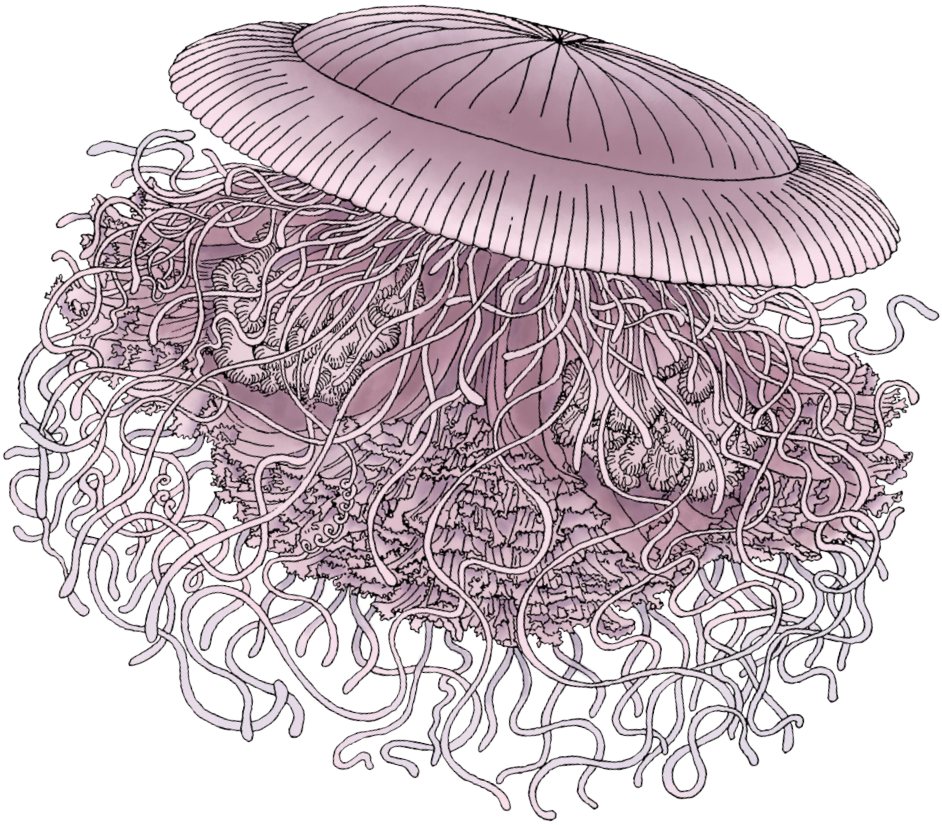
O'REILLY®

Vom Monolithen zu Microservices

Patterns, um bestehende Systeme Schritt für Schritt umzugestalten



Sam Newman

Übersetzung von Thomas Demmig

Vom Monolithen zu Microservices

Patterns, um bestehende Systeme Schritt für Schritt umzugestalten

Sam Newman

Deutsche Übersetzung von Thomas Demmig

O'REILLY®

Sam Newman

Lektorat: Ariane Hesse
Übersetzung: Thomas Demmig
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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

ISBN:
Print 978-3-96009-140-0
PDF 978-3-96010-423-0
ePub 978-3-96010-424-7
mobi 978-3-96010-425-4

1. Auflage 2021

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Monolith to Microservices* ISBN 9781492047841

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

5 4 3 2 1 0

Inhalt

Vorwort

Noch vor ein paar Jahren plauderten manche von uns über Microservices als eine interessante Idee. Inzwischen ist es im Handumdrehen zur Standardarchitektur für Hunderte von Firmen auf der ganzen Welt geworden (von denen viele eventuell als Start-ups begannen, um die Probleme zu lösen, die durch Microservices verursacht werden), und jeder versucht, noch auf den fahrenden Zug aufzuspringen, bevor er hinter dem Horizont verschwindet.

Ich muss zugeben, dass ich daran nicht ganz unschuldig bin. Seit ich 2015 mein Buch *Building Microservices* (*https://oreil.ly/building-microservices-2e*) zu diesem Thema schrieb, habe ich mein Geld damit verdient, Menschen dabei zu helfen, diese Art von Architektur zu verstehen. Mein Ziel war immer, jenseits des Hypes Firmen dabei zu unterstützen, für sich herauszufinden, ob Microservices für sie das Richtige sind. Für viele meiner Kunden mit bestehenden (nicht auf Microservices ausgerichteten) Systemen lag die Herausforderung darin, wie die Microservices-Architektur übernommen werden konnte. Wie nehmen Sie ein bestehendes System und passen seine Architektur an, ohne die ganzen anderen Aufgaben zu kurz kommen zu lassen? Darum geht es in diesem Buch. Ich möchte Ihnen dabei auch ganz ehrlich zeigen, welchen Herausforderungen Sie sich bei der Microservices-Architektur gegenübersehen werden, und Ihnen dabei helfen, herauszufinden, ob diese Reise für Sie überhaupt die richtige ist.

Was Sie lernen werden

Dieses Buch soll tief in das Thema einsteigen: Wie planen Sie das Aufbrechen bestehender Systeme in eine Microservices-Architektur, und wie setzen Sie das dann auch um? Wir werden viele Aspekte dazu ansprechen, aber der Schwerpunkt liegt auf dem Auseinandernehmen von Dingen. Eine allgemeinere Beschreibung finden Sie in meinem vorherigen Buch *Building Microservices*. Tatsächlich möchte ich Ihnen das Buch als Begleitung zu diesem Buch sehr ans Herz legen.

Kapitel 1 liefert einen Überblick darüber, was Microservices sind und welche Ideen hinter dieser Art von Architekturen stehen. Es sollte denjenigen helfen, für die dieses Thema neu ist. Aber auch wenn Sie schon mehr Erfahrung mit Microservices haben, empfehle ich Ihnen, es nicht zu überspringen. Denn ich habe das Gefühl, dass einige der zentralen Ideen von Microservices bei all der technologischen Hektik leicht verloren gehen: Es sind Konzepte, auf die dieses Buch immer wieder zurückgreifen wird.

Es ist zwar gut, mehr über Microservices zu wissen, aber es ist doch etwas anderes, wenn Sie herausfinden wollen, ob sie das Richtige für Sie sind. In Kapitel 2 zeige ich Ihnen, wie Sie feststellen können, ob Microservices zu Ihnen passen, und gebe Ihnen wichtige Richtlinien dafür an die Hand, wie Sie einen Übergang von einer monolithischen zu einer Microservices-Architektur organisieren. Hier werden wir alles ansprechen – vom Domain-Driven Design bis hin zu Modellen zur Veränderung von Organisationen. Das sind wichtige Grundlagen, die Ihnen auch dann helfen, wenn Sie sich dafür entscheiden, keine Microservices-Architektur umzusetzen.

