Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Architektur einer Microservice-Anwendung und Deployment in OpenShift am Beispiel einer Partnerdatenbank

Inhaltsverzeichnis

zur

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Christoph Ruhsam, BSc

Betreuer:	DI (FH) Thomas Reidinger, 3 Banken IT GmbH
Begutachter:	FH-Prof. DI Dr. Herwig Mayr

November, 2019

Inhaltsverzeichnis

Kapitel 1

Einleitung ... 5.5 Seiten

1.1 Motivation zur Architektur von Microservices ... 1.5 Seiten

1.2 Motivation zum Einsatz von Cloudtechnologien ... 1.5 Seiten

1.3 Zielsetzung der Implementierung der Partnerdatenbank ... 1 Seite

1.4 Ziel des Deployments der Partnerdatenbank in OpenShift ... 1 Seite

1.5 Leitfaden und Gliederung der Schrift ... 0.5 Seiten

Kapitel 2

Serviceorientierte Architektur und Microservices

Das folgende Kapitel behandelt Microservices, ihre Vor- und Nachteile und die Abgrenzung zu serviceorientierten Architekturen, sowie die Unterschiede zu monolithischen Architekturen.

2.1 Definition und Abgrenzung

Im Folgenden wird die Definition von Microservices und die Abgrenzung zu serviceorientierten Architekturen beschrieben.

2.1.1 Definition von Microservices

Microservice-Architekturen sind ein Ansatz, um große Softwarearchitekturen in kleine, konsistente, voneinander klar abgegrenzte Services zu zerlegen. Diese Services sind isoliert und kommunizieren miteinander. [Posta, 2016, Kapitel 1]

Microservices werden typischerweise von kleinen Teams implementiert, gebaut und deployt. Diese kleinen Services sind so autonom, dass das Team, welches für den Service zuständig ist, die Implementierungsdetails ändern kann, ohne großen Einfluss auf das gesamte restliche System zu nehmen. Jedes Team ist selbst für sein Service verantwortlich. Das Team muss für die Aufgabenstellung die richtige Technologie wählen, das Service deployen und managen und etwaige Fehler beheben. Bei Microservice-Architekturen kann die Abgrenzung der verschiedenen Services ganz klar definiert werden. [Posta, 2016, Kapitel 1] Dies hilft, um [Posta, 2016, Kapitel 1]

1. die Logik des Services einfach zu verstehen, ohne den Kontext der gesamten Applikation kennen zu müssen.
2. den Service schnell lokal bauen und ausführen zu können.
3. die richtige Technologie für ein Problem zu nützen.
4. den Service einfach und schnell testen zu können, ohne die gesamte Applikation dabei hochfahren zu müssen.
5. schnellere Releasezeiten zu ermöglichen.
6. schnellere horizontale Skalierung zu ermöglichen.
7. die Belastbarkeit des Systems zu verbessern.

Die oben angeführten Punkte werden im Folgenden näher ausgeführt [Posta, 2016, Kapitel 1]:

Punkt 1: Jedes einzelne Microservice sollte nur einen Teil der gesamten Anwendung ausmachen. Ein Microservice deckt meist nur einen Geschäftsfall der gesamten Businesslogik ab. Dieser Geschäftsfall ist für sich einfacher zu verstehen, als das gesamte System. Kommt ein neuer Entwickler in das Team, muss er sich nur in den Code des Microservice, an dem er arbeitet, einlesen und muss nicht das gesamte System verstehen.

Punkt 2: Muss bei einem neuen Feature die gesamte Anwendung gebaut, deployt und ausgeführt werden, ist dies sehr aufwändig und kostet sehr viel Zeit. Ein Microservice hingegen steht für sich selbst und kann einfach lokal gestartet und getestet werden. Dies erleichtert dem Entwickler die Arbeit und spart Zeit.

Punkt 3: Meist werden für bestimmte Geschäftsfälle verschiedene Technologien benötigt. Ist die gesamte Businesslogik in einem Monolithen abgebildet, kann lediglich eine Technologie eingesetzt werden. Ist aber z.B. für einen bestimmten Geschäftsfall Machine Learning nötig, ist wahrscheinlich Java die falsche Lösung und man setzt lieber auf Python. Microservices sind unabhängig voneinander und kommunizieren meist durch REST oder SOAP miteinander. Dadurch kann in jedem Service die gewünschte Technologie eingesetzt werden.

