Analyse mit Elastic Stack	21.3
Betrieb	Tracing mit Zipkin	22.2

Die Projekte sind alle auf GitHub verfügbar. In den Projekten gibt es jeweils eine Datei `WIE-LAUFEN.md` mit einer Schritt-für-Schritt-Anleitung, wie die Demos installiert und gestartet werden können.

Die Beispiele bauen nicht aufeinander auf. Dadurch ist es möglich, mit einem beliebigen Beispiel loszulegen.

Danksagung

Ich möchte allen danken, mit denen ich über Microservices diskutiert habe, die mir Fragen gestellt oder mit mir zusammengearbeitet haben. Es sind viel zu viele, um sie alle zu nennen. Der Dialog hilft sehr und macht Spaß!

Viele der Ideen und auch die Umsetzungen sind ohne meine Kollegen bei der innoQ nicht denkbar. Insbesondere möchte ich Alexander Heusingfeld, Christian Stettler, Christine Koppelt, Daniel Westheide, Gerald Preissler, Jörg Müller, Lucas Dohmen, Marc Giersch, Michael Simons, Michael Vitz, Philipp Neugebauer, Simon Kölsch, Sophie Kuna und Stefan Lauer danken.

Weiteres wichtiges Feedback kam von Merten Driemeyer und Olcay Tümce.

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.

Bei den Entwicklern der Werkzeuge von *https://www.softcover.io/* möchte ich mich ebenfalls bedanken.

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

Website

Die Website zum Buch ist *http://microservices-praxisbuch.de/*. Dort finden sich die Errata und Links zu den Beispielen.

Teil I

Architekturgrundlagen

Der erste Teil des Buchs stellt die grundlegenden Ideen der Microservices-Architektur vor.

Microservices

Das Kapitel 1 klärt die Grundlagen von *Microservices*: Was sind Microservices? Welche Vor- und Nachteile hat diese Architektur?

Self-contained Systems

Das Kapitel 3 beschreibt *Self-contained Systems*. Sie sind eine Sammlung von Best Practices für Microservices-Architekturen, bei der eine starke Unabhängigkeit und Web-Anwendungen im Mittelpunkt stehen. Neben Vor- und Nachteilen geht es um mögliche Variationen dieser Idee.

Mikro- und Makro-Architektur

Microservices bieten viele Freiheiten. Dennoch müssen einige Entscheidungen übergreifend über alle Microservices eines Systems getroffen werden. Das Kapitel 2 stellt das Konzept der *Mikro- und Makro-Architektur* vor. Die Mikro-Architektur umfasst alle Entscheidungen, die für jeden Microservice anders getroffen wer-

den können. Die Makro-Architektur sind die Entscheidungen, die für alle Microservices gelten. Neben den Bestandteilen einer Mikro- und Makro-Architektur stellt das Kapitel auch vor, wer eine Makro-Architektur entwirft.

Migration

Die meisten Microservices-Projekte migrieren ein vorhandenes System in eine Microservices-Architektur. Daher stellt das Kapitel 4 mögliche Ziele einer *Migration* und verschiedene Migrationsstrategien vor.

1 Microservices

Dieses Kapitel bietet eine Einführung in das Thema »Microservices«. Das Studium dieses Kapitels vermittelt dem Leser:

- Vorteile (Abschnitt 1.2) und Nachteile (Abschnitt 1.3) von Microservices, um die Einsetzbarkeit dieses Architektur-Ansatzes in einem konkreten Projekt abschätzen zu können.
- Die Vorteile zeigen auf, welche Probleme Microservices lösen und wie der Architekturansatz für bestimmte Szenarien angepasst werden kann.
- Die Nachteile verdeutlichen, wo technische Risiken auftauchen können und wie man mit ihnen umgehen kann.
- Schließlich haben Vor- und Nachteile Einfluss auf Technologie- und Architektur-Entscheidungen, die Vorteile verstärken und Nachteile vermindern sollen.

1.1 Microservices: Definition

Leider gibt es für den Begriff »Microservice« keine allgemein anerkannte Definition. Im Rahmen dieses Buchs gilt folgende Definition:

Microservices sind unabhängig deploybare Module.