In Kapitel 3 und 4 steigen wir tiefer in die technischen Aspekte ein, die zum Auseinandernehmen eines Monolithen gehören, wir schauen uns Beispiele aus der realen Welt an und ziehen daraus unsere Migrations-Patterns. In Kapitel 3 geht es vor allem um die Aspekte des Dekonstruierens, während Kapitel 4 tief in die Datenthematik einsteigt. Wollen Sie wirklich von einem monolithischen System zu einer Microservices-Architektur wechseln, müssen wir auch ein paar Datenbanken auseinandernehmen!

In Kapitel 5 geht es schließlich um die Herausforderungen, denen Sie sich gegenübersehen, wenn Ihre Microservices-Architektur wächst. Diese Systeme können große Vorteile bieten, aber sie bringen auch viel Komplexität und diverse Probleme mit, die Sie zuvor nicht hatten. Dieses Kapitel ist mein Versuch, Ihnen dabei zu helfen, diese Probleme schon dann zu erkennen, wenn sie sich gerade erst entwickeln, und Ihnen Wege aufzuzeigen, wie Sie mit den wachsenden Schmerzen umgehen, die mit Microservices verbunden sind.

Konventionen in diesem Buch

Die folgenden typografischen Konventionen werden in diesem Buch genutzt:

Kursiv
: Für neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateierweiterungen.

`Nichtproportionalschrift`
: Für Programmlistings, aber auch für Codefragmente in Absätzen, wie zum Beispiel Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter.

Dieses Symbol steht für einen Tipp oder Vorschlag.

Dieses Symbol steht für eine allgemeine Anmerkung.

Dieses Symbol steht für eine Warnung oder Vorsichtsmaßnahme.

Danksagung

Ohne die Hilfe und das Verständnis meiner wundervollen Frau Lindy Stephens wäre dieses Buch gar nicht möglich gewesen. Es ist ihr gewidmet. Lindy – ich bitte um Entschuldigung, dass ich so mürrisch war, wenn die verschiedenen Deadlines kamen und gingen. Ich möchte mich auch beim lieben Gillman-Stynes-Clan für seine Unterstützung bedanken – ich bin froh, solch eine tolle Familie zu haben.

Dieses Buch hat stark von all denen profitiert, die freiwillig Zeit und Energie dafür aufgebracht haben, die verschiedenen Entwürfe zu lesen und Vorschläge zu machen. Insbesondere möchte ich Chris O'Dell, Daniel Bryant, Pete Hodgson, Martin Fowler, Stefan Schrass und Derek Hammer danken. Und es gibt noch andere Menschen, die auf die eine oder andere Art und Weise direkt beigetragen haben, daher möchte ich Graham Tackley, Erik Doernenberg, Marcin Zasepa, Michael Feathers, Randy Shoup, Kief Morris, Peter Gillard-Moss, Matt Heath, Steve Freeman, Rene Lengwinat, Sarah Wells, Rhys Evans und Berke Sokhan danken. Finden Sie Fehler in diesem Buch, sind das nicht ihre, sondern meine.

Das Team von O'Reilly hat mich ebenfalls außerordentlich unterstützt, und ich möchte meinen Lektorinnen Eleanor Bru und Alicia Young danken, dazu Christopher Guzikowski, Mary Treseler und Rachel Roumeliotis. Ebenfalls ein großer Dank gebührt Helen Codling und ihren Kollegen überall auf der Welt, die meine Bücher auf alle möglichen Konferenzen mitgenommen haben, Susan Conant, die mich auf dem Weg durch die sich verändernde Welt der Bücher geleitet hat, und Mike Loukides, der den ersten Kontakt mit O'Reilly hergestellt hat. Ich weiß, dass hinter den Kulissen noch viel mehr Menschen geholfen haben, bei denen ich mich ebenfalls bedanken möchte.

Neben denjenigen, die direkt zu diesem Buch beigetragen haben, bedanke ich mich auch bei allen, die auf anderen Wegen dabei geholfen haben, dieses Buch Realität werden zu lassen – ob ihnen das nun bewusst ist oder nicht. Daher möchte ich

mich (ohne besondere Reihenfolge) bedanken bei Martin Kelppmann, Ben Stopford, Charity Majors, Alistair Cockburn, Gregor Hohpe, Bobby Woolf, Eric Evans, Larry Constantine, Leslie Lamport, Edward Yourdon, David Parnas, Mike Bland, David Woods, John Allspaw, Alberto Brandolini, Frederick Brooks, Cindy Sridharan, Dave Farley, Jez Humble, Gene Kim, James Lewis, Nicole Forsgren, Hector Garcia-Molina, Sheep & Cheese, Kenneth Salem, Adrian Colyer, Pat Helland, Kresten Thorup, Henrik Kniberg, Anders Ivarsson, Manuel Pais, Steve Smith, Bernd Rucker, Matthew Skelton, Alexis Richardson, James Governor und Kane Stephens.

Wie das immer in solchen Fällen ist, habe ich bestimmt einige vergessen, die wichtige Beiträge zu diesem Buch geliefert haben. Auch denen möchte ich meinen Dank aussprechen und um Entschuldigung bitten, dass ich sie nicht mit ihrem Namen erwähne. Ich hoffe, Sie können mir verzeihen.

Schließlich werde ich gelegentlich gefragt, mit welchen Tools ich dieses Buch geschrieben habe. Ich habe in AsciiDoc geschrieben und dabei Visual Studio Code zusammen mit dem AsciiDoc-Plug-in von João Pinto genutzt. Das Buch wurde in Git versioniert, und es kam das Atlas-System von O'Reilly zum Einsatz. Größtenteils schrieb ich auf meinem Laptop mit einer externen mechanischen Razer-Tastatur, aber zum Ende hin kam auch oft ein iPad Pro mit Working Copy zum Einsatz, um die letzten paar Dinge abzuschließen. Damit konnte ich auf Reisen schreiben – insbesondere blieb mir eine Fährfahrt zu den Orkneys in Erinnerung, auf der ich über das Refaktorieren von Datenbanken schrieb. Die daraus entstandene Seekrankheit war es wert.

KAPITEL 1

Gerade genug Microservices

> Mann, das ist ganz schön eskaliert!
>
> – Ron Burgundy, *Anchorman – Die Legende von Ron Burgundy*

Bevor wir uns eingehender damit befassen, wie man mit Microservices arbeitet, ist es wichtig, ein gemeinsames Verständnis von der Microservices-Architektur zu erlangen. Ich möchte ein paar Missverständnisse ansprechen, denen ich regelmäßig begegne, aber auch auf Feinheiten hinweisen, die oft übersehen werden. Sie werden diese Grundlagen benötigen, um das Beste auf dem Rest des Buchs herausholen zu können. Daher finden Sie in diesem Kapitel eine Erläuterung von Microservices-Architekturen, es wird kurz ein Blick darauf geworfen, wie sich Microservices entwickelt haben (womit wir logischerweise auch auf Monolithen eingehen müssen), und einige der Vorteile und Herausforderungen bei der Arbeit mit Microservices werden unter die Lupe genommen.

Was sind Microservices?