Punkt 4: Ähnlich zu **Punkt 2**, wo das Bauen und Ausführen eines Microservice beschrieben wird, kann ein Service auch unabhängig vom gesamten System getestet werden.

Punkt 5: Heutzutage müssen neue Features schnell ausgeliefert werden, ansonsten ist womöglich der Konkurrent schneller und man verliert Kunden. Muss bei einem neuen Release das gesamte System neu deployt werden, sind damit verschiedenste Risiken verbunden. Wird z.B. ein fehlerhafter Code deployt, stürzt der gesamte Monolith ab. Bei Microservices ist in dieser Zeit lediglich das neu deployte Service nicht verfügbar.

Punkt 6: Hat man ein monolithisches System und möchte einen neuen Geschäftsfall implementieren, muss dieser in das bestehende System eingeflochten werden, was dazu führen kann, dass ein bestehender Code nicht mehr so funktioniert wie gewollt. Bei einer Microservice-Architektur wird ein neues Service für diesen Geschäftsfall angelegt und bestehender Code nicht oder kaum für Aufrufe des neuen Service geändert.

Punkt 7: Fällt ein Service aus, heißt das nicht, dass das gesamte System nicht mehr verfügbar ist. Teile des Systems können meist ohne Probleme weiterverwendet werden.

2.1.2 Abgrenzung serviceorientierter Architekturen und Microservices

Microservices und serviceorientierte Architekturen gehören beide zu den service-basierten Architekturen. Das heißt, dass beide Architekturen einen großen Schwerpunkt auf Services als primäre Architekturkomponenten legen. Obwohl Microservices und serviceorientierte Architekturen sehr unterschiedliche Muster aufweisen, haben sie auch vieles gemeinsam [Richards, 2016, Kapitel 1].

Alle serviceorientierten Architekturen sind verteilte Architekturen. Die Servicekomponenten werden über Remote-Protokolle, wie z.B. Representational State Transfer (REST), Simple Object Access Protocol (SOAP) oder Java Message Service (JMS), angesprochen. Beide bieten viele Vorteile im Vergleich zu monolithischen und schichten-basierten Architekturen. Sie bieten höhere Skalierbarkeit, losere Kopplung zwischen den Services und bessere Kontrolle über die Entwicklung, Tests und das Deployment. Komponenten in einer verteilten Anwendung tendieren mehr dazu in sich verschlossen zu sein und bieten daher besseres Änderungsmanagement und einfachere Wartung, was zu mehr Flexibilität und zu robusteren Anwendungen führt. Diese Architekturen führen auch zu loser gekoppelten und modulareren Anwendungen. Modularität

ist das Kapseln einzelner Teile der Anwendung in abgeschlossene Services, die dadurch individuell designed, entwickelt, getestet und deployt werden können, ohne großen Einfluss auf andere Services zu nehmen [Richards, 2016, Kapitel 1].

Sowohl bei Microservices, als auch bei serviceorientierten Architekturen müssen *Service Contracts* getroffen werden. Service Contracts werden zwischen einem Service und einem Servicekonsument (Client) vereinbart. Dadurch wird das Format der eingehenden und ausgehenden Daten spezifiziert (z.B. XML, JavaScript Object Notation [JSON], Java Object, etc.) [Richards, 2016, Kapitel 1].

Microservice-Teams bestehen meist aus maximal fünf bis sieben Personen. Das Team muss die Entwicklung, das Deployment und den Betrieb des Service alleine bewältigen. Je größer das Team, desto höher ist der Kommunikationsaufwand. Jedes Teammitglied sollte die gesamte Codebasis überblicken und warten können. Zu große Teams sind meist ausschlaggebend dafür, dass das Service aufgesplittet werden sollte [Richards, 2016, Kapitel 2].

