Planung und Erstellung einer Backend-Microservices-Architektur aus den Anforderungen durch das Spiel Stirnraten.

Michael Rothkegel

7. April 2019

Inhaltsverzeichnis

1 Glossar

Begriff	Erklärung
Deployen	Ein Softwareprodukt updaten.
Hosten	Ein Softwareprodukt auf einem Server bereitstellen. Dafür stehen heutzutage viele Möglichkeiten bereit, vom eigenen Server, gemieteten Servern oder Cloudlösungen (Azure, AWS, Google Cloud)
CI/CD	Continuous Integration/Continuous Delivery beschreibt den kontinuierlichen Prozess ein Softwareprodukt von der Entwicklung bis zur tatsächlichen Veröffentlichung zu begleiten. Dies sollte automatisch funktionieren, schnell gehen und leicht auszuführen sein.
User Interface	Oberfläche für Benutzer
VM	Virtuelle Maschine, Kapselung eines Rechnersystems innerhalb eines anderen
Legacy	Es handelt sich um Altsysteme, die eine Erneuerung benötigen oder abgelöst werden müssen. Das kann verschiedene Gründe haben, wie z.B. schlechte Programmierung, Fehlentscheidungen oder technologische Veraltung
Technologiestack	
Overhead	
Downtime	Zeitpunkt, die eine Anwendung benötigt, bis sie neugestartet ist. Während der Downtime ist eine Anwendung nicht erreichbar.
technische Schuld	
Monolith	monolitsche Struktur
Microservice	Service
Domain Driven Design (DDD)	
Bounded Contexts	
up-stream	
down-stream	
Pattern	
REST	
Monitoring	
Deployment	
Logging	
Traffic	

2 Einleitung

3 Grundlagen

Hier erklären was kommt

3.1 Microservices

Für den Begriff Microservices existiert keine einheitlich anerkannte Definition. Während Wolff unter Microservices unabhängig, deploybare Module versteht[7], spricht Newman von kleinen, autonomen Services, die zusammenarbeiten. Cockcroft verwendet den Begriff Microservice gekoppelt mit einem Architekturbegriff: Eine Microservice Architektur sind gekoppelte Services, welche für einen gewissen Kontextbereich zuständig sind.[4] D.h. jeder Service behandelte gewisse, fachliche Aufgaben und kann genau für diese genutzt werden. Eine Vielzahl von solchen Services bildet dann die gesamte Anwendung.

Amudsen schreibt dem Microservice an sich die Eigenschaft zu, dass er unabhängig zu anderen Microservices sein muss, d.h. ein Microservice kann losgelöst von anderen geupdated (deployed) werden. Weiter ist ein Microservice wie schon bei Cockcroft für einen gewissen Aufgabenbereich zuständig. Eine Microservice-Architektur ist ein zusammenschluss von miteinander kommunizierenden Microservices.[4]

In *Flexible Software Architecture*[8] werden Microservices zu den bisherigen noch weitere, teils technische Eigenschaften zugeschrieben: Microservices sind technologisch unabhängig, d.h. eine Microservice Architektur ist beispielsweise nicht an eine bestimmte Programmiersprache oder Datenbank gebunden. Weiter müssen Microservices einen privaten Datenspeicher haben und sie kommunizieren mit anderen Services über das Netzwerk (z.B. über REST). Ebenfalls werden Microservices verwendet, um große Programme in kleine Teile zu unterteilen. Diese kleine Teile lassen sich automatisch bauen und deployen.

Basierend auf den folgenden Definitionen wird der Microservice Begriff wie folgt verwendet: Microservices sind

- klein in der Größe
- kommunizieren mit anderen Services über Netzwerkschnittstellen (z.B. REST) sind unabhängig voneinander deploybar
- können unabhängig voneinander entwickelt werden (d.h. Microservice A muss nicht auf B,C,D ... warten und/oder umgekehrt)
- eingeschränkt in ihrer Geschäftslogik, d.h. ein Microservice kümmert sich immer um einen speziellen Kontext, der im vorhinein definiert werden muss

- dezentral, d.h. sie können auf unterschiedlichsten Plattformen gehosted werden werden automatisch gebaut und deployed

Abschließend handelt es sich um eine Microservice-Architektur, wenn viele Microservices nach Definition verwendet werden.