Microservices sind unabhängig deploybare Services, die rund um eine Businessdomäne modelliert wurden. Sie kommunizieren untereinander über das Netzwerk und bieten als Architektur viele Möglichkeiten, Probleme zu lösen, denen Sie sich gegenübersehen. Damit basiert eine Microservices-Architektur auf vielen zusammenarbeitenden Microservices.

Es handelt sich um einen *Typ* einer serviceorientierten Architektur (SOA), wenn auch mit einer starken Meinung dazu, wo die Servicegrenzen gezogen werden sollten und dass eine unabhängige Deploybarkeit entscheidend ist. Microservices haben zudem den Vorteil, aus Technologiesicht agnostisch zu sein.

Aus technologischer Perspektive stellen Microservices die Businessfähigkeiten bereit, die sie über einen oder mehrere Endpunkte im Netz kapseln. Microservices kommunizieren untereinander über dieses Netzwerk – und machen sich damit zu einem verteilten System. Zudem kapseln sie das Speichern und Einlesen von Daten sowie das Bereitstellen von Daten über wohldefinierte Schnittstellen. Daher werden Datenbanken innerhalb der Servicegrenzen verborgen.

Es gibt dabei manches, was genauer zu betrachten ist, daher wollen wir uns einige dieser Ideen im Detail anschauen.

Unabhängige Deploybarkeit

Unabhängige Deploybarkeit ist die Idee, einen Microservice ändern und in eine Produktivumgebung deployen zu können, ohne dabei andere Services anfassen zu müssen. Wichtiger ist noch, dass es nicht darum geht, es tun zu *können* – es ist der Weg, wie Sie Deployments in Ihrem System *tatsächlich* umsetzen. Dabei handelt es sich um eine Disziplin, die Sie für den allergrößten Teil Ihrer Releases einhalten. Eine einfache Idee, die dennoch in der Ausführung kompliziert ist.

Wenn Sie aus diesem Buch nur eine Sache mitnehmen wollen, dann sollte es diese sein: Stellen Sie sicher, dass Sie das Konzept der unabhängigen Deploybarkeit Ihrer Microservices verstanden haben. Machen Sie es sich zur Gewohnheit, Änderungen an einem einzelnen Microservice in die Produktivumgebung zu bringen, ohne etwas anderes deployen zu müssen. Aus dieser Disziplin folgen viele gute Dinge.

Um eine unabhängige Deploybarkeit zu garantieren, müssen wir sicherstellen, dass unsere Services *lose gekoppelt* sind – mit anderen Worten: Wir müssen einen Service ändern können, ohne etwas anderes ändern zu müssen. Wir brauchen dazu also explizite, wohldefinierte und stabile Verträge zwischen Services. Bei der Implementierung können manche Entscheidungen dazu führen, dass das kompliziert wird – so ist beispielsweise die gemeinsame Verwendung von Datenbanken besonders problematisch. Der Wunsch nach lose gekoppelten Services mit stabilen Schnittstellen bringt unser Denken dazu, nach Servicegrenzen Ausschau zu halten.

Modellierung rund um eine Businessdomäne

Es ist teuer, eine Änderung über eine Prozessgrenze hinweg vorzunehmen. Müssen Sie zwei Services anpassen, um ein Feature bereitzustellen, und das Deployen dieser zwei Änderungen orchestrieren, ist das mehr Arbeit, als die gleiche Änderung in einem einzelnen Service vorzunehmen (oder einem Monolithen). Daraus folgt, dass wir Wege finden wollen, auf denen sichergestellt ist, dass wir serviceübergreifende Änderungen so selten wie möglich vornehmen.

Mit dem gleichen Ansatz wie dem in *Building Microservices* nutzt dieses Buch eine Beispieldomäne und eine Beispielfirma, um bestimmte Konzepte deutlich zu machen, bei denen ich keine realen Vorkommnisse erzählen kann. Die fragliche Firma ist Music Corp – eine große internationale Organisation, die es irgendwie schafft, im Geschäft zu bleiben, obwohl sie sich fast vollständig darauf konzentriert, CDs zu verkaufen.

Wir haben uns dazu entschieden, Music Corp trotz aller Widerstände ins 21. Jahrhundert zu befördern, und dazu gehört auch, die bestehende Systemarchitektur

unter die Lupe zu nehmen. In Abbildung 1-1 sehen wir eine einfache Architektur mit drei Schichten. Wir haben eine webbasierte Benutzeroberfläche, eine Businessschicht (einen Business Layer) in Form eines monolithischen Backends und die Datenablage in einer klassischen Datenbank. Diese Schichten gehören – wie das so üblich ist – verschiedenen Teams.

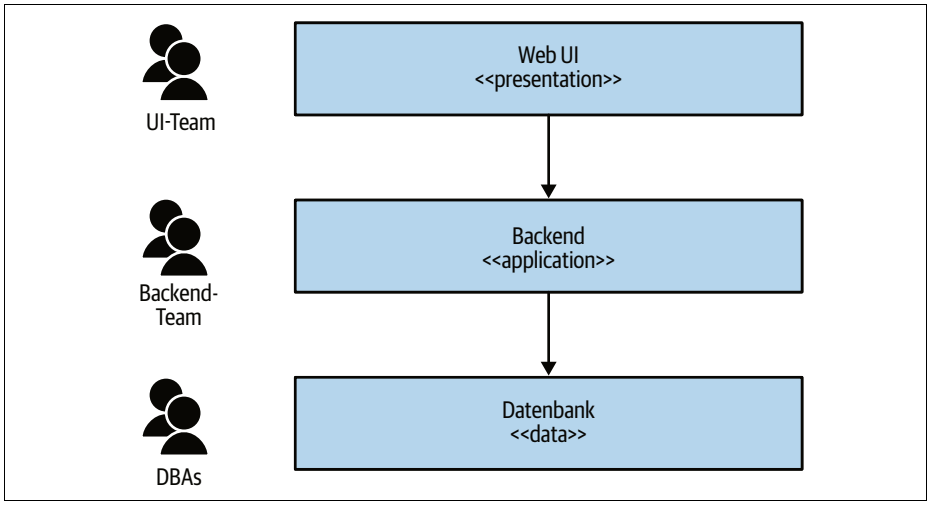


Abbildung 1-1: Die Systeme von Music Corp als klassische Architektur mit drei Schichten

Wir wollen eine einfache Änderung an unserer Funktionalität vornehmen: Unsere Kunden sollen ihr bevorzugtes Musikgenre angeben können. Für diese Änderung müssen wir die Benutzeroberfläche anpassen, um das Genre auswählen zu können, der Backend-Service muss dafür sorgen, dass das Genre im UI erscheinen und die Werte geändert werden können, und die Datenbank muss diese Änderung übernehmen. All diese Anpassungen müssen von den einzelnen Teams gemanagt werden (siehe Abbildung 1-2), und das Ganze muss in der richtigen Reihenfolge geschehen.