Serviceorientierte Architekturen sind eher für große, komplexe Enterprise-Systeme geeignet, die eine Interaktion mit vielen heterogenen Applikationen und Services benötigen. Für Anwendungen, die viele Komponenten benötigen und von mehreren Services geteilt werden, sind serviceorientierte Architekturen passender. Für Applikationen mit einem wohldefinierten Arbeitsablauf und wenig gemeinsam geteilten Komponenten, wie z.B. Security-Komponenten, sind Microservice-Architekturen sinnvoll. Microservice-Architekturen sind besser geeignet für kleine, wohl definierte web-basierte Systeme, wie für groß skalierbare Enterprise-Systeme. Die fehlende Middleware ist ein Faktor, weshalb Microservice-Architekturen nicht oder schlecht für komplexe Businessanwendungen geeignet sind. Microservices sollten dort eingesetzt werden, wo sich die Businesslogik auf kleine abstrahierte Geschäftsfälle herunterbrechen lässt [Richards, 2016, Kapitel 2].

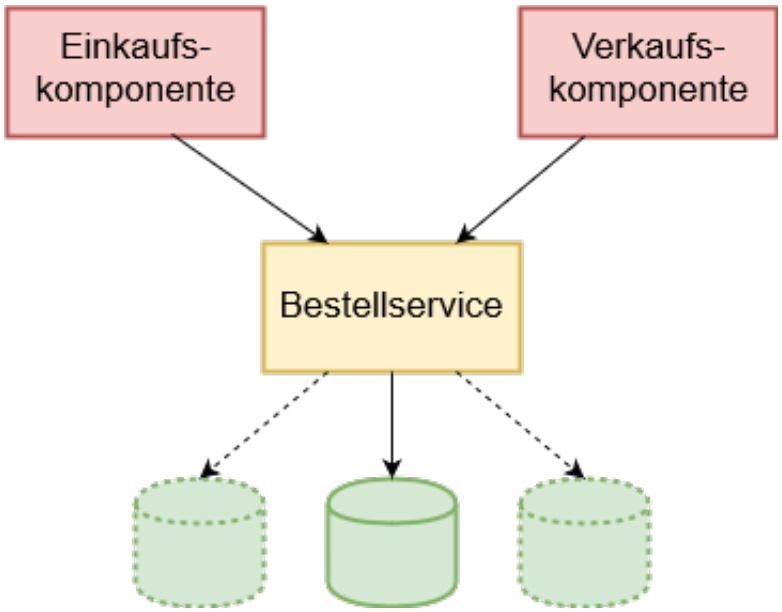

Abbildung 2.1: Serviceorientierte Architektur - Bestellservice

[Richards, 2016, Kapitel 3]

Microservices und serviceorientierte Architekturen sind auch bezüglich Teilen von Komponenten sehr unterschiedlich. Serviceorientierte Architekturen basieren auf dem Prinzip *share-as-much-as-possible*, wohingegen Microservices auf dem Prinzip *share-as-little-as-possible* basieren. Angenommen ein großes Warenhaus benötigt zum Verwalten der Einkäufe und Verkäufe jeweils ein

Bestell-Service. Bei serviceorientierten Architekturen, wie in Abbildung 2.1 zu sehen, würde man dazu das Bestell-Service einmal implementieren und die Einkaufs- und Verkaufskomponente teilen sich dieses Service. Das Bestell-Service selbst muss in diesem Fall wissen, welche Aktionen es bei den verschiedenen Anfragen ausführen muss. Obwohl das *share-as-much-as-possible*-Prinzip das Problem der Codeduplizierung löst, ist damit eine enge Kopplung von Businesskomponenten verbunden und erhöht das Risiko von Änderungen. Angenommen man ändert den Code im Business-Service, ist diese Änderung schwer zu testen, da das Service global verfügbar ist und je nach Aufruf andere Aktionen auslöst [Richards, 2016, Kapitel 3].