3.2 Monolithische Struktur

Eine monolithische Struktur ist ein einziges Softwareprogramm (Monolith), welches in sich geschlossen ist. Dies bedeutet im Detail, dass ein Monolith aus mehreren Ebenen besteht auf die über Schnittstellen zugegriffen werden kann. Innerhalb der Ebenen werden Komponenten wie z.B. Frameworks oder selbstgeschriebene Klassen eingebunden und verwendet.[1] Durch die sich aufeinander aufbauenden Ebenen folgt daraus, dass sämtliche Geschäftslogiken, User Interfaces sowie die Datenbank und Datenbankzugriffe Abhängigkeiten haben. All dies ist in einem Programm vereint.[1] Natürlich kann die monolithische Struktur innerhalb noch einmal Modular sein. In einem Monolithen existierten also ggf. mehrere Module, welche verschiedene Geschäftslogiken abbilden oder ein Modul, welches nur zum Erstellen einer grafischen Oberfläche verwendet wird. Dennoch können diese Module nicht unabhängig von der gesamten Anwendung deployed werden.[5]

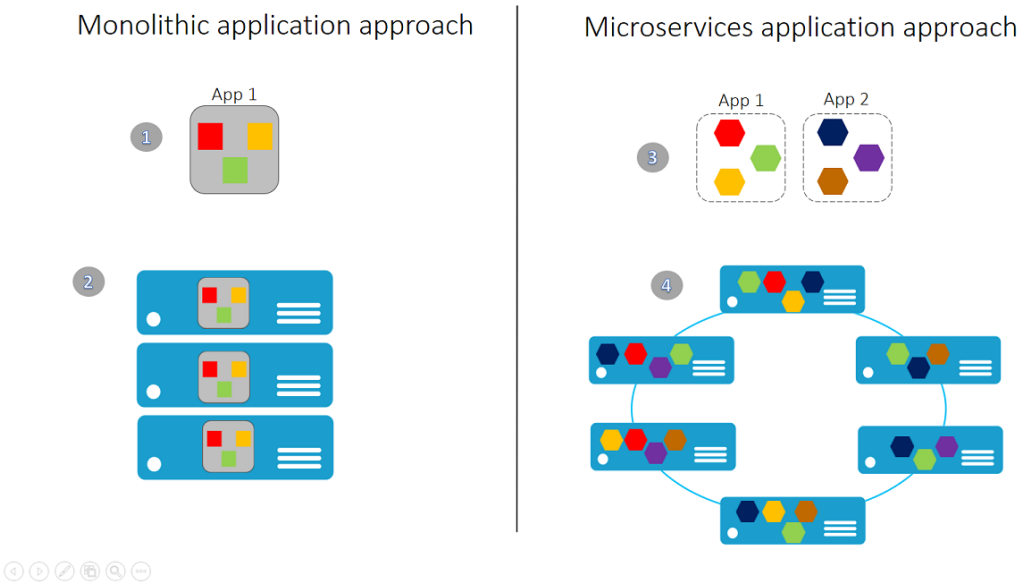

Abbildung 1: Monolith und Microservice-Architektur.[1]

Abbildung 1 zeigt eine vereinfachte Gegenüberstellung der beiden Architekturen. Die App 1 ist in drei klassische Funktionen (Web, Business und Data) unterteilt. Die Skalierung (2) kann durchgeführt werden, in dem App 1 über mehrere Server oder VMs geklont wird.

Bei der Microservice Architektur werden die Funktionen auf unterschiedliche Dienste aufgeteilt. Konkreter könnte dies bedeuten, dass App 1 bei (3) zuständig für eine Benutzerkontoverwaltung ist und App 2 für ein Abrechnungssystem. Die Microservices (4) werden nicht geklont, sondern können unabhängig voneinander bereitgestellt werden.