Beispielsweise kann ein E-Commerce-System in Module für den Bestellprozess, die Registrierung oder die Produktsuche aufgeteilt werden. Normalerweise wären alle diese Module gemeinsam in einer Anwendung implementiert. Dann kann eine Änderung in einem der Module nur in Produktion gebracht werden, indem eine neue Version der Anwendung und damit aller Module in Produktion gebracht wird. Wenn die Module aber als Microservices umgesetzt sind, kann der Bestellprozess nicht nur unabhängig von den anderen Modulen geändert werden, sondern er kann sogar unabhängig in Produktion gebracht werden.

Das beschleunigt das Deployment und verringert die Anzahl der notwendigen Tests, da nur ein Modul deployt wird. Im Extremfall wird durch die größere Entkopplung ein großes Projekt zu einer Menge kleinerer Projekte, die jeweils einen der Microservices verantworten.

Technisch ist es dazu notwendig, dass der Microservice ein eigener Prozess ist. Besser wäre eine eigene virtuelle Maschine oder ein Docker-Container, die Microservices noch stärker entkoppeln. Ein Deployment ersetzt dann den Docker-Container durch einen neuen Docker-Container, fährt die neue Version hoch und lässt dann die Request auf die Version umschwenken. Die anderen Microservices bleiben davon unbeeinflusst.

1.1.1 Vorteile der Microservices-Definition

Diese Definition von Microservices als unabhängig deploybare Module hat mehrere Vorteile:

- Sie ist sehr *kompakt*.
- Sie ist sehr *allgemein* und umfasst praktisch alle Arten von Systemen, die üblicherweise als Microservices bezeichnet werden.
- Die Definition beruft sich auf *Module* und damit auf ein altes, gut verstandenes Konzept. So können viele Ideen zur Modularisierung übernommen werden. Außerdem wird so deutlich, dass Microservices Teile eines größeren Systems sind und niemals für sich stehen können. Deswegen müssen Microservices zwangsläufig mit anderen Microservices integriert werden.
- Das unabhängige Deployment ist eine Eigenschaft, die zu vielen Vorteilen führt (siehe Abschnitt 1.2) und daher sehr wichtig ist. So zeigt die Definition trotz der Kürze, was die *wesentliche Eigenschaft* eines Microservices tatsächlich ist.

1.1.2 Deployment-Monolith

Ein System, das nicht aus Microservices besteht, kann nur als Ganzes deployt werden. Es ist ein *Deployment-Monolith*. Natürlich kann der Deployment-Monolith in Module aufgeteilt sein. Über den internen Aufbau sagt dieser Begriff nichts aus.

1.1.3 Größe eines Microservice

Die Definition von Microservices trifft keine Aussage über die Größe eines Microservice. Der Name »Microservice« legt den Verdacht nahe, dass es um besonders kleine Services geht. Aber in der Praxis findet man sehr unterschiedliche Größen von Microservices. Einige Microservices beschäftigen ein ganzes Team, während andere nur hundert Zeilen lang sind. Die Größe eignet sich also tatsächlich nicht als Teil der Definition.

1.2 Gründe für Microservices

Für die Nutzung von Microservices gibt es eine Vielzahl von Gründen.

1.2.1 Microservices zum Skalieren der Entwicklung

Ein Grund für den Einsatz von Microservices ist die Skalierung der Entwicklung. Große Teams sollen gemeinsam an einem komplexen Projekt arbeiten. Mithilfe von Microservices können die Teams weitgehend unabhängig arbeiten:

- Die meisten technischen Entscheidungen können die Teams allein treffen. Wenn die Microservices als Docker-Container ausgeliefert werden, muss jeder Docker-Container nur eine Schnittstelle für andere Container anbieten. Der interne Aufbau des Containers ist egal, solange die Schnittstelle vorhanden ist und korrekt funktioniert. Deswegen ist es beispielsweise egal, in welcher Programmiersprache der Microservice geschrieben wurde. Also kann das Team diese Entscheidung allein treffen. Natürlich kann die Wahl der Programmiersprache eingeschränkt werden, um Wildwuchs und zu große Komplexität zu vermeiden. Aber auch wenn die Wahl der Programmiersprache in einem Projekt eingeschränkt worden ist: Ein Bug Fix für eine Library kann ein Team immer noch unabhängig von den anderen Teams in einen Microservice einbauen.
- Wenn ein neues Feature nur Änderungen an einem Microservice benötigt, kann es nicht nur unabhängig entwickelt werden, sondern es kann auch unabhängig in Produktion gebracht werden. So können die Teams vollständig unabhängig an Features arbeiten und sind fachlich unabhängig.

Durch Microservices können die Teams somit fachlich und technisch unabhängig arbeiten. Das erlaubt es, auch große Projekte ohne großen Koordinierungsaufwand zu stemmen.

1.2.2 Legacy-Systeme ablösen

Die Wartung und Erweiterung eines Legacy-Systems ist eine Herausforderung, weil der Code meistens schlecht strukturiert ist und Änderungen oft nicht durch Tests abgesichert sind. Dazu kann noch eine veraltete technologische Basis kommen.

Microservices helfen bei der Arbeit mit Legacy-Systemen, weil der Code nicht unbedingt geändert werden muss. Stattdessen können neben das alte System neue Microservices gestellt werden. Dazu ist eine Integration zwischen dem alten System und den Microservices notwendig – beispielsweise per Datenreplikation, per REST, Messaging oder auf der UI-Ebene. Außerdem müssen Probleme wie ein einheitliches Single Sign On über das alte System und die neuen Microservices gelöst werden.

Dafür sind die Microservices dann praktisch ein Greenfield: Es gibt keine vorhandene Codebasis, auf die aufgesetzt werden muss. Ebenso kann ein komplett anderer Technologiestack genutzt werden. Das erleichtert die Arbeit gegenüber einer Modifikation des Legacy-Systems erheblich.

1.2.3 Nachhaltige Entwicklung

Microservices versprechen, dass Systeme auch langfristig wartbar bleiben.

Ein wichtiger Grund dafür ist die Ersetzbarkeit der Microservices. Wenn ein einzelner Microservice nicht mehr wartbar ist, kann er neu geschrieben werden. Das ist im Vergleich zu einem Deployment-Monolithen mit weniger Aufwand verbunden, weil die Microservices kleiner sind als ein Deployment-Monolith.

Allerdings ist es schwierig, einen Microservice zu ersetzen, von dem viele andere Microservices abhängen, weil Änderungen die anderen Microservices beeinflussen können. Also müssen für die Ersetzbarkeit auch die Abhängigkeiten zwischen den Microservices gemanagt werden.

Die Ersetzbarkeit ist eine wesentliche Stärke von Microservices. Viele Entwickler arbeiten daran, Legacy-Systeme zu ersetzen. Aber beim Entwurf eines neues Systems wird viel zu selten die Frage gestellt, wie das System abgelöst werden kann, wenn es zu einem Legacy-System geworden ist. Die Ersetzbarkeit von Microservices ist eine mögliche Antwort.

Für die Wartbarkeit müssen die Abhängigkeiten zwischen den Microservices langfristig gemanagt werden. Auf dieser Ebene haben klassische Architekturen oft Schwierigkeiten: Ein Entwickler schreibt Code und führt dabei unabsichtlich eine neue Abhängigkeit zwischen zwei Modulen ein, die eigentlich in der Architektur verboten war. Das merkt der Entwickler üblicherweise noch nicht einmal, weil er nicht die Architektur-Ebene, sondern nur die Code-Ebene des Systems im Blick hat. Aus welchem Modul die Klasse stammt, zu der er gerade eine Abhängigkeit einführt, ist oft nicht sofort zu erkennen. So entstehen mit der Zeit immer mehr Abhängigkeiten. Gegen die ursprüngliche Architektur mit den geplanten Abhängigkeiten wird immer mehr verstoßen und am Ende steht ein völlig unstrukturiertes System.