Microservices basieren auf dem Konzept des *share-as-little-as-possible* und verstärken dadurch das domain-driven Design Konzept *Bounded Context*. Microservices sind lose gekoppelt und agieren daher als einzelne geschlossene Einheit mit wenig Abhängigkeiten. Es gibt lediglich ein wohldefiniertes Interface und wohldefinierte Vereinbarungen nach außen. Realistischerweise wird es immer gemeinsam geteilte Services geben, auch in Microservice-Architekturen, z.B. Sicherheits- oder Infrastrukturservices. Serviceorientierte Architekturen maximieren das Teilen von Komponenten, wohingegen Microservice-Architekturen dieses durch den klar definierten Bounded Context minimieren [Richards, 2016, Kapitel 3].

Es gibt viele Vorteile, die das Verstärken des Bounded Context bringt. Die Wartung der Services wird wegen der geringeren Abhängigkeiten weitaus einfacher. Dies erleichtert auch das Ändern und Weiterentwickeln des Service. Auch das Deployment gestaltet sich dadurch um einiges einfacher, da weniger Code zu deployen ist und es weniger Risiken gibt, andere Bereiche des gesamten Systems zu brechen. Dies fördert auch die Robustheit des gesamten Systems [Richards, 2016, Kapitel 4].

2.2 Vergleich zu monolithischer Architektur

Microservices und Monolithen sind sehr unterschiedliche Designstrategien, um eine Anwendung zu entwickeln. Beide Strategien haben ihre Vor- und Nachteile. Die Strategie sollte deshalb mit Bedacht gewählt werden. Die monolithische Architektur ist derzeit noch der Standardweg eine Applikation zu entwickeln [Tilkov, 2015].

Im Folgenden werden die wichtigsten Unterschiede monolithischer und Microservice-Architekturen beschrieben [Jangla, 2018, Kapitel 3]

2.2.1 Wartung

Microservices sind viel einfacher zu warten, da sich die Komplexität eines Services in Grenzen hält. Microservices sind modulare und unabhängige Services. Neue Entwickler können sich in die Codebasis schneller einlesen und schneller neue Features implementieren, als bei monolithischen Architekturen.

Ein Monolith besteht aus einer Codebasis, wo alle Geschäftsfälle abgedeckt sind. Wird ein neues Feature hinzugefügt, wird es dabei in die bestehende Codebasis eingewebt. Dies erhöht die Komplexität des Codes und erschwert die Entwicklung und Wartung. Neue Entwickler müssen sich in die gesamte Codebasis und in den gesamten Workflow einlesen, um neue Features entwickeln zu können.

2.2.2 Deployment

Ein kontinuierliches Deployment gestaltet sich bei Monolithen sehr schwierig, da der Monolith immer größer und komplexer wird. Zudem muss immer der gesamte Monolith deployt werden, auch wenn lediglich wenig Codestücke geändert wurden. Dies ist natürlich sehr zeitaufwändig, aber auch gefährlich. Wird ein fehlerhafter Code deployt, bringt dieser den gesamten Monolith zum Stillstand.

Bei Microservices hingegen wird die Codebasis in kleine, überschaubare Services unterteilt, die einzeln und jederzeit schnell deployt werden können. Wird bei Microservice-Architekturen ein fehlerhafter Code deployt, ist es möglich, dass andere Teile des gesamten Systems noch ohne Probleme laufen.

2.2.3 Tests

Auch beim Testen eines Monolithen werden die große Codebasis und die vielen verschiedenen Anwendungsfälle zum Problem. Das Testen aller Szenarien bei einer derart großen Codebasis wird dabei schnell unübersichtlich.

Bei Microservices kann durch den Bounded Context jede Komponente individuell getestet werden. Dies erleichtert das Testen eines jeden Services ungemein und verringert dazu auch die Fehleranfälligkeit eines Systems.

2.2.4 Startup-Zeit

Bei monolithischen Architekturen wird der gesamte Monolith hochgefahren. Je komplexer und größer dieser wird, desto mehr steigt die Startzeit.

Die Startzeit bei Microservices ist viel schneller, da jedes Service einzeln hochgefahren werden kann. Werden alle Services gemeinsam gestartet, bringt dies bezüglich der Startzeit wieder wenig.

