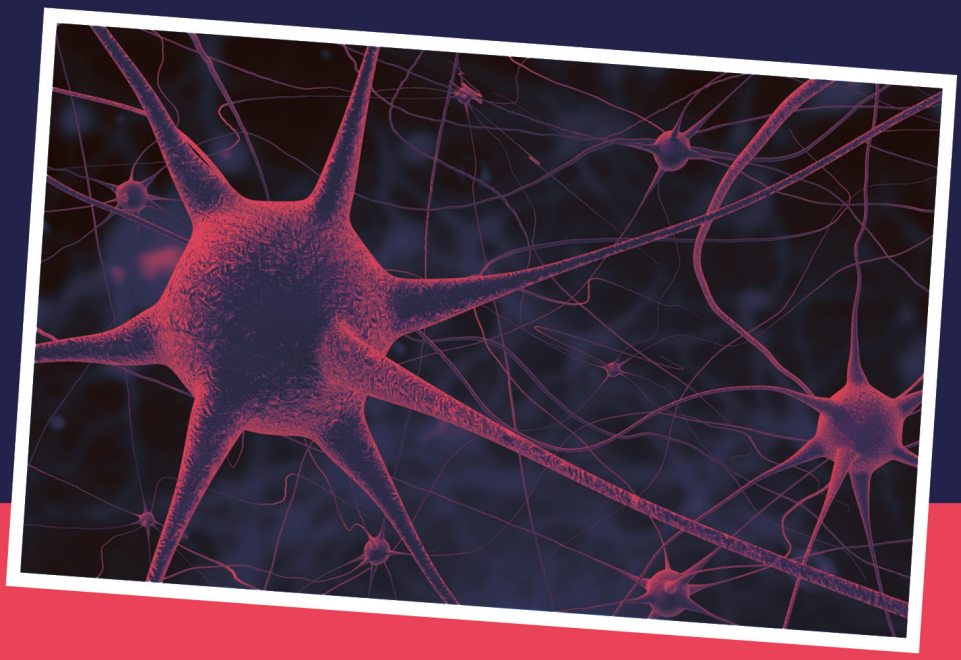

Eberhard Wolff

Microservices

Ein Überblick

INNOQ

Microservices - Ein Überblick

Eberhard Wolff

Dieses Buch wird verkauft unter
http://leanpub.com/microservices-ueberblick

Diese Version wurde veröffentlicht am 2020-09-30

Inhaltsverzeichnis

1 Vorwort

Microservices sind mittlerweile ein Hype. Wie jeder Architektur-Ansatz haben Microservices einen bestimmten Einsatzkontext, in dem sie ihre Vorteile voll ausspielen können. Die Vorteile erkaufen sie mit Herausforderungen in anderen Bereichen.

Diese Broschüre gibt eine kurze Einführung in das Thema Microservices und stellt dar, was Microservices sind und welche Vorteile sie haben. Das erleichtert einen Einstieg in das Thema und hilft, die Einsetzbarkeit und den Nutzen von Microservices in einem bestimmten Kontext abzuschätzen. So werden Microservices vom Hype zum sinnvollen Bestandteil im Werkzeugkasten eines Architekten.

2 Was sind Microservices?

Die Idee der Microservices[1] ist nicht neu. Einen ganz ähnlichen Ansatz verfolgt schon die UNIX Philosophie. Sie basiert auf drei Ideen:

- Ein Programm soll nur eine Aufgabe erfüllen — die dafür sehr gut.
- Die Programme sollen zusammenarbeiten können.
- Außerdem sollen die Programme eine universelle Schnittstelle nutzen. In UNIX sind das Textströme.

Dadurch entstehen wiederverwendbare Programme - letztendlich eine Art Komponenten.

Microservices dienen dazu, große Systeme aufzuteilen. Damit sind Microservices ein Modularisierungskonzept. Es gibt viele Modularisierungskonzepte, aber Microservices sind anders. Sie können unabhängig voneinander in Produktion gebracht werden. Eine Änderung an einem Microservice bedeuten, dass nur dieser Microservices in Produktion gebracht werden muss. Bei anderen Modularisierungskonzepten müssen alle Module gemeinsam ausgeliefert werden. Eine Änderung an einem Modul erfordert dann ein erneutes Deployment der gesamten Anwendung mit allen Modulen.

Microservice = virtuelle Maschine

Microservices können nicht mit den Modularisierungskonzepten der Programmiersprachen umgesetzt werden. Denn diese Konzepte

[1] Eberhard Wolff: Microservices - Grundlagen flexibler Softwarearchitekturen, dpunkt Verlag, 2015, ISBN 978-3864903137

erfordern, dass alle Module zusammen in einem Programm ausgeliefert werden. Stattdessen müssen Microservices als virtuelle Maschinen, leichtgewichtigere Alternativen wie Docker-Container oder einzelne Prozesse umgesetzt werden. Nur dann können sie einzeln in Produktion gebracht werden.

Daraus ergeben sich weitere Vorteile: So sind Microservices nicht an eine bestimmte Technologie gebunden. Sie können in jeder Programmiersprache oder mit jeder Plattform implementiert sein. Und natürlich können Microservices eigene Unterstützungsdienste wie Datenbanken oder andere Infrastruktur mitbringen.

Microservice sollen getrennte Datenhaushalte haben. Jeder Microservice ist also Herr über seine Daten. Die Erfahrung lehrt, dass die gemeinsame Nutzung von Datenbank-Schemata Änderungen an den Datenstrukturen praktisch unmöglich machen. Das behindert die Änderbarkeit von Software so stark, dass Microservices diese Art der Kopplung ausschliessen.

Kommunikation zwischen Microservices

Microservices müssen miteinander kommunizieren können. Dazu sind unterschiedliche Ansätze möglich:

- Die Microservices können Daten *replizieren*. Gemeint ist damit nicht das Kopieren der Daten ohne Änderung des Schemas. Dann sind Änderungen an dem Schema unmöglich, weil mehrere Microservices dasselbe Schema nutzen. Wenn aber ein Microservice Bestellungen verwaltet und ein anderer die Daten der Bestellungen analysiert, sind Datenformate und Zugriffe unterschiedlich: Die Analyse liest die Daten vor allem, für die Verwaltung der Bestellungen sind Lesen und Schreiben eher gleichberechtigt. Auch klassische Datawarehouses arbeiten mit Replikation für die Analyse großer Datenmengen.
- Wenn Microservices eine HTML-UI haben, können sie *Links* auf anderen Microservices nutzen. Es ist außerdem möglich,

dass ein Microservice HTML-Code anderer Microservices integriert.
- Schließlich können die Microservices mit Protokollen wie *REST* oder *Messaging* über das Netzwerk miteinander kommunizieren.