Microservices haben klare Grenzen durch ihre Schnittstelle – egal ob die Schnittstelle als REST-Schnittstelle oder durch Messaging implementiert ist. Wenn ein Entwickler eine neue Abhängigkeit zu einer solchen Schnittstelle einführt, merkt er das, weil die Schnittstelle entsprechend bedient werden muss. Aus diesem Grund ist es unwahrscheinlich, dass auf der Ebene der Abhängigkeiten zwischen den Microservices Architektur-Verstöße geschehen. Die Schnittstellen der Microservices sind sozusagen Architektur-Firewalls, weil sie Architektur-Verstöße aufhalten. Das Konzept einer Architektur-Firewall setzen auch Architektur-Managementwerkzeuge wie Sonargraph (*https://www.hello2morrow.com/products/sonargraph*), Structure101 (*http://structure101.com/*) oder jQAssistant (*https://jqassistant.org/*) um. Fortgeschrittene Modul-Konzepte können ebenfalls

solche Firewalls erzeugen. In der Java-Welt beschränkt OSGi (*https://www.osgi.org/*) andere Module auf den Zugriff über die Schnittstelle. Der Zugriff kann sogar auf einzelne Packages oder Klassen eingeschränkt werden.

Also bleiben einzelne Microservices wartbar, weil sie ersetzt werden können, wenn sie nicht mehr wartbar sind. Die Architektur auf Ebene der Abhängigkeiten zwischen den Microservices bleibt ebenfalls wartbar, weil Entwickler Abhängigkeiten zwischen Microservices nicht mehr unbeabsichtigt einbauen können.

Daher können Microservices langfristig eine hohe Qualität der Architektur sicherstellen und damit eine nachhaltige Entwicklung, bei der die Änderungsgeschwindigkeit auch langfristig nicht abnimmt.

1.2.4 Continuous Delivery

Continuous Delivery[1] ist ein Ansatz, bei dem Software kontinuierlich in Produktion gebracht wird. Dazu wird eine Continuous-Delivery-Pipeline genutzt. Die Pipeline bringt die Software durch die verschiedenen Phasen in Produktion (siehe Abbildung 1–1).

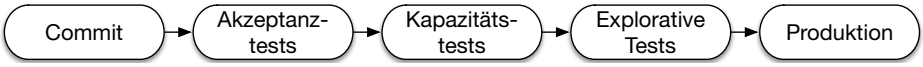

Abb. 1–1 *Continuous-Delivery-Pipeline*

Typischerweise wird die Software in der Commit-Phase kompiliert, die Unit Tests und eine statische Code-Analyse werden durchgeführt. In der Akzeptanztestphase überprüfen automatisierte Tests die fachlich korrekte Funktion der Software. Die Kapazitätstests überprüfen die Performance für die zu erwartende Last. Explorative Tests dienen dazu, bisher noch nicht bedachte Tests durchzuführen oder neue Funktionalitäten zu testen. Die explorativen Tests können so Aspekte untersuchen, die automatisierte Tests noch nicht abdecken. Am Ende wird die Software in Produktion gebracht.

Microservices stellen unabhängig deploybare Module dar. Also hat jeder Microservice eine eigene Continuous-Delivery-Pipeline. Das erleichtert Continuous Delivery:

- Der Durchlauf durch die Continuous-Delivery-Pipelines ist wesentlich *schneller*, weil die Deployment-Einheiten kleiner sind. Daher ist das Deployment schneller und so können Tests schneller Umgebungen aufbauen. Auch die Tests sind schneller, da sie weniger Funktionalitäten testen müssen. Nur die Features im jeweiligen Microservice müssen getestet werden, während bei einem Deployment-Monolithen wegen möglicher Regressionen die gesamte Funktionalität getestet werden muss.