2.2.5 Technologie

Ein Monolith wird in einer Programmiersprache entwickelt und greift meist nur auf eine Datenbank zu. Neue Technologien können nicht, bzw. nur sehr umständlich hinzugefügt werden. Die Einarbeitungszeit gestaltet sich jedoch bezüglich Technologie einfacher, da nur eine Sprache zum Einsatz kommt und neue Entwickler nur diese beherrschen müssen.

Mit Microservices können Entwickler die Technologie jedes Services selbst wählen. Die Services greifen oft auch auf verschiedene Datenbanken zu und Entwickler können auch die Wahl der Datenbank selbst wählen. Da Microservices so klein und überschaubar sind, können diese schnell ersetzt werden und es kann dadurch immer die aktuellste Version oder Technologie verwendet werden.

2.2.6 Skalierbarkeit

Microservices können einfach skaliert werden. Platform as a Service Plattformen, wie OpenShift, bieten eine einfache Möglichkeit, Services je nach Bedarf hoch- und runterskalieren zu können.

Einen komplexen Monolithen zu skalieren, gestaltet sich sehr schwierig. Zum Beispiel wird ein kontinuierliches Deployment mit vielen Entwicklern und einer sehr großen Codebasis sehr kompliziert.

2.3 Charakteristiken

Im Folgenden werden einige Charakteristiken von Microservices beschrieben, die auch für serviceorientierte Architekturen gelten [V, 2016]:

2.3.1 Service-Vereinbarung

Microservices werden durch wohldefinierte Service-Vereinbarungen beschrieben. Zum Datenaustausch kann z.B. REST oder SOAP verwendet werden. Als Markup-Sprache kann JSON, XML oder auch YAML eingesetzt werden. In der Welt der Microservices wird meist die Kombination von REST und JSON eingesetzt. Es gibt aber mehrere Möglichkeiten zur Definition von Service-Kontrakten. JSON Schema, WADL, Swagger und RAML sind nur ein paar wenige Beispiele.

2.3.2 Lose Kopplung

Microservices sind unabhängig und lose gekoppelt. Meist akzeptieren Microservices ein Event als Input und reagieren darauf erneut mit einem Event. Nachrichten, HTTP und REST werden häufig zur Interaktion zwischen den einzelnen Services eingesetzt. Nachrichten-basierte Endpunkte bieten dabei ein höheres Level an Entkopplung.

2.3.3 Service-Abstraktion

Service-Abstraktion bei Microservices ist nicht nur die abstrahierte Realisierung des Service, sondern bietet auch eine komplette Abstraktion von allen Bibliotheken und Umgebungsdetails.

2.3.4 Wiederverwendung

Microservices sind wiederverwendbare Business-Services. Sie werden von mobilen Geräten, Desktop-Anwendungen und auch anderen Microservices verwendet.

2.3.5 Zustandslosigkeit

Wohl-designte Microservices sind zustandslos und teilen sich nichts. Wird ein gemeinsamer Zustand in verschiedenen Services benötigt, wird dieser meist in einer Datenbank abgebildet.

2.3.6 Interoperabilität

Services sind durch Standardprotokolle und Nachrichtenaustausch-Standards interoperabel. Nachrichtenaustausch, HTTP und REST werden zum Nachrichtentransport verwendet. Eine Kombination aus REST und JSON ist die populärste Methode zum Entwickeln von interoperablen Services. Wird weitere Optimierung benötigt, kommen auch andere Protokolle, wie RabbitMQ,

Avro, Zero MQ oder Protocol Buffers zum Einsatz. Der Einsatz dieser Protokolle kann dabei aber auch die Interoperabilität der Services einschränken.

2.3.7 Zusammensetzung

Microservices sind zusammensetzbar. Dies wird durch Service Orchestrierung und Service Choreografie erreicht. Service Orchestrierung und Service Choreografie sind zwei Designvarianten von Microservices. Auf diese wird im nächsten Abschnitt näher eingegangen.

2.4 Varianten

Im Folgenden werden die beiden Varianten *Service Orchestrierung* und *Service Choreografie* zum Design von Microservices vorgestellt.