In einem Microservice-System muss definiert werden, welche Kommunikationsvarianten genutzt werden, um zu garantieren, dass die Microservices auch mit diesen Technologien erreicht werden könne.

2.1 Größe

Der Begriff "Microservice" stellt die Größe eines Microservices in den Mittelpunkt. Das ist zur Abgrenzung gegenüber anderen Service-Begriffen auch sinnvoll. Aber die konkrete Größe eines Microservices anzugeben, ist gar nicht so einfach. Schon die Einheit ist ein Problem: Lines of Code (LoC, Codezeilen) sind keine gute Einheit. Schließlich hängt die Anzahl der Zeilen eines Programms nicht nur von der Formatierung, sondern auch von der Programmiersprache ab. Überhaupt ist es wenig sinnvoll, einen Architekturansatz mit solchen Maßzahlen zu bewerten. Schließlich kann die Größe eines Systems kaum sinnvoll absolut angegeben werden, sondern nur im Verhältnis zu den abgebildeten Geschäftsprozessen und ihrer Komplexität.

Daher ist es besser, die Größe eines Microservices anhand von oberen und unteren Grenzen zu definieren. Für Microservices gilt eigentlich, dass kleiner besser ist:

- Ein Microservice sollte von einem *Team* weiterentwickelt werden. Daher darf ein Microservice auf keinen Fall so groß sein, dass mehr als ein Team an ihm entwickeln muss.

- Microservices sind ein Modularisierungsansatz. Entwickler sollten einzelne *Module* verstehen können - daher müssen Module und damit Microservices so klein sein, dass sie ein Entwickler noch verstehen kann.
- Schließlich soll ein Microservice *ersetzbar* sein. Wenn der Microservice nicht mehr wartbar ist oder eine leistungsfähigere Technologie genutzt werden soll, kann der Microservice durch eine neue Implementierung ausgetauscht werden. Microservices sind der einzige Ansatz, der bereits bei der Entwicklung die Ablösung des Systems oder zumindest von Teilen des Systems betrachtet.

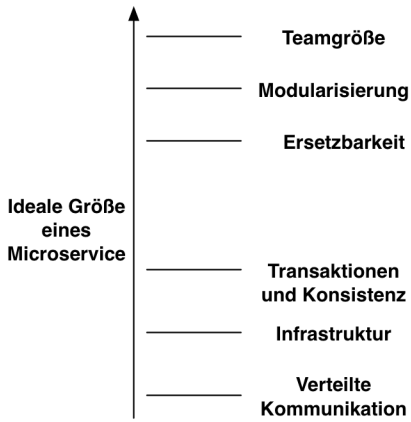

Abb. 1: Die ideale Größe von Microservices

Die Frage ist , warum man Microservices nicht möglichst klein baut. Schließlich verstärken sich die Vorteile, wenn die Microservices besonders klein sind. Aber es gibt Gründe, warum sehr kleine Microservices nicht so ohne weitere möglich sind:

- *Verteilte Kommunikation* zwischen Microservices über das Netz ist aufwändig. Wenn die Microservices größer sind, ist die Kommunikation eher lokal in einem Microservice und damit weniger aufwändig.

- Das *Verschieben von Logik* über die Grenzen von Microservices hinweg ist schwierig. Der Code muss in ein anderes System bewegt werden. Wenn das System eine andere Technologie oder Programmiersprache nutzt, kann eine Neuimplementierung des Codes die einzige Möglichkeit sein, um Funktionalität in den anderen Microservice zu bringen. Natürlich ist es auch immer möglich, aus den Funktionalitäten einen eigenen Microservice zu machen, auf den dann die anderen Microservices zugreifen können. Innerhalb eines Microservices ist Refactoring mit den üblichen Mechanismen recht einfach umsetzbar.
- Eine *Transaktion* in einem Microservice ist einfach umsetzbar. Über die Grenzen eines Microservice hinaus ist das nicht mehr so einfach, weil dazu verteilte Transaktionen notwendig sind. Es bietet sich also an, die Größe eines Microservice so zu wählen, dass eine Transaktion vollständig in einem Microservice abgearbeitet wird.
- Ähnliches gilt für die *fachliche Konsistenz* der Daten: Wenn beispielsweise der Kontostand mit dem Ergebnis aus den Einnahmen und Ausgaben übereinstimmen soll, ist das in einem Microservice einfach umsetzbar, aber über Microservices hinweg kaum praktikabel. Daher sollten Microservices also so groß sein, dass Daten, die konsistent sein müssen, im selben Microservice verwaltet werden.
- Jeder Microservice muss unabhängig in Produktion gebracht werden und daher eine *eigene Umgebung* haben. Das verbraucht Hardware-Ressourcen und bedeutet außerdem, dass der Aufwand für die Administration des Systems steigt. Wenn es größere und damit weniger Microservices gibt, sinkt dieser Aufwand.

Bis zu einem gewissen Maße hängt die Größe eines Microservice von der Infrastruktur ab: Wenn die Infrastruktur einfach ist, kann sie sehr viele und damit auch sehr kleine Microservices unterstützen. Die Vorteile der Microservice-Architektur sind dann

dementsprechend größer. Schon einfache Maßnahmen können den Aufwand bei der Infrastruktur reduzieren: Wenn es Templates für Microservices gibt oder andere Möglichkeiten, um Microservices einfacher zu erstellen und einheitlich zu verwalten, kann das den Aufwand reduzieren und so kleinere Microservices möglich machen.

Nanoservices

Bestimmte technologische Ansätze können die Größe eines Service weiter reduzieren. Statt einen Microservices als virtuelle Maschine oder Docker-Container auszuliefern, können die Services Anwendungen in einem Java-EE-Application-Server sein. *Java EE* definiert verschiedene Deployment-Formate und kann in einem Application Server mehrere Anwendungen laufen lassen. Die Services kommunizieren dann genauso wie Microservices beispielsweise über REST oder Messaging. Dann sind die Services aber nicht mehr so gut gegeneinander isoliert: Wenn eine Anwendung in einem Application Server viel Speicher verbraucht, zieht das die anderen Anwendungen auf dem Application Server in Mitleidenschaft.

Eine andere Alternative sind OSGi-Bundles. Auch dieser Ansatz definiert ein Modul-System auf Basis von Java. Im Gegensatz zu Java EE erlaubt *OSGi* aber Methodenaufrufe zwischen Bundles, so dass eine Kommunikation über REST oder Messaging nicht zwingend ist.