3.3 Monolith vs. Microservices

Aus den vorherigen Abschnitten sind diverse Unterschiede zwischen den Architekturen erkennbar. Nun gilt es festzustellen, für welche Problemstellungen, welche Architektur sinnvoller ist. [8] [2]

	Monolithische Architektur	Microservice-Architektur
Abhängigkeiten	alles in einer Anwendung	entkoppelt, da Prinzip von Modularisierung verwendet wird
Größe	linear steigend	einzelne Services sind klein
Geschwindigkeit Zugriffe	schnell, da alles in einer Anwendung	Zugriffe können länger dauern
Deployment	schwieriger desto größer das Projekt, aufgrund von • Abhängigkeiten • Größe	einfach, da Microservices • klein und • modular sind
Organisation	leichter, da alles an einem Ort	schwerer, da mehr Domänenlogik (wer macht was?) beachtet werden muss
Legacy-Systeme ablösen	ggf. schwierig, da System sehr verzahnt miteinander	leicht, da Microservices durch neue abgelöst werden können
Technologie	beschränkt	vielfältig
Nachhaltige Entwicklung	wartbar mit Einschränkungen	leicht wartbar
Robustheit	weniger, da ganzes System bei schweren Fehlern abstürzt	sehr, da im Zweifel immer nur ein Service abstürzt
Skalierung	horizontale und vertikale Skalierung, Umsetzung kann sehr komplex werden	horizontale und vertikale Skalierung
Betrieb	nur ein System	komplex, da mehr Services verwaltet werden müssen

Aus der Tabelle ergeben sich verschiedene Punkte: Der Monolith eignet sich besonders dann sehr gut, wenn die Projekt- sowie Teamgrößen absehbar sind und auch die

Technologie entschieden ist. Zusätzlich kann es beim Projektanfang ein Vorteil sein, da die Abhängigkeiten innerhalb des Projektes liegen und so Entwicklungsgeschwindigkeit nicht durch komplizierte Infrastrukturen blockiert wird. Ist die Projektgröße allerdings nicht absehbar, treten früher oder später mehrere Schwierigkeiten auf: Zum einen bindet der am Anfang des Projektes festgelegte Technologiestack und die Nutzung oder der Austausch neuer Technologien sind in der Regel mit sehr viel Arbeit verbunden. Des Weiteren führen die anfangs eingegangen Abhängigkeiten zu problemenen im Deployment (A kann erst updaten, wenn B soweit ist) und einem erhöhten Aufwand in der Kommunikation zwischen den Teams (A kann erst beginnen, wenn B xy erledigt hat).

Zusätzlich ist die Skalierung von Microservices unabhängiger. Es können sich feingranular Services gesucht werden, welche skaliert werden sollen. Diese benötigen nicht zwingend mehr Hardware (vertikale Skalierung), sondern könnten z.B. auch auf verschiedene Server verteilt werden (horizontale Skalierung). Dies ist bei einem Monolithen natürlich auch möglich, dennoch muss immer der ganze Monolith skaliert werden, welcher zum einen immer mehr Hardware als einzelne Microservices benötigt und zum anderen auch durch die Komplexität in der Regel auch schwerer zu skalieren ist.[7] Als abschließender Punkt ist die Robustheit zu erwähnen: Wenn ein Microservice einen Fehler enthält, stürzt dieser im schlechtesten Fall ab. Im besten Fall übernimmt dieser Service eine weniger wichtige Funktion und der Nutzer bemerkt den Ausfall noch nicht einmal. Beim Monolithen dagegen stürzt die gesamte Anwendungen ab. In der Regel startet so eine Anwendung automatisch neu, jedoch ist betrifft die Downtime alle Nutzer.

Aus den genannten Punkten lässt sich schließen, dass eine generelle Aussage, ob eine monolithische oder Microservice-Architektur besser oder schlechter ist, sich nicht treffen lässt. Es kommt immer drauf an, welche Zielsetzung und wie viele Ressourcen für das Projekt festgelegt sind. REWE Digital beispielsweise hat ihr Produkt zuerst als Monolithen gestartet und ist erst später auf eine Microservice-Architektur umgeschwenkt.[3] Zwei mögliche Gründe könnten dafür sein, dass zum einen ein lauffähiges Produkt schneller mit einer monolithischer Struktur zu erreichen ist, zum anderen ist nicht gegeben, ob ein entwickeltes Projekt überhaupt die Nachfrage erzeugt, so dass eine Microservice-Architektur notwendig ist. Dementsprechend muss abgewogen werden, welche Architektur für welchen Anwendungsfall besser geeignet ist.[7]