2.4.1 Service Orchestrierung

Bei der Service Orchestrierung gibt es ein zentrales Kompositionsservice, dass das entsprechende Microservice aufruft und von diesem die Antwort erwartet. Diese Nachrichten können mittels HTTP oder Socketaufrufen durchgeführt werden. Wichtig hier ist, dass dieses Muster sowohl synchron, als auch asynchron durchgeführt werden kann. Das aufgerufene Microservice muss also nicht sofort antworten, sondern kann auch später mit der Antwort ein Event im Kompositionsservice auslösen. In diesem Muster reden die Microservices nicht miteinander, sondern nur mit dem Kompositionsservice. Dabei muss jedoch nicht zu jedem Microservice dasselbe Protokoll verwendet werden, z.B. redet das Kompositionsservice mit Service A über HTTP und mit Service B über das XML/RPC Protokoll [Sharma, 2017, Kapitel 3].

Wie in Abbildung 2.2 zu sehen, wird dabei das Kompositionsservice zur zentralen Autorität. Hier beginnt auch die gesamte Logik, was schlechtes Design für Microservices bedeutet. Die gesamte Logik im Kompositionsservice zu haben, führt zu einer Fülle von Abhängigkeiten zwischen den Services, z.B. kann die Logik sein, dass wenn Service A antwortet, Service B danach aufgerufen werden muss. Dies führt zu Abhängigkeiten zwischen Service A und Service B [Sharma, 2017, Kapitel 3].

Da das Kompositionsservice der zentrale Punkt der Anwendung ist, ist dieser auch die Schwachstelle. Dieses Service muss verfügbar sein, ansonsten steht die gesamte Anwendung still. Das Kompositionsservice ist der *Single-Point-of-Failure* in dieser Architektur. Dies kann aber durch Tools und cloud-basierte Lösungen, wie Skalieren oder Clustering gelöst werden. Zusätzlich ist das Orchestrierungsmuster schwer zu implementieren. Hat man einen Entscheidungspunkt, wird es schwer, dem verteilten Design zu folgen [Sharma, 2017, Kapitel 3].

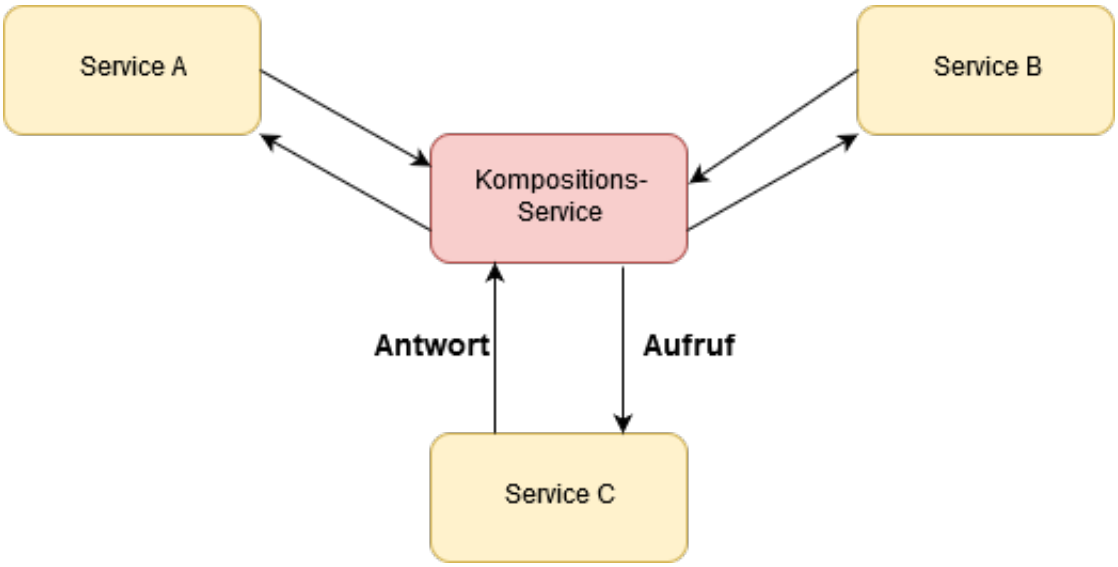

Abbildung 2.2: Varianten - Service Orchestrierung

[Sharma, 2017, Kapitel 3]

2.4.2 Service Choreografie

Anders als bei der Service Orchestrierung, ist bei der Service Choreographie jedes Service in Koordination mit den anderen Services auf einen bestimmten Anwendungsfall zugeschnitten. Service Orchestrierung repräsentiert die Kontrolle von einem einzelnen Standpunkt aus. Bei der Choreografie wird jedes Service in die Interaktion mit eingebunden [Sharma, 2017, Kapitel 3].