Leider sind beide Ansätze beim unabhängigen Deployment problematisch: In der Praxis müssen Java-EE-Application-Server und OSGi-Laufzeitumgebungen beim Deployment neuer Module oft neu gestartet werden. Also beeinflusst ein Deployment auch andere Module und eine wesentliche Eigenschaft der Microservices ist nicht mehr erfüllt.

Dafür sinkt der Aufwand bei der Infrastruktur und auch der Kommunikationsaufwand, weil beispielsweise bei OSGi lokale Methodenaufrufe genutzt werden können. Das erlaubt kleinere Services.

Um solche Services klar von Microservices abzugrenzen, ist es sinnvoll, einen alternativen Begriff wie „Nanoservices“ für diesen Ansatz zu nutzen. Schließlich bieten sie weder die Isolation von Microservices noch das unabhängige Deployment

2.2 Bounded Context und Domain-Driven Design

Ein Ziel von Microservices ist es, fachliche Änderungen auf einen Microservice zu begrenzen. Fachliche Änderungen können die UI umfassen. Daher sollte ein Microservice auch UI-Elemente integrieren. Aber auch in einem anderen Bereich sollten Änderungen im selben Microservice erfolgen – nämlich bei den Daten.

Ein Dienst, der einen Bestellprozess implementiert, sollte auch die Daten für eine Bestellung verwalten und ändern können. Microservices haben einen eigenen Datenhaushalt und können daher Daten passend speichern. Aber ein Bestellprozess benötigt mehr als nur die Daten der Bestellung. Auch die Daten des Kunden oder der Waren sind für den Bestellprozess relevant.

An dieser Stelle ist Domain-Driven Design [2] (DDD) hilfreich. Domain-Driven Design dient zur Analyse einer fachlichen Domäne. Wesentliche Grundlage ist *Ubiquituous Language*. Das ist wie andere Bestandteile von Domain Driven Design auch ein Pattern und ist daher in *kursiv* gesetzt. *Ubiquituous Language* besagt, dass an der Software Beteiligten dieselben Begriffe nutzen sollen. Fachbegriffe wie Bestellung, Rechnung usw. sollen sich direkt in der Software wiederfinden. Oft ergeben sich in einem Unternehmen eine ganz eigene Sprache. Ggenau die sollte dann auch in der Software so umgesetzt werden.

Das Domänenmodell kann aus verschiedenen Elementen bestehen:

[2] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley,2003, ISBN 978-0-32112-521-7

- *Entity* ist ein Objekt mit einer eigenen Identität. In einer E-Commerce-Anwendung kann der Kunde oder die Ware eine Entity sein. *Entities* werden typischerweise in einer Datenbank gespeichert.
- *Value Objects* haben keine eigene Identität. Ein Beispiel ist eine Adresse sein, die nur im Kontext mit einem Kunden sinnvoll ist und daher keine eigene Identität hat.
- *Aggregates* 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. Die Bedingung muss durch eine Berechnung von Werten aus den Bestellzeilen erfüllt werden, 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 an solchen Objekten modelliert werden. Dafür gibt es *Services*.
- *Repositories* dienen dazu, auf die Gesamtheit aller Entities eines Typs zuzugreifen. Typischerweise verbirgt sich dahinter eine Persistenz beispielsweise in einer Datenbank.

Die Implementierung eines Domänenmodells aus diesen Bestandteilen und auch die Idee von *Ubiquituous Language* helfen dabei, objekt-orientierte Systeme zu designen und zu entwickeln. Aber es ist zunächst unklar, welche Bedeutung DDD für Microservices haben können.

Bounded Context

Domain Driven Design gibt aber nicht nur eine Richtlinie dafür an, wie ein Domänenmodell implementiert werden kann, sondern auch

für die Beziehung von Domänenmodellen untereinander. Mehrere Domänenmodelle erscheinen zunächst ungewöhnlich. Immerhin sind Konzepte wie Kunde oder Bestellung eigentlich zentral für das gesamte Unternehmen. Es erscheint also zunächst attraktiv, genau ein Domänenmodell umzusetzen und dabei alle Aspekte des Modells genau zu betrachten. Dann sollte es einfach sein, auf Basis dieser Elemente die Software-Systeme in der Firma umzusetzen.

Bounded Context besagt aber, dass ein solches allgemeines Modell nicht umgesetzt werden kann. Nehmen wir als Beispiel den Kunden aus dem E-Commerce-Shop: Im Kontext des Liefervorgangs ist die Lieferadresse des Kunden relevant. Beim Bestellungsprozess hingegen die speziellen Vorlieben des Kunden und schließlich geht es bei der Rechnungserstellung um die verschiedenen Bezahlmöglichkeiten, zu denen der Kunde Daten hinterlegt hat – beispielsweise seine Kreditkartennummer oder Informationen für einen Lastschrifteinzug.

Theoretisch wäre es vielleicht denkbar, alle diese Informationen in ein allgemeines Kunden-Profil zusammenzutragen. Dieses Modell wäre dann aber extrem komplex. Außerdem wäre es praktisch nicht handzuhaben: Wenn sich die Daten aus einem Kontext ändern, muss das Modell geändert werden und das betrifft dann alle Komponenten, die das Kunden-Datenmodell nutzen – und das können sehr viele sein. Und die Analyse, um zu einem solchen Kundenmodell zu kommen, wäre so komplex, dass sie in der Praxis kaum zu leisten ist.

Bounded Context und Microservices

Daher ist ein Domänenmodell nur in einem bestimmten Kontext sinnvoll – eben in einem *Bounded Context*. Für Microservices bietet es sich an, einen Microservice so zu schneiden, dass er einem *Bounded Context* entspricht. Das gibt eine Orientierung für die fachliche Aufteilung von Microservices. Diese Aufteilung ist besonders wichtig, weil eine gute fachliche Aufteilung die unabhängige Arbeit

an fachlichen Features ermöglicht. Wenn die fachliche Aufteilung sicherstellt, dass jedes Feature in einem eigenen Microservice implementiert wird, kann so die Umsetzung der Features entkoppelt werden. Da ein Microservice sogar alleine in Produktion gebracht werden kann, ist es möglich, fachliche Features nicht nur getrennt zu entwickeln, sondern auch getrennt auszurollen.

Der unabhängigen Entwicklung von Features kommt die Aufteilung in Bounded Contexts ebenfalls zu Gute: Wenn ein Microservice auch die Hoheit über einen bestimmten Ausschnitt der Daten hat, kann der Microservice Features einführen, ohne dabei Änderungen in anderen Microservices zu verursachen.