3.4 Architektur von Micrsoservices

Wie bereits erwähnt, ist die Entkopplung von Microservices ein großer Vorteil gegenüber dem Monolithen. Dennoch ist es sinnvoll Richtlinien, Regeln und/oder Festlegungen zu schaffen, damit die Microservices nicht blockierend oder technologisch unnötig gegensätzlich arbeiten. Die Entscheidungsebene kann global (Makroarchitektur) oder nur für einen einzelnen Services (Mikroarchitektur) gelten.[8] Welche Festlegungen und mit welcher Strenge diese eingehalten werden müssen, hängt von verschiedenen Faktoren ab, welche technologisch, organisatorisch oder wirtschaftlich motiviert sein können.[3]

In dem folgenden Abschnitt wird das Grundprinzip der Software-Modellierungs-Methodik Domain Driven Design untersucht. Zusätzlich wird erläutert, welche Fälle makro- oder microarchitektonisch einzuordnen sind.

3.4.1 Domain Driven Design

Domain Driven Design (DDD) ist ein Vorgehen mit dem ein Softwaresystem modelliert werden kann. Im Sinne einer Microservice-Architektur kann dies als Werkzeug genutzt werden, um Microservices fachlich einzuteilen.[6] Beim sogenannten *Strategic Design* wird dafür das Softwaressytem in verschiedene *Bounded Contexts* eingeilt, welche an ein *Domänenmodell* gebunden sind. Ein Domänenmodell bildet die Geschäftslogik ab, d.h. inwiefern einzelne Objekte innerhalb des Kontexts in Relation zueinander stehen, welche Eigenschaften sie haben und wie sich verhalten. Dabei kann ein Domänenmodell - je nach Entwurfsmuster - von einem oder mehreren Bounded Contexts genutzt werden wie in Abbildung XXX zu sehen ist.[7]

Wolff verdeutlicht das Prinzip von Bounded Contexts mit Hilfe von vier Microservices, welche einen Onlineshop repräsentieren: *Suche*, *Check Out*, *Inkasso* und *Lieferung*. Während das Datenmodell der Suche detaillierte Informationen über die Produkte enthält, reicht es im Warenkorb (hier Check Out), wenn ggf. nur der Produktname gespeichert wird. Bei Inkasso ist es ähnlich: An dieser Stelle sind Zahlungsdaten des Benutzers relevant, während bei Lieferung die ggf, nur Adresse notwendig ist.[7] An diesem Beispiel wird deutlich, dass zwar von selben Begrifflichkeiten wie Benutzer, Produkt usw. gesprochen wird, allerdings jeder Service sein eigenes Domänenmodell hat. Durch diese Technik, was dabei hilft, eine saubere Microservice-Architecktur zu erstellen.[7] [6]

Neben der fachlichen Trennung bildet DDD auch die Kommunikation zwischen Kontexten ab. Dabei wird grundsätzlich vom *up-stream* (vorgeschaltet) und dem *down-stream* (nachgeschaltet) gesprochen.[7]. Der up-stream stellt dem down-stream Informationen bereit. Wie dies technisch umgesetzt ist, also ob der down-stream nachfragt oder der up-stream aktiv Daten schickt, ist frei wählbar.

Nach dem Anwenden von DDD sollte die Struktur der Software erkennbar sein: D.h. welche Art Microservices werden benötigt und iwiefern sie mit anderen in Abhängigkeiten bzw. Kommunikation stehen.