Diese Architektur ist gar nicht schlecht. Alle Architekturen sind schließlich auf bestimmte Ziele hin optimiert. Die Drei-Schichten-Architektur ist so verbreitet, weil sie universell ist – jeder hat schon davon gehört. Ein Grund für das häufige Auftreten dieses Patterns ist, dass viele eine Architektur wählen, die ihnen an anderer Stelle bereits begegnet ist. Aber ich denke, der Hauptgrund liegt darin, dass das Muster darauf basiert, wie wir unsere Teams organisieren.

Das mittlerweile berühmte Gesetz von Conway besagt:

> Organisationen, die Systeme entwerfen, [...] sind gezwungen, Entwürfe zu erstellen, die die Kommunikationsstrukturen dieser Organisationen abbilden.
>
> – Melvin Conway, *How Do Committees Invent?*

Die Drei-Schichten-Architektur ist ein gutes Beispiel dafür. In der Vergangenheit haben IT-Organisationen ihre Mitarbeiter*innen anhand ihre Kernkompetenz grup-

piert: Datenbankadministratoren befanden sich in einem Team mit anderen Datenbankadministratoren, Java-Entwickler zusammen mit anderen Java-Entwicklern, und Frontend-Entwickler (die heutzutage so exotische Dinge wie JavaScript und die Entwicklung nativer Mobile-Apps beherrschen) steckten wieder in einem anderen Team. Wir bringen die Leute anhand ihrer Kernkompetenz zusammen, daher erzeugen wir auch IT-Produkte, die zu diesen Teams passen.

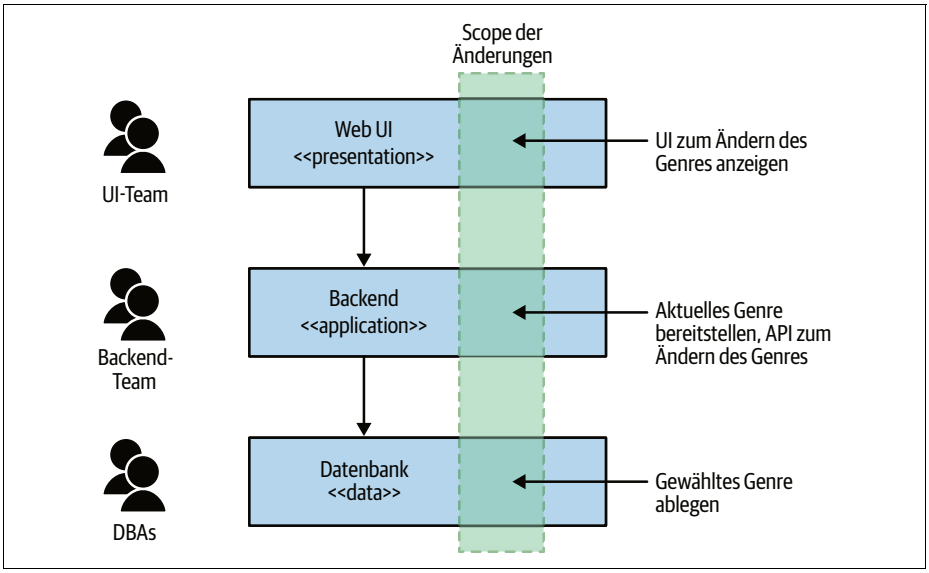


Abbildung 1-2: Eine Änderung über alle drei Schichten ist aufwendiger.

Das erklärt, warum diese Architektur so verbreitet ist. Sie ist nicht schlecht, sondern nur entlang bestimmter Kräfte optimiert – so wie wir traditionell die Leute nach ihren Kenntnissen gruppiert haben. Aber die Kräfte haben sich geändert. Unsere Ansprüche rund um unsere Software haben sich geändert. Wir fassen die Menschen jetzt in fähigkeitsübergreifenden Teams zusammen, um Übergaben und Silos zu reduzieren. Wir wollen Software schneller als je zuvor ausliefern. Das bringt uns dazu, beim Organisieren unserer Teams andere Entscheidungen zu treffen, womit wir auch unsere Systeme anders aufteilen.

Änderungen an der Funktionalität sind meist Änderungen an der Businessfunktionalität. Aber in Abbildung 1-1 ist unsere Businessfunktionalität ineffektiv über alle drei Schichten verteilt, was die Wahrscheinlichkeit erhöht, dass eine Änderung an der Funktionalität schichtübergreifend erfolgen muss. Das ist eine Architektur, in der wir einen engen Zusammenhang verwandter Technologien, aber nur einen losen Zusammenhang der Businessfunktionalität haben. Wollen wir Änderungen vereinfachen, müssen wir das Gruppieren unseres Codes verändern – wir wählen einen engen Zusammenhang der Businessfunktionalität statt der Technologien. Jeder Service kann dann eventuell aus einer Mischung dieser drei Schichten bestehen, aber das ist Sache der lokalen Serviceimplementierung.

Vergleichen wir das mit einer potenziellen alternativen Architektur, die Sie in Abbildung 1-3 sehen. Wir haben einen dedizierten Customer-Service, der ein UI bereitstellt, auf dem die Kunden ihre Informationen aktualisieren können. Der Status des Kunden wird ebenfalls innerhalb dieses Service gespeichert. Die Wahl eines Lieblingsgenres ist mit einem bestimmten Kunden verbunden, daher ist diese Änderung deutlich lokaler. In Abbildung 1-3 sehen Sie auch, dass die Liste der verfügbaren Genres von einem Catalog-Service geholt wird, der vermutlich in der einen oder anderen Form schon vorhanden ist. Ebenfalls zu finden ist dort ein neuer Recommendation-Service, der unser Lieblingsgenre abruft – etwas, das sich leicht in einem Folge-Release umsetzen ließe.

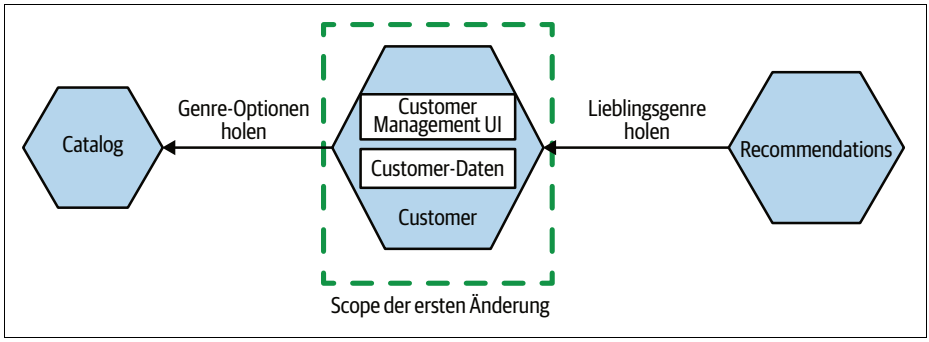


Abbildung 1-3: Ein dedizierter Customer-Service kann es deutlich erleichtern, das bevorzugte Musikgenre eines Kunden zu erfassen.

In solch einer Situation kapselt unser Customer-Service eine dünne Scheibe jeder der drei Schichten – er besitzt ein bisschen UI, ein bisschen Anwendungslogik und ein bisschen Datenablage –, aber diese Schichten sind alle in dem einen Service gekapselt.