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

- Der Aufbau der Continuous-Delivery-Pipeline ist *einfacher*. Der Aufbau einer Umgebung für einen Deployment-Monolithen ist kompliziert. Meistens werden leistungsfähige Server benötigt. Ebenso sind oft Drittsysteme für Tests notwendig. Ein Microservice braucht weniger leistungsfähige Hardware. Es sind auch nicht so viele Drittsysteme in den Testumgebungen notwendig. Es kann allerdings notwendig sein, die Microservices zusammen in einem Integrationstest zu testen. Das kann diesen Vorteil zunichte machen.
- Das Deployment eines Microservice hat ein *geringeres Risiko* als das Deployment eines Deployment-Monolithen. Bei einem Deployment Monolithen wird das komplette System neu deployt, bei einem Microservice nur ein Modul. Dabei sind weniger Probleme zu erwarten, weil weniger Funktionalität geändert wird.

Microservices helfen also bei Continuous Delivery. Die bessere Unterstützung von Continuous Delivery alleine kann schon ein Grund für eine Migration eines Deployment-Monolithen zu Microservices sein.

Microservices-Architekturen können aber nur dann funktionieren, wenn das Deployment automatisiert ist. Microservices erhöhen die Anzahl der deploybaren Einheiten gegenüber einem Deployment-Monolithen erheblich. Das ist nur machbar, wenn die Deployment-Prozesse automatisiert werden.

Tatsächlich unabhängiges Deployment bedeutet, dass die Continuous-Delivery-Pipelines vollständig unabhängig sind. Integrationstests widersprechen dieser Unabhängigkeit: Sie führen Abhängigkeiten zwischen den Continuous-Delivery-Pipelines verschiedener Microservices ein. Also müssen die Integrationstests auf ein Minimum reduziert werden. Abhängig von der Kommunikationsart gibt es dafür unterschiedliche Ansätze (siehe Abschnitt 13.1 und Abschnitt 10.3).

1.2.5 Robustheit

Microservices-Systeme sind robuster. Wenn in einem Microservice ein Speicherleck existiert, stürzt nur dieser Microservice ab. Die anderen Microservices laufen weiter. Natürlich müssen die anderen Microservices den Ausfall eines Microservice kompensieren. Man spricht von Resilience (etwa Widerstandsfähigkeit). Microservices können dazu beispielsweise Werte cachen und diese Werte bei einem Ausfall nutzen. Oder es gibt einen Fallback mit einem vereinfachten Algorithmus.

Ohne Resilience kann die Verfügbarkeit eines Microservices-Systems problematisch sein. Dass irgendein Microservice ausfällt, ist recht wahrscheinlich. Durch die Aufteilung in mehrere Prozesse sind viel mehr Server an dem System beteiligt. Jeder dieser Server kann ausfallen. Die Kommunikation zwischen den Microservices verläuft über das Netzwerk. Das Netzwerk kann ebenfalls ausfallen. Also müssen die Microservices Resilience umsetzten, um Robustheit zu erreichen.

Der Abschnitt 14.5 zeigt, wie Resilience in einem synchronen Microservice-System konkret umgesetzt werden kann.

1.2.6 Unabhängige Skalierung

Jeder Microservice kann unabhängig skaliert werden: Es ist möglich, mehr Instanzen eines Microservices zu starten und die Last für den Microservice auf die Instanzen zu verteilen. Das kann die Skalierbarkeit eines Systems erheblich verbessern. Dazu müssen die Microservices natürlich entsprechende Voraussetzungen schaffen. So dürfen die Microservices keinen State enthalten. Sonst können Clients nicht auf eine andere Instanz umschwenken, die ja den State dann nicht hätte.

Mehr Instanzen eines Deployment-Monolithen zu starten, kann aufgrund der benötigten Hardware schwierig sein. Außerdem kann der Aufbau einer Umgebung für einen Deployment-Monolithen komplex sein: So können zusätzliche Dienste notwendig sein oder eine komplexe Infrastruktur mit Datenbanken und weiteren Software-Komponenten. Bei einem Microservice kann die Skalierung feingranularer erfolgen, sodass üblicherweise weniger zusätzliche Dienste notwendig sind und Rahmenbedingungen weniger komplex sind.