Wenn in dem E-Commerce-System beispielsweise auch die Bezahlung mit PayPal möglich gemacht werden soll, muss dank *Bounded Context* nur eine Änderung im Microservice für das Rechnungswesen vorgenommen werden. Dort werden die UI-Elemente und die neue Logik implementiert. Da der Microservice für das Rechnungswesen die Daten für den *Bounded Context* verwaltet, müssen lediglich die PayPal-Daten zu den Daten in dem Microservice hinzugefügt werden. Eine Änderung an einem getrennten Microservice, der die Daten verwaltet, ist nicht notwendig. Also kommt der *Bounded Context* auch der Änderbarkeit zugute.

Beziehungen zwischen Bounded Contexts

In seinem Buch beschreibt Eric Evans verschiedene Arten, wie *Bounded Contexts* zusammenarbeiten können. Beispielsweise kann bei *Shared Kernel* ein gemeinsamer *Bounded Context* genutzt werden, in dem die gemeinsamen Daten abgelegt werden. Ein krasser Gegenentwurf ist *Separated Ways*: Beide *Bounded Contexts* nutzen völlig voneinander getrennte Modelle. *Anticorruption Layer* entkoppelt zwei Domänenmodelle. Dadurch kann beispielsweise verhindert werden, dass ein altes und mittlerweile kaum noch sinnvolles Datenmodell aus einem Mainframe im Rest des Systems genutzt werden muss. Durch ein *Anticorruption Layer* werden

die Daten in eine neue, einfach zu verstehende Repräsentation überführt.

Abhängig von dem genutzten Modell für die Beziehung zwischen den *Bounded Contexts* ist natürlich mehr oder weniger Kommunikation zwischen den Teams notwendig, die an den Microservices arbeiten.

Generell ist es also denkbar, dass ein Domänenmodell auch mehrere Microservices erfasst. Vielleicht ist es in dem E-Commerce-System sinnvoll, dass die Modellierung von Basisdaten eines Kunden in einem Microservice umgesetzt wird und nur spezifische Daten in den jeweiligen anderen Microservices abgelegt werden – ganz im Sinne von *Shared Kernel*. Allerdings kann dann auch ein höherer Grad an Koordination zwischen den Microservices notwendig sein, was die getrennte Entwicklung behindern kann.

2.3 Das Gesetz von Conway

Das Gesetz von Conway[3] stammt von dem amerikanischen Informatiker Melvin Edward Conway und besagt:

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

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

[3] http://www.melconway.com/research/committees.html

damit eine Kommunikationsbeziehung zwischen den organisatorischen Einheiten, die für die jeweiligen Teile zuständig sind.

Das Gesetz als Begrenzung der Architektur

Ein Beispiel für die Auswirkung des Gesetztes: Eine Organisation bildet je ein Team aus Experten für die Web-UI, für die Logik im Backend und für die Datenbank (siehe Abb. 2. Diese Organisation hat Vorteile: Der technische Austausch zwischen Experten ist recht einfach und auch Urlaubsvertretungen sind leicht zu organisieren. Die Idee, Mitarbeiter mit ähnlicher Qualifikation jeweils in einem Team arbeiten zu lassen, ist nicht besonders abwegig.

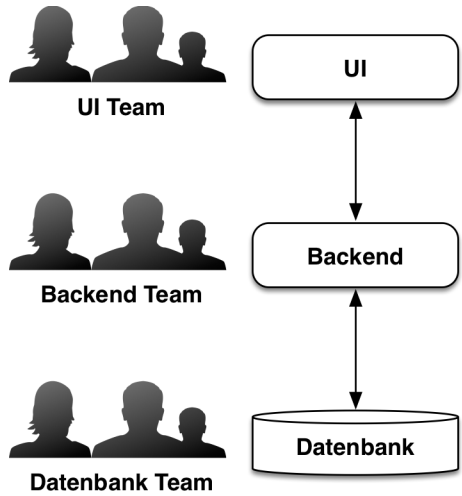

Abb. 2: Aufteilung der Teams nach technischen Skills

Aber nach dem Gesetz von Conway werden die drei Teams jeweils Artefakte in der Architektur schaffen. Es entsteht also eine Datenbankschicht, eine Backend-Schicht und eine Datenbank-Schicht.

Diese Architektur hat einige Nachteile:

- Für die Umsetzung eines Features muss der Anforderer mit allen drei Teams sprechen. Jedem dieser Teams muss er die

Anforderungen erläutern. Wenn der Anforderer kein tiefes Wissen über die Architektur des Systems hat, müssen die Teams zusammen mit dem Anforderer überlegen, wie die Funktionalitäten in den Schichten untergebracht werden können.
- Die Teams müssen sich koordinieren und beispielsweise Schnittstellen abstimmen.
- Ebenso muss sichergestellt sein, dass die Zulieferungen der Teams rechtzeitig erfolgen. Das Backend kann kaum ohne Änderungen an der Datenbank neue Features implementieren. Ebenso kann die UI ohne die Änderungen am Backend nicht umgesetzt werden.
- Die Abhängigkeiten und Koordination verlangsamen auch die Umsetzung von Features. Das Datenbank-Team kann erst am Ende seines Sprints die Änderungen liefern. Darauf baut dann das Backend-Team auf, und das ist wiederum die Basis für die Arbeit des UI-Teams. Die Umsetzung dauert dann drei Sprints. Natürlich sind Optimierungen möglich, aber eine vollständig parallele Umsetzung ist praktisch unmöglich.

Durch die Aufteilung der Teams ergibt sich also eine Architektur, die eine schnellen Umsetzung von Features behindert. Dieses Problem ist vielen gar nicht bewusst, weil der Zusammenhang zwischen der Organisation und der Architektur nicht bekannt ist.

Das Gesetzt von Conway als Enabler

Man kann mit dem Gesetzt von Conway aber auch ganz anders umgehen. Ziel von Microservices ist es ja gerade, eine fachliche Aufteilung umzusetzen, um so die Arbeit an Fachlichkeiten zu vereinfachen und auch die Arbeit zu parallelisieren. Im Zusammenhang mit Microservices gibt es daher einen anderen Umgang mit dem Gesetz von Conway: Statt die Architektur durch die Organisation beeinflussen zu lassen, wird die Architektur zum

Treiber der Organisation. Die Organisation wird so aufgestellt, dass sie die Architektur unterstützt.