Unsere Businessdomäne wird die treibende Kraft unserer Systemarchitektur, wodurch Änderungen hoffentlich einfacher umgesetzt werden können und es uns leichter fällt, unsere Teams rund um unsere Businessdomäne zu organisieren. Das ist so wichtig, dass wir vor dem Ende dieses Kapitels erneut das Konzept des Modellierens von Software rund um eine Domäne betrachten wollen, damit ich ein paar Ideen zum Domain-Driven Design aufzeigen kann, die unser Denken über unsere Microservices-Architektur beeinflussen.

Die eigenen Daten besitzen

Eines der Dinge, mit denen die Menschen meiner Beobachtung nach die größten Probleme haben, ist die Vorstellung, dass Microservices keine gemeinsamen Datenbanken nutzen sollten. Möchte ein Service auf Daten zugreifen, die von einem anderen Service gehalten werden, sollte dieser Service den anderen danach fragen. Damit hat der Service die Möglichkeit, zu entscheiden, was bereitgestellt und was verborgen wird. Es erlaubt ihm auch, interne Implementierungsdetails zu verste-

cken, die sich aus den unterschiedlichsten Gründen ändern können, und einen stabileren öffentlichen Vertrag und damit stabilere Serviceschnittstellen zu ermöglichen. Stabile Schnittstellen zwischen den Services sind sehr wichtig, wenn wir eine unabhängige Deploybarkeit haben wollen – ändert sich die von einem Service bereitgestellte Schnittstelle immer wieder, wird das einen Dominoeffekt verursachen, durch den sich auch andere Services ändern müssen.

Nutzen Sie keine gemeinsamen Datenbanken, sofern Sie das nicht müssen. Und selbst dann versuchen Sie, das so weit wie möglich zu vermeiden. Meiner Meinung nach sind gemeinsame Datenbanken das Schlimmste, was Sie tun können, wenn Sie versuchen, eine unabhängige Deploybarkeit zu erreichen.

Wie wir schon im vorherigen Abschnitt besprochen haben, wollen wir unsere Services als End-to-End-Scheiben der Businessfunktionalität betrachten, die UI, Anwendungslogik und Datenablage sauber kapseln. Denn wir wollen den Aufwand verringern, der notwendig ist, um businessbezogene Funktionalität zu verändern. Das so vorgenommene Kapseln von Daten und Verhalten sorgt für einen engen Zusammenhalt der Businessfunktionalität. Indem wir die Datenbank verbergen, die unseren Service unterstützt, stellen wir auch sicher, dass wir Kopplungen reduzieren. Zu Kopplungen und Zusammenhalt kommen wir gleich noch mal zurück.

Es kann schwer sein, sich das verständlich zu machen, besonders wenn Sie ein bestehendes monolithisches System mit einer riesigen Datenbank vor sich haben. Zum Glück gibt es Kapitel 4, das sich nur darum dreht, von monolithischen Datenbanken wegzukommen.

Welche Vorteile können Microservices haben?

Es gibt eine ganze Reihe unterschiedlicher Vorteile von Microservices. Die unabhängige Natur der Deployments eröffnet ganz neue Modelle für das Verbessern der Größe und Robustheit von Systemen und ermöglicht es Ihnen, Technologien sehr gemischt einzusetzen. Da an Services parallel gearbeitet werden kann, können Sie mehr Entwickler an ein Problem setzen, ohne dass sie sich in die Quere kommen. Es kann für diese Entwickler auch einfacher sein, ihren Teil des Systems zu verstehen, da sie ihre Aufmerksamkeit ganz auf diesen einen Teil richten können. Prozessisolierung ermöglicht uns zudem, verschiedene Technologien zu wählen, vielleicht unterschiedliche Programmiersprachen, Programmierstile, Deployment-Plattformen oder Datenbanken zu mischen, um den richtigen Mix zu finden.

Außerdem bieten Ihnen Microservices-Architekturen vor allem Flexibilität. Sie eröffnen Ihnen so viel mehr Möglichkeiten zum Lösen zukünftiger Probleme.

Aber es ist wichtig, darauf hinzuweisen, dass all diese Vorteile ihren Preis haben. Es gibt viele Wege, Ihr System auseinanderzunehmen, und Ihre Ziele sind dabei entscheidend dafür, wie Sie das umsetzen. Daher ist es wichtig, zu verstehen, was Sie mit Ihrer Microservices-Architektur erreichen wollen.

Welche Probleme werden entstehen?

Serviceorientierte Architekturen wurden unter anderem deshalb so beliebt, weil Computer billiger wurden und wir mehr davon hatten. Statt Systeme auf einzelnen, riesigen Mainframes zu deployen, war es sinnvoll, mehrere billigere Maschinen einzusetzen. Serviceorientierte Architekturen waren ein Versuch, herauszufinden, wie sich Anwendungen am besten bauen lassen, die über mehrere Maschinen verteilt sind. Eine der größten Herausforderungen ist dabei der Weg, auf dem diese Computer miteinander reden: die Netzwerke.

Die Kommunikation zwischen Computern über Netzwerke geschieht nicht instantan (das hat offensichtlich etwas mit Physik zu tun). Wir müssen uns also mit Latenzen befassen – insbesondere mit Latenzen, die weit über das hinausgehen, was wir bei lokalen In-Process-Operationen gewohnt sind. Es wird noch schlimmer, wenn wir daran denken, dass diese Latenzen variieren können, wodurch das Systemverhalten unvorhersagbar werden kann. Und wir müssen berücksichtigen, dass Netzwerke manchmal fehlerhaft sein können – Pakete gehen verloren, Netzwerkkabel werden abgezogen und so weiter.

Diese Herausforderungen sorgen dafür, dass Aktivitäten viel schwieriger werden können, die bei einem Monolithen mit jeweils einem Prozess recht einfach sind – wie zum Beispiel Transaktionen –, und zwar so schwierig, dass Sie Transaktionen (und deren Sicherheit) mit wachsender Komplexität Ihres Systems vermutlich irgendwann hinauswerfen müssen, um auf andere Techniken umzusteigen (die leider wieder ganz andere Nachteile besitzen).

Darüber nachzudenken, dass jedes Netzwerk fehlerhaft agieren kann und wird, dass der Service, mit dem Sie kommunizieren, aus irgendwelchen Gründen offline gehen oder dass er sich komisch verhalten kann, wird Ihnen Kopfschmerzen bereiten. Außerdem müssen Sie sich darüber im Klaren werden, wie Sie eine konsistente Sichtweise auf Daten erhalten, die auf mehrere Maschinen verteilt sind.

Und dann haben wir natürlich noch ein ganzes Füllhorn an neuen Microservices-orientierten Technologien zu berücksichtigen – neuen Technologien, die, falsch eingesetzt, dazu führen können, dass Sie viel schneller viel interessantere, teurere Fehler machen. Ehrlich gesagt, scheinen Microservices eine echt dumme Idee zu sein – wären da nicht die ganzen Vorteile.