1.2.7 Technologiewahlfreiheit

Jeder Microservice kann mit einer eigenen Technologie umgesetzt werden. Das erleichtert die Migration auf eine neue Technologie, da man jeden Microservice einzeln migrieren kann. Ebenso ist es einfacher und risikoärmer, Erfahrungen mit neuen Technologien zu sammeln. Sie können zunächst nur für einen Microservice genutzt werden, bevor sie in mehreren Microservices zum Einsatz kommen.

1.2.8 Sicherheit

Microservices können untereinander isoliert werden. So ist es möglich, zwischen den Microservices Firewalls in die Kommunikation einzuführen. Außerdem kann die Kommunikation zwischen den Microservices abgesichert werden, um zu garantieren, dass die Kommunikation tatsächlich von einem anderen Microservice kommt und authentisch ist. So kann verhindert werden, dass bei einer Übernahme eines Microservices auch andere Microservices kompromittiert sind.

1.2.9 Allgemein: Isolation

Letztendlich lassen sich viele Vorteile der Microservices auf eine stärkere Isolation zurückführen.

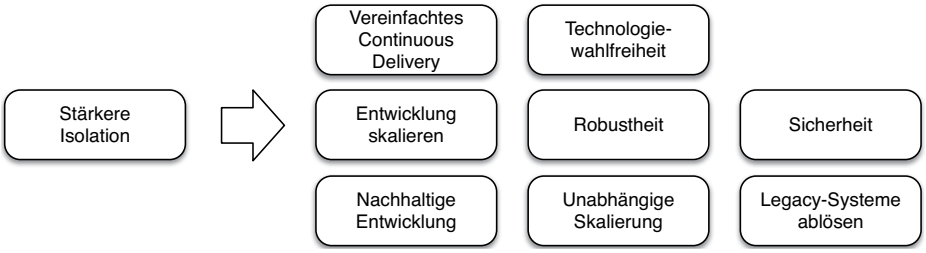

Abb. 1–2 *Isolation als Quelle der Vorteile*

Microservices können isoliert deployt werden, was Continuous Delivery vereinfacht. Sie sind bezüglich Ausfällen isoliert, was der Robustheit zugute kommt. Gleiches gilt für Skalierbarkeit: Jeder Microservice kann isoliert von anderen Microservices skaliert werden. Die eingesetzten Technologien können isoliert für einen Microservice bestimmt werden, was Technologiewahlfreiheit ermöglicht. Die Microservices sind so isoliert, dass sie nur über das Netzwerk miteinander kommunizieren. Die Kommunikation kann daher durch Firewalls abgesichert werden, was der Sicherheit zugute kommt.

Weil dank der starken Isolation die Modulgrenzen kaum noch aus Versehen überschritten werden können, wird die Architektur kaum noch verletzt. Das sichert die Architektur im Großen ab. Jeder Microservice kann isoliert durch einen neuen ersetzt werden. So kann risikoarm ein Microservice abgelöst werden und die Architektur der einzelnen Microservices sauber gehalten werden. Dadurch ermöglicht die Isolation eine langfristige Wartbarkeit der Software.

Mit der Isolation treiben Microservices die Entkopplung als wichtige Eigenschaft von Modulen auf die Spitze: Während Module normalerweise nur bezüglich Änderungen am Code und bezüglich der Architektur voneinander entkoppelt sind, geht die Entkopplung der Microservices darüber weit hinaus.

1.2.10 Vorteile priorisieren

Welcher Grund für Microservices der wichtigste ist, hängt vom jeweiligen Szenario ab. Der Einsatz von Microservices in einem Greenfield-System ist eher die Ausnahme als die Regel. Oft gilt es, einen Deployment-Monolithen durch ein Microservices-System zu ersetzen (siehe auch Kapitel 4). Dabei spielen unterschiedliche Vorteile eine Rolle:

- Die einfachere *Skalierung der Entwicklung* kann ein wichtiger Grund für die Einführung von Microservices in einem solchen Szenario sein. Oft haben große Organisationen Schwierigkeiten, einen Deployment-Monolithen mit einer Vielzahl von Entwicklern schnell genug weiterzuentwickeln.
- Die *einfache Migration* weg von dem Legacy-Deployment-Monolithen erleichtert die Einführung von Microservices in einem solchen Szenario.