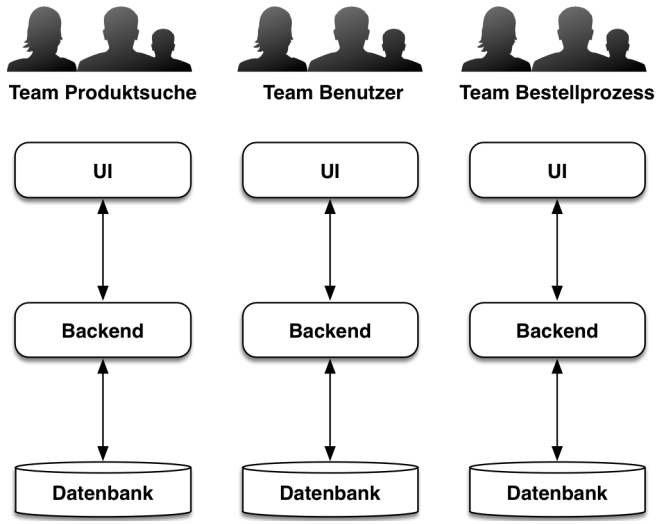

Abb. 3: Aufteilung nach Fachlichkeiten

Abb. 3 zeigt eine mögliche Architektur en Microservices-System: Es gibt jeweils eine Komponente für die Produktsuche, für die Verwaltung von Benutzern und für den Bestellprozess. Für jeden Microservice gibt es jeweils ein Team, dass diesen Microservice umsetzt. Dadurch wird die fachliche Aufteilung in Microservices nicht nur in der Architektur umgesetzt, sondern auch in der Organisation. So wird die Architektur unterstützt: Überschreitungen der Architektur sind schwierig, weil nach dem Gesetz von Conway schon die Organisationsform eine fachliche Architektur erzwingt.

Dennoch müssen die verschiedenen technischen Artefakte für die Microservices umgesetzt werden. Dementsprechend muss es auch in den Teams die jeweiligen technischen Skills geben. In einem Team ist die Koordination der Experten wesentlich einfacher als über Team-Grenzen hinweg. Aus diesem Grund sind nun fachliche Anforderungen, die über verschiedene technische Artefakte hinweg implementiert werden müssen, einfacher zu implementieren.

Organisationskompromisse

In der Praxis kommt es in einem so strukturierten System mit einer unterstützenden Organisation dennoch zu Herausforderungen. Schließlich sollen in dem System Features implementiert werden. Diese Features sind manchmal nicht auf einen Microservice begrenzt, sondern erfordern Änderungen an mehreren Microservices. Außerdem müssen manchmal an einem Microservice mehr Änderungen durchgeführt werden, als das Team umsetzten kann. In der Praxis hat sich dann ein Vorgehen bewährt, bei dem ein Team zwar die Verantwortung für einen Microservice hat, aber andere Teams ebenfalls Änderungen an den Microservices vornehmen dürfen. Wenn ein Feature Änderungen an mehreren Microservices bedingt, kann ein Team alle diese Änderungen durchführen, ohne Änderungen bei einem anderen Team priorisieren zu lassen. Außerdem können mehrere Teams an einem Microservice arbeiten, um so mehr Features in einem Microservice umzusetzen. Das für den Microservice zuständige Team ist dennoch für den Microservice verantwortlich. Insbesondere muss es alle Änderungen überprüfen, um eine sinnvolle Weiterentwicklung sicherzustellen.

Microservice pro Team?

Übrigens ist es nicht unbedingt notwendig, dass ein Team nur einen Microservice umsetzt. Das Team kann auch durchaus mehrere Microservices implementieren. Wichtig ist allerdings, dass ein Team eine möglichst genau definierte fachliche Zuständigkeit hat und dass eine Fachlichkeit in möglichst wenigen Microservices umgesetzt wird. Es kann durchaus wünschenswert sein, kleinere Microservices zu implementieren, so dass ein Team mehr als einen Microservice verantwortet.

2.4 Fazit

Die Größe von Microservices stellt bei der Definition eher den technischen Aufbau des Systems in den Mittelpunkt. Bei der Aufteilung anhand von Bounded Context steht die fachliche Architektur im Mittelpunkt. Das Gesetzt von Conway zeigt, dass Microservices auch eine Auswirkung auf die Organisation haben. Nur zusammen ergeben diese Aspekte ein echtes Bild von Microservices. Welcher dieser Aspekte der Wichtigste ist, hängt vom Einsatzkontext der Microservice-Architektur ab.

3 Warum Microservices?

Es gibt mehr als einen Grund, Microservices einzusetzen. Je nach Einsatzkontext können die Architekturentwürfe völlig unterschiedlich aussehen. Es ist also nicht nur wichtig, die Vorteile zu kennen, sondern sie auch für das konkrete abzuwägen und eine passende Architektur umzusetzen.

3.1 Agilität skalieren

Wie schon in Abschnitt 2.3 erwähnt, können Microservices Auswirkungen auf die Organisation haben. Idealerweise sollte jeder Microservice von einem Team weiterentwickelt werden oder es sollte zumindest genau ein Team zuständig sein.

Das eröffnet Möglichkeiten für die Skalierung agiler Projekte: Normalerweise müssten alle Teams sich abstimmen und gemeinsam an den Features arbeiten. Wenn jedes Team einen eigenen Strom von Anforderungen hat und sie durch Änderungen am eigenen Microservice umsetzten kann, können die Teams weitgehend unabhängig voneinander an Features arbeiten. Damit können auch größere Projekte agil angegangen werden. Im Prinzip wird das System nun in mehrere kleine Projekte aufgeteilt, die jeweils unabhängig arbeiten können. Neben der fachlichen Aufteilung hilft es auch, dass Microservices Features ohne eine Beeinflussung anderer Microservices in Produktion bringen können. So ist eine weitgehend unabhängige Weiterentwicklung möglich.

Neben der fachlichen Unabhängigkeit bieten Microservices auch eine technische Unabhängigkeit: Technologie-Entscheidungen können auf einen Microservice begrenzt werden. So wird die Unabhängigkeit der Teams erweitert: Sie können nicht nur weitgehend

unabhängig Features umsetzten, sondern auch jeweils eigene technische Entscheidungen treffen.

So ermöglichen Microservices die unabhängige Entwicklung der einzelnen Microservices und damit ein einfaches Skalieren agiler Prozesse auf größere Projekt-Organisationen.

3.2 Legacy-Anwendungen migrieren

Die Arbeit mit Legacy-Code ist oft schwierig: Das ist System ist schlecht strukturiert, so dass es schwierig ist, sich einen Überblick zu verschaffen. Ebenso ist die Qualität des Codes niedrig und schließlich fehlen Tests. Oft ist dann noch die technologische Basis veraltet, so dass moderne Ansätze nicht genutzt werden können.