Es sei darauf hingewiesen, dass so gut wie alle Systeme, die wir als »Monolithen« betrachten, ebenfalls verteilte Systeme sind. Eine Ein-Prozess-Anwendung liest sehr wahrscheinlich Daten aus einer Datenbank, die auf einem anderen Rechner läuft, und übergibt Daten an einen Webbrowser. Das sind schon mindestens drei Rechner, die untereinander über das Netzwerk kommunizieren. Der Unterschied liegt darin, in welchem Umfang monolithische Systeme im Vergleich zu Microservices-Architekturen verteilt sind. Wenn Sie mehr Computer in diesem Spiel im Einsatz haben und mehr Kommunikation über mehr Netzwerke stattfindet, werden

Sie auch eher schon auf die unerfreulichen Probleme stoßen, die mit verteilten Systemen verbunden sind. Diese schon kurz angerissenen Probleme müssen nicht sofort auftauchen, aber mit der Zeit (und einem wachsenden System) werden Sie vermutlich über die meisten, wenn nicht alle, stolpern.

Wie mein früherer Kollege, Freund, Gefährte und Microservices-Experte James Lewis gesagt hat: »Microservices erkaufen Ihnen Möglichkeiten.« James hat seine Worte bewusst gewählt – sie *erkaufen* Ihnen *Möglichkeiten*. Es gibt Microservices nicht umsonst, und Sie müssen entscheiden, ob die Möglichkeiten die Kosten wert sind. Dieses Thema werden wir detaillierter in Kapitel 2 angehen.

Benutzeroberflächen

Allzu häufig sehe ich, dass sich Menschen bei ihrer Arbeit darauf konzentrieren, Microservices rein auf der Serverseite anzuwenden – und die Benutzeroberfläche als eine einzelne monolithische Schicht zu belassen. Wollen wir eine Architektur haben, die es uns erleichtert, neue Features schneller zu deployen, kann es ein großer Fehler sein, das UI als monolithischen Klumpen zu belassen. Wir können – und sollten – auch unsere Benutzeroberflächen aufbrechen. Darum kümmern wir uns in Kapitel 3.

Technologie

Es kann verlockend sein, sich haufenweise neue Technologien zu schnappen und sie zusammen mit Ihrer schicken neuen Microservices-Architektur einzusetzen, aber ich empfehle Ihnen dringend, dieser Verlockung zu widerstehen. Es kostet immer etwas, neue Technologien einzusetzen – sie führen zu Umbruch und Unruhe. Hoffentlich ist es das wert (wenn Sie die richtige Technologie eingesetzt haben, ist es das mit Sicherheit!), aber wenn Sie eine Microservices-Architektur das erste Mal übernehmen, haben Sie auch so schon genug zu tun.

Um herauszufinden, wie Sie eine Microservices-Architektur sauber entwickeln und betreuen, müssen Sie eine Vielzahl an Herausforderungen rund um verteilte Systeme meistern – Herausforderungen, denen Sie zuvor vielleicht noch nie begegnet sind. Ich denke, es ist viel sinnvoller, sich damit zu befassen, wenn Sie auf sie stoßen, während Sie einen Technologie-Stack einsetzen, der Ihnen vertraut ist. Dann können Sie sich immer noch Gedanken darüber machen, ob es sinnvoll ist, Ihre bestehende Technologie auszutauschen, um die Probleme zu lösen.

Wie wir schon erkannt haben, sind Microservices im Prinzip technologieagnostisch. Solange Ihre Services miteinander über ein Netzwerk kommunizieren können, ist der Rest nicht so wichtig. Das kann ein großer Vorteil sein – Sie können so Technologie-Stacks ganz nach Belieben mischen.

Sie müssen Kubernetes, Docker, Container oder eine öffentliche Cloud nicht einsetzen. Sie müssen nicht in Go, Rust oder was auch immer programmieren. Tat-

sächlich ist die Wahl Ihrer Programmiersprache in Bezug auf Microservices-Architekturen ziemlich unwichtig, abgesehen davon, dass ein paar der Sprachen ein umfangreicheres Ökosystem aus unterstützenden Bibliotheken und Frameworks mitbringen. Wenn Sie sich in PHP am besten auskennen, dann beginnen Sie in PHP![1] Es gibt da draußen viel zu viel technischen Snobismus in Bezug auf bestimmte Technologie-Stacks, der oft leider zu einer Verachtung derjenigen führt, die mit anderen Tools arbeiten.[2] Seien Sie nicht Teil des Problems! Wählen Sie einen Ansatz, der für Sie funktioniert, und ändern Sie Dinge, um Probleme anzugehen, wenn Sie auf sie stoßen.

Größe

»Wie groß sollte ein Microservice werden?«, das ist vermutlich die Frage, die mir am häufigsten gestellt wird. Das ist nicht überraschend, steckt doch der Begriff »Micro« im Namen. Aber wenn Sie verstanden haben, wie Microservices als Architektur funktionieren, ist das Konzept der Größe tatsächlich einer der uninteressantesten Punkte.

Wie messen Sie die Größe? Ist es die Anzahl der Codezeilen? Das klingt nicht so sinnvoll. Es kann sein, dass jemand 25 Zeilen Java-Code benötigt, die auch in 10 Zeilen Clojure geschrieben werden könnten. Das heißt nicht, dass Clojure besser oder schlechter als Java ist, sondern nur, dass manche Sprachen expressiver als andere sind.

Was »Größe« meiner Meinung nach am nächsten kommt, ist etwas, das der Microservices-Experte Chris Richardson einmal so beschrieben hat: Das Ziel von Microservices sei es, eine »möglichst kleine Schnittstelle zu haben«. Das passt zum Konzept des *Information Hiding* (auf das wir noch kommen werden), steht aber eher für einen Versuch, im Nachhinein noch eine Bedeutung dafür zu finden – als wir darüber erstmals sprachen, lag unser Hauptaugenmerk zumindest zu Beginn vor allem darauf, diese Dinge sehr einfach ersetzen zu können.

Letztendlich ist das Konzept der Größe sehr kontextabhängig. Sprechen Sie mit jemandem, der seit 15 Jahren mit einem System arbeitet, wird er die 100.000 Zeilen Code leicht verständlich finden. Fragen Sie jemand anderen, der bei dem Projekt ganz frisch dabei ist, wird er es als viel zu groß ansehen. Oder fragen Sie eine Firma, die gerade mit der Umwandlung in Microservices begonnen hat und bei der vielleicht zehn oder weniger Microservices im Einsatz sind, erhalten Sie eine andere Antwort als bei einer Firma gleicher Größe, die seit Jahren mit Microservices arbeitet und nun Hunderte davon nutzt.

1 Mehr dazu finden Sie in *PHP Web Services* von Lorna Jane Mitchell (O'Reilly).

2 Nachdem ich Aurynn Shaws Blogpost »Contempt Culture« gelesen habe (*http://bit.ly/2oeICgL*), erkannte ich, dass ich in der Vergangenheit auch schon eine gewisse Verachtung für verschiedene Technologien und die entsprechenden Communitys gezeigt habe.