3.4.2 Macro und Mikroarchitektur

Wenn durch das DDD entworfen wird, welche Microservices voraussichtlich benötigt werden, ist es sinnvoll einen Art Bauplan zu verfassen, welcher impliziert, an welche Regeln sich ein Microservice halten muss. Diese Regeln können wie bereits erwähnt auf globaler Ebene getroffen werden, d.h. sie gelten für alle Services (Makroarchitektur) oder sie gelten nur im Microservice selber (Microarchitektur). REWE Digital unterscheidet dabei

zwischen *Must*, *Should*, *Could*. D.h. es gibt Regeln, die Microservices erfüllen müssen wie z.B. das Kommunzieren über REST oder das Implementieren eine einheitlicher Authorisierung.[3] Andere Regeln dagegen sind viel mehr Richtlinien (should) oder komplett optional (could). Das Ziel ist stets, dass durch die makroarchitektonischen Entscheidungen nicht die Vorteile von Microservices beschnitten werden.[7][4].

Es gibt verschiedene Einflussfaktoren wie sich die Makroarchitektur für ein Unternehmen definiert: Zum einen empfiehlt es sich ein Gremium zu gründen, welches sich stetig mit den Regeln der Makroarchitektur auseinandersetzt, sie entsprechend erweitert, überarbeitet und die getroffenen Entscheidungen auch immer begründen kann.[7] Zum anderen besteht immer ein technischer Einfluss[7]:

- Gewählte Technologien müssen in die Infrastruktur des Unternehmens passen: Angenommen die Auslastung eines Microservices muss überwacht werden und dies wird firmenweit mit Tool A erledigt, dann wäre es sehr aufwendig, wenn der besagte Service nur eine Schnittstelle für Tool B anbietet und der nächste Service nur für Tool C. Dies würde einerseits sehr unübersichtlich werden und andererseits viel Aufwand bedeuten.

- Technologien sind immer von dem Personal abhängig: Gerade wenn Unternehmen klein bis mittelständig sind, emphielt es sich Technologien zu nutzen, die mehrere Entwickler beherrschen, um Inselwissen zu reduizieren.

- Ebenfalls können gezielt strategische Entscheidungen getroffen werden, z.B. wenn ein Unternehmen die Dateninfrastruktur zu einem Cloudanbieter auslagern möchte, hat dies entsprechende makroarchitekonisches Auswirkungen.

Basierend auf Wolff und Nadareishvili ist folgende Tabelle entstanden, welche einen Überblick darüber gibt, wie gängige Entscheidungspunkte einzuordnen sind. [7][4][3]

	Mikroarchitektur	Makroarchitektur
Programmiersprache	x	x
Datenbank	x	x
Look and Feel (UI)	x	x
Dokumentation	x	x
Datenformat		x
Kommunikationsprotokoll		x
Authentifizierung		x
Integrationtests		x
Autorisierung	x	
Unittests	x	
Continuous-Delivery-Pipeline	x	

In der Tabelle sieht man, dass gerade die ersten Punkten sehr von der Unternehmenskultur und den technischen sowie personellen Freiheiten abhängt. In der Theorie sollte die Programmiersprache sinnvoll für jeden Microservices gewählt werden, dennoch ergibt es auch Sinn einen Pool an Programmiersprachen auf Makroebene zu definieren, um Inselwissen zu reduzieren und nachhaltige Codequalität zu gewährleisten. Ähnliches gilt beispielsweise für die Wahl der Datenbank: Ist bereits eine globale Infrastruktur für Datenbank X geschaffen, sollte diese nicht ohne weiteres aufgebrochen werden, nur weil es technisch möglich ist.

Bei der Dokumentation sowie beim Look & Feel ist es sinnvoll, globale Richtlinien zu definieren, damit klar ist, wo bei jedem Microservices die Dokumentation zu finden ist oder wie ein User Interface grundsätzlich angeordnet und gestaltet werden soll. Dennoch können diese Punkte im Detail je nach Microservice abweichen.

Ein Kommunikationsprotokoll (z.B. REST) sowie Datenformate (z.B. JSON) sollten festgeschrieben werden.[3] [7] Als Grund wird zum einen das Vermeiden von technischen Mehraufwand angegeben, zum anderen sind Microservices zwar eigenständig deploybare Einheiten, dennoch sollten sie technisch zum Gesamtsystem passen und nicht dagegen arbeiten.

Während die Authentifzierung (um wen handelt es sich) einmalig festgelegt werden sollte, liegt die Überprüfung der Autorisierung (was darf der Benutzer) in jedem Microservice selbst. Die Alternative wäre, dass jede eingehende Anfrage noch einmal geprüft wird, was zu unnötig hohem Traffic und Verzögerungen führen würde.