Einige dieser Probleme sind lösbar, wenn man den Ansatz für die Modifikation des Systems etwas verändert: Statt den Codes des Legacy-Systems zu modifizieren, wird das System an seinen externen Schnittstellen durch Microservices ergänzt oder teilweise abgelöst. Der Vorteil: Statt den Legacy-Code zu editieren, wird dieser Code praktisch gar nicht modifiziert, sondern nur durch externe Systeme ergänzt.

Das Fernziel ist die vollständige Ablösung des Legacy-Systems durch eine Menge von Microservices. Aber man kann das Legacy-System durch Microservices ergänzen, ohne dabei zu viel Zeit in Vorbereitungen zu stecken und den Weg einfach ausprobieren. Wenn die Microservices keine sinnvolle Lösung sind, können sie auch aus dem System recht einfach wieder entfernt werden. Die einfache Integration von Microservices ist ein Grund, warum Microservices so interessant sind. Die Ablösung eines Legacy-Systems durch eine Menge von Microservices ist für viele ein guter Weg, um Vorteile wie Continuous Delivery möglichst schnell in einem System zu realisieren.

3.3 Nachhaltige Entwicklung

Durch Microservices wird ein System in mehrere, getrennt deploybare Dienste aufgeteilt. Die Aufteilung des Systems in Microservices ist eine wichtige Architektur-Entscheidung. Sie legt die Zuständigkeit der Komponenten fest.

In einem Deployment-Monolithen gibt es eine solche Architektur auch. Aber in einem Deployment-Monolithen geht sie oft nach einiger Zeit verloren. Es ist nämlich sehr einfach, neue Abhängigkeiten in einen Deployment-Monolithen einzubauen: Es muss nur eine Klasse neu referenziert werden. Die Architektur eines E-Commerce-Systems definiert beispielsweise, dass der Bestellprozess die Rechnungserstellung aufrufen soll. Die Rechnungserstellung darf aber den Bestellprozess nicht aufrufen. Abhängigkeiten in nur eine Richtung haben den Vorteil, dass Module änderbar bleiben. Im Beispiel würde ist eine Änderung des Bestellprozesses möglich, ohne dass der Rechnungsprozess geändert werden müsste. Eine Änderung des Rechnungsprozesses hingegen kann den Bestellprozess beeinflussen, da der Bestellprozess den Rechnungsprozess nutzt.

Bei der Implementierung eines Features im Rechnungsprozess hat ein Entwickler dann doch Funktionalitäten aus dem Bestellprozess genutzt. Das passiert recht schnell. Die Erfahrung ist, dass auf diese erste Abhängigkeit bald weitere folgen und irgendwann können die beiden Komponenten nicht mehr unabhängig voneinander weiterentwickelt werden, weil sie sich wechselseitig benutzen. Bei Microservices ist es nicht so ohne weiteres möglich, ein anderen Microservice zu nutzen. Die Microservices haben eine Schnittstelle, so dass die Nutzung nur möglich ist, wenn man diese Schnittstelle verwendet. Dazu muss die Schnittstelle mit Technologien wie REST oder Messaging angesprochen werden. Das passiert nicht einfach aus Versehen.

Wenn der zu verwendende Microservice von einem anderen Team weiterentwickelt wird, kann es sogar notwendig sein, mit diesem

Team zu sprechen. Letztendlich ist also die Aufteilung der Architektur in Microservices relativ stabil und im Gegensatz zu Deployment Monolithen kann es nicht so ohne weiteres zu einem Verlust der Architektur kommen. Ähnliche Ergebnisse lassen sich natürlich auch erzielen, wenn andere Maßnahmen ergriffen werden, um die Integrität der Architektur zu erzwingen. Beispielsweise gibt es Architektur-Werkzeuge, die Entwickler auf die Verletzung von Architektur-Regeln hinweisen. Bei Microservices sind diese Maßnahmen aber schon in das System integriert.

Ersetzbarkeit

Eine weitere wichtige Eigenschaft von Microservices ist die Ersetzbarkeit: Ein Microservice kann ohne großen Aufwand durch eine Neuimplementierung ersetzt werden. Damit ist ein weiteres Problem der Legacy-Systeme lösbar: Wenn ein Legacy-System nicht mehr gewartet werden kann, kann es oft auch nicht neu geschrieben werden, weil der Aufwand dafür zu hoch wäre. Microservices zu ersetzten ist hingegen nicht sonderlich schwierig.

Fazit

Innerhalb eines Microservice sollte es auch langfristig einfach sein, neue Features zu implementieren, da der Microservice klein ist. Wenn er doch nicht mehr wartbar ist, kann er ersetzt werden. Die Architektur im Zusammenspiel der Microservices sollte auch langfristig erhalten bleiben. Damit kann die Wartbarkeit des Systems auch langfristig gewährleistet werden. Microservice-Systeme versprechen also eine langfristige gute Wartbarkeit und Änderbarkeit der Software-Systeme.

3.4 Robustheit

In einem Microservice-System ist die Robustheit für bestimmte Probleme sehr hoch: Wenn in einem Deployment-Monolithen eine Funktionalität sehr viel CPU oder Speicher verbraucht, werden andere Module ebenfalls beeinflusst. Wenn im Extremfall ein Modul das System zum Absturz bringt, sind alle anderen Module ebenfalls nicht mehr verfügbar.

Ein Microservice ist ein eigener Prozess oder sogar eine eigene virtuelle Maschine. Ein Problem in einem Microservice beeinflusst einen anderen Microservice also nicht, weil das Betriebssystem oder die Virtualisierung die Microservices gegeneinander isoliert.

Dennoch sind Microservices ein verteiltes System. Sie laufen also auf mehreren Servern und nutzen das Netzwerk. Server und Netzwerk können ausfallen. Also sollte ein Microservices-System als ganzes nicht sehr robust sein, weil es diesen Gefahren verstärkt ausgesetzt ist. Microservices müssen daher gegen den Ausfall anderer Microservices abgesichert sein. Man spricht von „Resilience". Die Implementierung von Resilience kann sehr unterschiedlich sein: Wenn der Bestellprozess nicht abgeschlossen werden kann, ist vielleicht ein erneuter, späterer Versuch eine Option. Wenn eine Kreditkarte nicht verifiziert werden kann, wird vielleicht bis zu einer gewissen Obergrenze die Bestellung dennoch durchgeführt. Welche Grenze das genau ist, müssen dann Fachexperten entscheiden.