Ich sage den Leuten immer, sie sollten sich nicht allzu viele Gedanken um die Größe machen. Wenn Sie mit dem ganzen Thema beginnen, ist es viel wichtiger, sich auf zwei zentrale Aspekte zu konzentrieren. Erstens: Mit wie vielen Microservices können Sie umgehen? Mit einer zunehmenden Zahl wird die Komplexität Ihres Systems wachsen, und Sie werden neue Fertigkeiten erlernen (und eventuell neue Technologien einsetzen) müssen, um das Ganze im Griff zu behalten. Aus diesem Grund rate ich deutlich dazu, schrittweise zu einer Microservices-Architektur zu migrieren. Zweitens: Wie definieren Sie die Grenzen Ihrer Microservices, um das Beste aus ihnen herauszuholen, ohne das Ganze in ein furchtbares Chaos ausarten zu lassen? Das sind die Themen, mit denen wir uns im Rest dieses Kapitels befassen wollen.

Die Geschichte des Begriffs »Microservices«

Als ich im Jahr 2011 noch bei einer Consulting-Firma namens ThoughtWorks arbeitete, interessierte sich mein Freund und damaliger Kollege James Lewis für etwas, das er als »Micro-Apps« bezeichnete. Er hatte dieses Pattern bei ein paar Firmen beobachtet, die eine serviceorientierte Architektur verfolgten – sie optimierten diese Architektur, um Services leicht ersetzen zu können. Die fraglichen Firmen waren daran interessiert, bestimmte Funktionalität schnell deployt zu bekommen, sie aber gleichzeitig auch komplett mit anderen Technologie-Stacks neu schreiben zu können, wenn die zu bedienenden Mengen wuchsen.

Damals war besonders beachtenswert, wie klein diese Services bezüglich ihres Einsatzbereichs waren. Einige der Services konnten in wenigen Tagen geschrieben (oder umgeschrieben) werden. James sagte gern: »Services sollten nicht größer als mein Kopf sein.« Die Idee war, dass der Scope der Funktionalität leicht verständlich und damit auch leicht änderbar sein sollte.

2012 präsentierte James diese Ideen bei einer Architekturkonferenz, auf der ein paar von uns anwesend waren. Bei dieser Session diskutierten wir darüber, dass diese Dinge eigentlich keine vollständigen Anwendungen wären und »Micro-Apps« nicht passte. Stattdessen schien »Microservices« ein besserer Begriff dafür zu sein.[3]

3 Ich weiß nicht mehr, wann wir den Begriff wirklich das erste Mal aufgeschrieben haben, aber ich erinnere mich noch genau daran, dass ich entgegen aller grammatikalischen Logik darauf bestand, den Begriff *ohne* Bindestrich zu schreiben. Im Nachhinein ließ sich diese Position nur schlecht begründen, aber ich blieb dabei. Ich blieb bei meiner unsinnigen, aber letztendlich erfolgreichen Wahl.

Und Ownership

Sind Microservices rund um eine Businessdomäne modelliert, sehen wir Übereinstimmungen zwischen unseren IT-Artefakten (den unabhängig deploybaren Microservices) und unserer Businessdomäne. Diese Idee hallt bei Technologiefirmen wider, die die Grenzen zwischen »Business« und »IT« niederreißen. In klassischen IT-Organisationen geschieht die Softwareentwicklung oft völlig getrennt vom Business, das die Anforderungen definiert und eine Verbindung zum Kunden hat (siehe Abbildung 1-4). Die Dysfunktionen dieser Art von Organisationen sind zahlreich und vielfältig, und wir müssen hier vermutlich nicht weiter darauf eingehen.

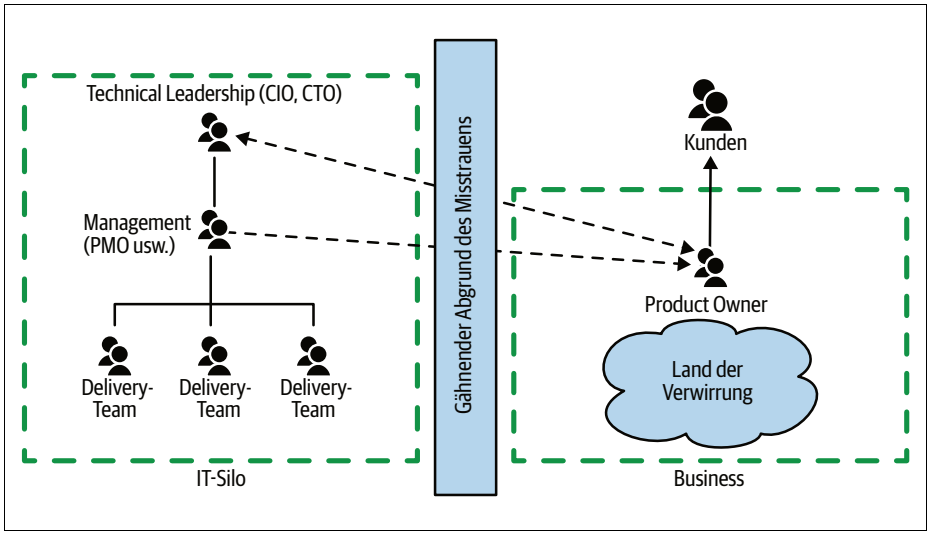


Abbildung 1-4: Eine Organisationssicht auf die klassische IT-Business-Aufteilung

Stattdessen sehen wir echte Technologieorganisationen, die diese zuvor getrennten organisatorischen Silos vereinen (siehe Abbildung 1-5). Product Owner arbeiten jetzt direkt als Teil der Delivery-Teams mit, die wiederum nicht mehr technischen Gruppen, sondern entsprechenden kundenzentrierten Produktlinien zugeordnet sind. Statt dass zentralisierte IT-Funktionen die Norm sind, ist das Ziel jeder zentralen IT-Funktion, diese kundenzentrierten Delivery-Teams zu unterstützen.

Auch wenn nicht alle Organisationen diesen Wechsel vollzogen haben, sorgen Microservices-Architekturen dafür, dass er leichter umzusetzen ist. Wollen Sie mit den Produktlinien abgestimmte Delivery-Teams und mit der Businessdomäne abgestimmte Services haben, wird es einfacher, die Ownership ganz klar diesen produktorientierten Delivery-Teams zuzuweisen. Es ist wichtig, immer weniger von mehreren Teams betreute Services zu haben, um Ärger bei der Auslieferung zu vermeiden – Microservices-Architekturen, die an Businessdomänen orientiert sind, erleichtern diesen Wechsel der Organisationsstrukturen.