Die hier erarbeitete Tabelle ist an dieser Stelle nicht als feststehendes Manfifest für alle Unternehmen zu verstehen, sondern als neutral betrachtete, sinnvolle Einordnung. Natürlich können architektonische Entscheidungen stark vom jeweiligen Anwendungszweck abhängen.

Die Tabelle zeigt lediglich allgemeine Beispiele und ist nicht als vollständig zu betrachten. Nicht aufgeführt ist beispielsweise der Umgang mit Konfigurationsdateien, Monitoring oder Logging. Dies liegt unter anderem daran, weil es auf Projekte oder von dem Microservice abhängt: Beim Monitoring könnte zum Beispiel global entschieden werden, *wo* Metriken abgelegt werden bzw. mit *welcher* Technologie gearbeitet wird. Aus microarchitektonischer Sicht könnten die Services selbst entscheiden, *was* gemessen wird.

Ebenso beim Deployment: Es gibt zahlreiche Methoden, um neue Updates bereitzustellen wie z.B. mittels Docker, Kubernetes oder indivudelle Installationsskripte.[7] Welche Technologie sich durchsetzt, muss anhand der Anforderung entschieden werden.

Aus den erarbeitenten Punkten lassen sich Vor- und Nachteile ableiten, zwischen denen abgewogen werden muss. Vorteile für microarchitektonische Entscheidungen sind ein sehr hohes Maß an Flexibilität und eine hohe Unabhängigkeit im Gesamtsystem,

was grundsätzlich das Ziel von Microservices ist. Dies wiederum kann dazu führen, dass Entwicklungsoverhead oder Inselwissen entsteht. Ebenfalls könnten Punkte wie z.B. das *Look & Feel* oder die *Dokumentation* darunter leiden.

Macroarchitektonische Entscheidungen haben zum Vorteil, dass es Regeln gibt, welche die Entwicklung vereinfachen sollen und gegebenfalls die Nachteile der Microarchitektur kompensieren können. Auf der anderen Seite schränken macroarchitekonische Entscheidungen ein. Zusätzlich müssen sie organisch, z.B durch ein extra dafür geschaffenes Gremium, durchgesetzt und werden.[7] Im Gesamten lässt sich daraus schließen, dass sehr genau abgewogen werden muss, welche Entscheidungen global oder individuell entschieden werden. Grundsätzlich gilt, dass jede Entscheidung begründbar sein muss.

3.5 Kommunikation

3.6 Authentifizierung und Authorisierung

I

4 Konzept

Das ist der Grundlagenteil

5 Implementierung

Das ist der Grundlagenteil

6 Fazit

Das ist der Grundlagenteil

Literatur

[1] Mark Fussel (msfussell). *Einführung in Microservices in Azure - Microsoft-Dokumentation*. 2017. URL: `https://docs.microsoft.com/de-de/azure/service-fabric/service-fabric-overview-microservices`.

[2] Ferdinand Birk. „Microservices - Eine State-of-the-Art Bestandsaufnahme und Abgrenzung zu SOA“. Universität Ulm, 2016.

[3] Sebastian Gauder. „A competitive food retail architecture with microservices“. 2019.

[4] Matt McLarty Irakli Nadareishvili Ronnie Mitra und Mike Amundsen. *Microservice Architecture - Aligning Principels, Practices, and Culture*. O Reilly, 2016.

[5] Dennis Peuser Nhiem Lu Gert Glatz. *Moving mountains – practical approaches for moving monolithic applications to Microservices*. University of Applied Science und Arts, Dortmund, Germany, adesso AG, Dortmund, Germany. 2017.

[6] Golo Roden. *Domain-driven Design erklärt*. 2016. URL: `https://www.heise.de/developer/artikel/Domain-driven-Design-erklaert-3130720.html?seite=all`.

[7] Eberhard Wolff. *Das Microservice Praxisbuch - Grundlagen, Konzepte und Rezepte*. dpunkt.verlag, 2018.

[8] Eberhard Wolff. *Flexible Software Architectures*. Leanpub, 2016.