Durch Resilience kann also ein Microservice-System sehr robust werden. Die Basis legt dazu die strikte Trennung in Prozesse oder virtuelle Maschinen.

3.5 Continuous Delivery

Continuous Delivery [4] ist ein Ansatz, bei dem Software regelmäßig in Produktion gebracht wird. Basis ist dabei vor allem ein weitgehend automatisierter Prozess, wie Abb. 4 ihn zeigt:

- In der Commit-Phase werden Unit Tests und statische Code-Analyse ausgeführt.
- Die automatisierten Akzeptanztests garantieren, dass die Software Features korrekt umsetzt.
- Kapazitätstest hingegen überprüfen, ob die Performance stimmt und die erwartete Last abgedeckt werden kann.
- Manuelle Tests können neue Features aber auch Fehlerschwerpunkt untersuchen.
- Schließlich kommt die Software in Produktion.

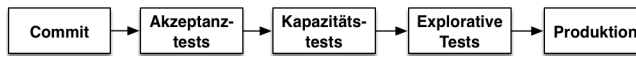

Abb. 4: Continuous Delivery Pipeline

Continuous Delivery ist mit Deployment-Monolithen nur schwer umsetzbar:

- Die Automatisierung der Tests und der Deployments ist aufwändig, weil Deployment-Monolithen schwierig in Produktion zu bringen sind. Eine Rolle spielt beispielsweise die Datenbank, die recht groß sein kann und viele Daten enthalten kann. Ebenso müssen viele Drittsysteme integriert oder zu simuliert werden.
- Die Tests sind aufwändig. Gerade bei einem Deployment-Monolithen können Änderungen leicht Nebenwirkungen haben, die nicht beabsichtigt sind. Daher muss für jede Ände-

[4] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, dpunkt Verlag, 2016, ISBN 978-3864903717

rung ein umfangreicher Regressionstest durchgeführt werden. Das ist aber aufwändig und macht die Continuous-Delivery-Pipeline langsam.
- Schließlich ist es aufwändig, eine Release abzusichern. So wäre es denkbar eine zweite Umgebung aufzubauen, in der Umgebung die neue Version zu deployen und erst auf die neue Version umzuschalten, wenn sie beispielsweise noch einmal durchgetestet worden ist. Ebenso ist dann ein Rückfall auf die alte Version machbar. Bei einem Deployment-Monolithen sind solche Ansätze nur schwer umsetzbar, weil die Umgebung zu groß und zu komplex ist.

Microservices sind unabhängige Deployment-Einheiten. Daher können sie auch unabhängige Continuous-Delivery-Pipelines haben. Diese Pipelines sind relativ einfach aufzubauen und die Microservices werden auch relativ schnell durch die Continuous-Delivery-Pipeline in Produktion gebracht. Auch sind die Deployments von Microservices einfacher abzusichern. Alle oben genannten Probleme der Deployment Monolithen sind durch die geringere Größe von Microservices lösbar und Continuous Delivery wird dadurch wesentlich vereinfacht.

Natürlich sind dazu einige Vorkehrungen notwendig, um das unabhängige Deployment tatsächlich zu ermöglichen. Dennoch sind gerade die Vorteile bei Continuous Delivery ein wichtiger Grund, warum sich viele für Microservices interessieren.

3.6 Unabhängige Skalierbarkeit

Jeder Microservice läuft als ein eigener Prozess, gegebenenfalls sogar in einer eigenen virtuellen Maschine. Wenn also eine bestimmte Funktionalität besonders stark genutzt wird, kann nur der dafür notwendige Microservices skaliert werden, während die anderen Microservices weiterhin mit derselben Kapazität laufen.

Das hört sich zunächst nach keinem besonders großen Vorteil an, aber in der Praxis ergeben sich dadurch erhebliche Vorteile, weil die Skalierung einfacher ist. Im Allgemeinen sind die Performance-Anforderungen nur für bestimmte Fälle wirklich anspruchsvoll. Die unabhängige Skalierbarkeit erlaubt es, sich auf diese Fälle zu konzentrieren und zwar mit weniger Aufwand, als dies bei einem Deployment-Monolithen der Fall wäre. Also kann dieser Grund schon wesentlich für die Einführung von Microservices sein.

3.7 Technologie-Wahlfreiheit

Im Prinzip kann jeder Microservice in einer anderen Technologie umgesetzt sein. Das macht natürlich das System insgesamt komplexer, wobei dem beispielsweise durch Standards für den Betrieb, das Monitoring, die Log-Formate oder das Deployment begegnet werden kann. Dann ist zumindest der Betrieb weitgehend einheitlich.

Dennoch kann beispielsweise für die Produktsuche nun eine eigene Such-Technologie genutzt werden, ohne dass dazu eine umfangreiche Koordination mit allen anderen Microservices und Teams notwendig ist. Wenn ein Team ein Bugfix in einer Bibliothek benötigt und daher eine neue Version nutzen will, ist die Änderung ebenfalls auf das eine Team beschränkt und kann daher auch von diesem einen Team durchgeführt werden, das dann auch das Risiko trägt. Bei einem Deployment-Monolithen wäre eine umfangreiche Abstimmung notwendig und auch entsprechend mehr Tests.

Schließlich können neue Technologien ohne einen großen Migrationsaufwand ausprobiert werden. Das Risiko und der Aufwand sind begrenzt: Es kann zunächst ein einziger Microservice migriert werden. Wenn das nicht funktioniert, fällt nur dieser aus und bei einem großen Problem muss nur dieser eine Microservice neu implementiert werden. Ein Projekt, um den Deployment-Monolithen auf eine neue Technologie zu heben, ist bei Microservices nicht

notwendig und die Migration ist viel einfacher. Das hat auch andere positive Konsequenzen: So können Mitarbeiter eher neue Technologien ausprobieren, was meistens die Motivation hebt.

3.8 Fazit

Microservices haben sehr viele Vorteile. Welche Vorteile tatsächlich die wichtigsten sind, hängt vom konkreten Kontext ab. Bei vielen Projekten steht die Ablösung eines Deployment-Monolithen im Vordergrund. Dann ist der einfache Umgang mit Legacy-Systemen ein wichtiger Vorteil bei der Migration. Grund für die Migration ist in solchen Fällen oft die Skalierung agiler Prozesse und die einfachere Umsetzung von Continuous Delivery.

Aber es gibt auch ganz andere Szenarien, in denen beispielsweise eine Anwendung im Betrieb stabiler werden soll. Dann ist die Robustheit ein wichtiger Treiber und die unabhängige Skalierung kann ein weiterer wichtiger Vorteil sein.