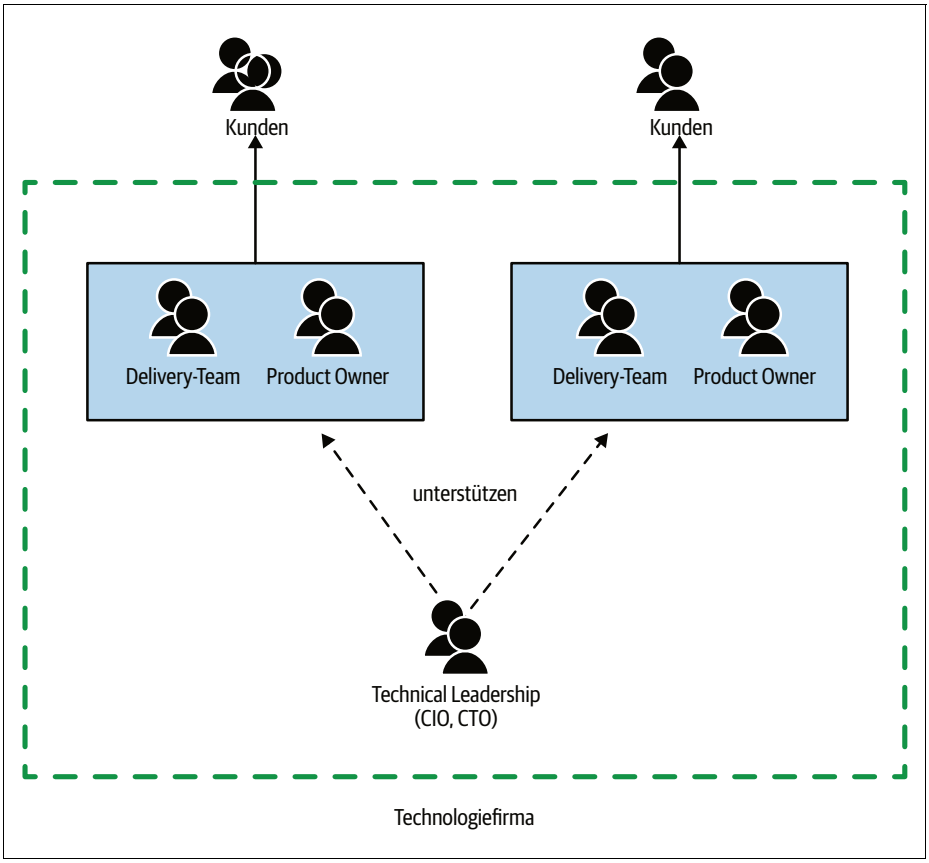


Abbildung 1-5: Ein Beispiel dafür, wie echte Technologiefirmen Software-Delivery integrieren

Der Monolith

Wir haben über Microservices gesprochen, aber in diesem Buch geht es darum, *von* Monolithen *zu* Microservices zu migrieren. Daher müssen wir festlegen, was wir mit dem Begriff *Monolith* meinen.

Rede ich in diesem Buch über die Monolithen, beziehe ich mich vor allem auf eine Deployment-Einheit. Muss die gesamte Funktionalität eines Systems gemeinsam deployt werden, betrachten wir es als einen Monolithen. Es gibt mindestens drei monolithische Systeme, die in dieses Schema passen: das Ein-Prozess-System, der verteilte Monolith und Black-Box-Systeme von Fremdherstellern.

Der Ein-Prozess-Monolith

Das Beispiel, das einem bei Gesprächen über Monolithen am häufigsten in den Sinn kommt, ist das eines Systems, bei dem der gesamte Code als *ein Prozess* deployt wird

(siehe Abbildung 1-6). Vielleicht haben Sie mehrere Instanzen dieses Prozesses, damit das System robuster ist oder besser skaliert, aber im Prinzip befindet sich der gesamte Code in einem einzelnen Prozess. In der Realität kann es sich bei diesen Ein-Prozess-Systemen um einfache verteilte Systeme handeln, da sie so gut wie immer Daten aus einer Datenbank lesen oder in diese schreiben.

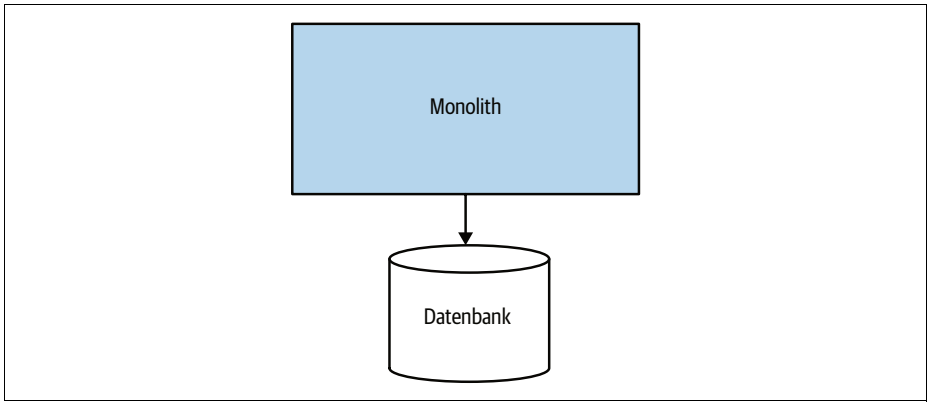


Abbildung 1-6: Ein Ein-Prozess-Monolith: Der gesamte Code befindet sich in einem einzelnen Prozess.

Diese Ein-Prozess-Monolithen repräsentieren vermutlich die große Mehrheit der monolithischen Systeme, mit denen die Menschen meiner Beobachtung nach hadern, daher werden wir uns vor allem darum kümmern. Nutze ich ab jetzt den Begriff »Monolith«, rede ich über diese Art von Monolithen, es sei denn, ich erwähne bewusst einen anderen Typ.

Und der modulare Monolith

Als Untermenge der Ein-Prozess-Monolithen handelt es sich beim *modularen Monolithen* um eine Abwandlung: Der eine Prozess besteht aus separaten Modulen, an denen jeweils unabhängig voneinander gearbeitet werden kann. Zum Deployen müssen aber alle wieder kombiniert werden (siehe Abbildung 1-7). Das Konzept, Software in Module aufzuteilen, ist nicht neu – wir werden auf seine Geschichte weiter unten in diesem Kapitel noch zu sprechen kommen.

Der modulare Monolith kann für viele Organisationen eine ausgezeichnete Wahl sein. Sind die Modulgrenzen gut definiert, wird damit ein hoher Grad an Parallelität beim Arbeiten ermöglicht, während gleichzeitig die Herausforderungen der verteilteren Microservices-Architektur durch viel einfachere Deployment-Überlegungen ersetzt werden können. Shopify ist ein gutes Beispiel einer Organisation, die diese Technik als Alternative zur Microservices-Aufteilung eingesetzt hat – und es scheint für sie sehr gut zu funktionieren.[4]

4 Kirsten Westeinde erzählt auf YouTube (*http://bit.ly/2oauZ29*) von den Überlegungen, die dazu führten, dass Shopify einem modularen Monolithen gegenüber Microservices den Vorzug gegeben hat.