Also hängen die wesentlichen Vorteile vom jeweiligen Kontext ab. Von den erwarteten Vorteilen hängt auch ab, wie Microservices bei dem System genau genutzt werden sollte.

4 Wie weiter?

Diese Broschüre kann nur eine kurze Einleitung in Microservices geben. Daher ist eine wichtige Frage, wie man nach dem Studium der Broschüre weitermachen kann.

4.1 Microservices: Nur ein Hype?

Microservices sind mehr als ein Hype. Amazon setzt die Trennung in Teams mit ihren jeweils eigenen Technologien schon seit 2006 ein. Diese Architektur und diesen Ansatz kennen wir heute als Microservices. Pioniere wie Netflix versprechen sich von diesem Architektur-Ansatz so große Vorteile, dass sie selber erheblich in den notwendigen Infrastrukturen investiert haben. Heutzutage stehen diese Technologien allen offen, so dass der Einstieg und die Nutzung einfacher und auch weniger kostenintensiv sind.

Auch der Trend zu Agilität, Continuous Delivery und Cloud findet in Microservices eine Entsprechung bei der Architektur. Auch darüber hinaus gibt es viele gute Gründe für Microservices – seien es die individuelle Skalierbarkeit oder die Robustheit. Microservices sind also nicht nur eine gute Ergänzung zu einigen anderen Trends, sondern sie sind vor allem eine Lösung für verschiedene Probleme. Der Trend basiert also auf einer ganzen Reihe von Gründen. Daher ist es unwahrscheinlich, dass nur ein kurzlebiger Hype ist.

4.2 Beispiele

Die Broschüre betrachtet das Thema Microservices nur theoretisch und zeigt auch keine Technologien, um Microservices umzusetzen.

Unter http://ewolff.com/microservices-demos.html finden sich einige Beispiele, die verschiedene Optionen für die Implementierung von Microservices zeigen:

Synchrone Kommunikation

Für synchrone Kommunikation gibt es folgende Beispiele:

- Die Consul-Demo[5] ist in Java mit Spring Cloud / Boot geschrieben. Die Demo nutzt Consul für Service Discovery, Apache httpd für Routing, Hystrix für Resilienz und Ribbon für Load Balancing Es gibt auch eine Prometheus-Installation[6] für das Monitoring und einen ELK-Stack[7] für die Analyse der Log-Dateien.
- Die Netflix-Demo[8] ist ebenfalls in Java mit Spring Cloud / Boot geschrieben. Die Demo verwendet Netflix Eureka für Service Discovery, Netflix Zuul für Routing, Hystrix für Resilience und Ribbon für Load Balancing.
- Kubernetes ist eine Umgebung für den Betrieb von Docker im Cluster. Auch die Kubernetes-Demo[9] ist in Java mit Spring Cloud / Boot geschrieben. Sie verwendet Kubernetes für Service Discovery, Routing und Load Balancing. Die Demo benutzt auch Hystrix für Resilience Der Code hängt nicht von Kubernetes ab.
- Cloud Foundry ist ein PaaS (Platform as a Service). Der PaaS kümmert sich darum, eine Anwendung zu deployen und zu betreiben. Das Cloud-Foundry-Beispiel[10] ist in Java geschrieben mit Spring Cloud / Boot. Es verwendet Cloud Foundry für Deployment, Service Discovery, Routing und Load Balancing. Die Demo nutzt ebenfalls Hystrix für Resilience. Der Code hängt nicht von Cloud Foundry ab.

[5] https://github.com/ewolff/microservice-consul
[6] https://github.com/ewolff/microservice-consul#prometheus
[7] https://github.com/ewolff/microservice-consul#elastic-stack
[8] https://github.com/ewolff/microservice
[9] https://github.com/ewolff/microservice-kubernetes
[10] https://github.com/ewolff/microservice-cloudfoundry

Asynchrone Kommunikation

Asynchrone Kommunikation macht den Umgang mit unzuverlässigen Services oder Netzwerken einfacher:

- Kafka[11] verwendet Kafka für Kommunikation, eine Message-oriented Middleware, die Nachrichten verschicken kann.
- Atom[12] verwendet REST / HTTP zur asynchronen Kommunikation mit dem Atom-Format.

UI Integration

Die Integration auf Ebene der UI führt zu einer sehr losen Kopplung:

- ESI[13] zeigt, wie Edge Side Includes (ESI) verwendet werden können, um die Benutzeroberfläche von Microservices zu integrieren. Ein Microservice ist in Java mit Spring Boot geschrieben, der andere mit Go. Der Go-Microservice wird mit Multi Stage Docker Containern gebaut.
- JQuery[14] zeigt, wie jQuery kann zur Integration der Benutzeroberfläche von Mikroservices genutzt werden.

Jedes dieser Beispiele hat eine umfangreiche Dokumentation, die auch erläutert, wie man die Beispiele laufen lassen kann.

4.3 Weitere Literatur

Die Broschüre kann nur einen groben Eindruck von Microservices geben. Sie kann nur eine Einleitung in Microservices geben. Weiterführende Literatur [15] kann nützlich sein. Dieses Buch gibt es auch

[11] https://github.com/ewolff/microservice-kafka
[12] https://github.com/ewolff/microservice-atom
[13] https://github.com/ewolff/SCS-ESI
[14] https://github.com/ewolff/SCS-jQuery
[15] Eberhard Wolff: Microservices - Grundlagen flexibler Softwarearchitekturen, dpunkt Verlag, 2015, ISBN 978-3864903137

in Englisch[16].

Für die technische Implementierung von Microservices bietet die kostenlosen Microservices Rezepte[17] einen guten Startpunkt. Sie zeigen verschiedene Implementierungsalternativen auf Basis der bereits erwähnten Demos. Auch diese Broschüre gibt es auf Englisch[18].

Eine umfangreiche Darstellung technischer Implementierungsmöglichen für Microservices bietet das Microservices-Praxisbuch[19], das es ebenfalls auf Englisch[20] gibt.

4.4 Loslegen

Das Risiko bei Microservices ist jedoch gering: Man muss lediglich einen Microservice entwickeln und in Produktion bringen. Der Service kann auch einen vorhandenen Deployment-Monolithen ergänzen. Und wenn der Ansatz nicht funktioniert, kann man den Microservice auch recht einfach wieder entfernen.

[16] http://microservices-book.com/
[17] http://microservices-praxisbuch.de/rezepte.html
[18] http://practical-microservices.com/recipes.html
[19] http://microservices-praxisbuch.de/
[20] http://practical-microservices.com/