Wie in Abbildung 2.3 zu sehen, redet bei der Service Choreografie jedes Service nach Abschluss des Tasks mit einem anderen Service, um die nächste Aufgabe zu triggern. Dies kann in den Mustern *one-to-one*, *one-to-many* oder *many-to-many* geschehen. Dies kann ebenfalls synchron oder asynchron durchgeführt werden. Idealerweise gibt es ein globales Protokoll, über das die Services miteinander kommunizieren. Meistens werden diese Aufrufe asynchron durchgeführt. Da es kein zentrales Service gibt, löst jedes Service ein Event nach Abschluss seines Tasks aus. Dies macht die Services unabhängig von einander und fördert die lose Kopplung. Andere Services registrieren sich auf dieses Event und reagieren darauf, wenn es ausgelöst wird [Sharma, 2017, Kapitel 3].

Durch nachrichtenorientierte Kommunikation kann das Choreografiemuster sehr nützlich werden. Zwischen den Services wird eine Nachrichtenschlange implementiert. Dadurch wird das Loslösen, Hinzufügen und Entfernen von Services sehr einfach gelöst. Auch die ganze Reihe an Abhängigkeiten, die oft bei Choreografiemustern entstehen, wird dadurch gelöst. Service Choreografie erhöht die Komplexität eines Systems. Jedes Service erzeugt und verarbeitet Nachrichten zur Laufzeit. Dadurch kann der exakte Stand einer Transaktion nicht herausgefunden werden [Sharma, 2017, Kapitel 3].

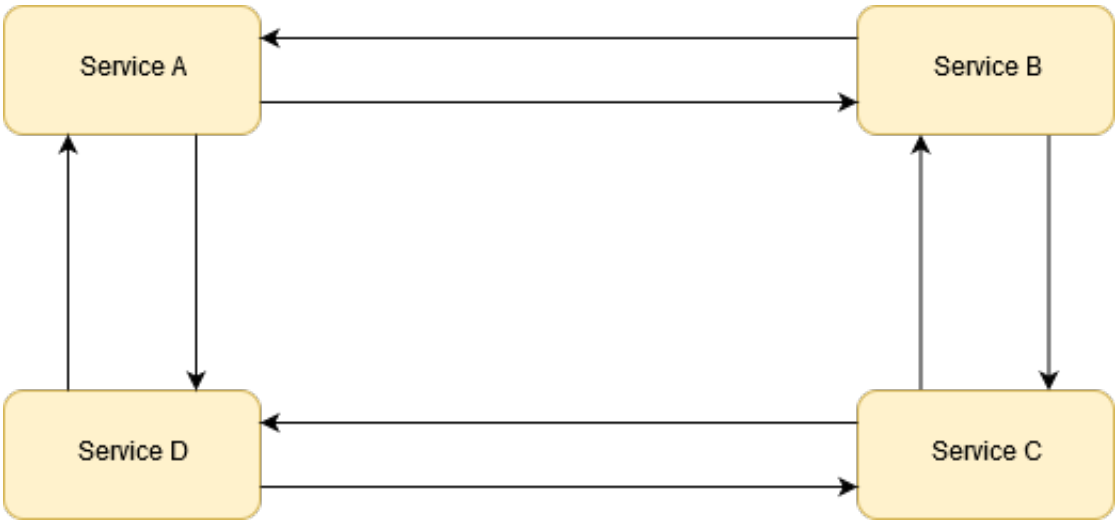

Abbildung 2.3: Varianten - Service Choreografie

[Sharma, 2017, Kapitel 3]

2.5 Vor- und Nachteile

Im folgenden Abschnitt werden die Vor- und Nachteile von Microservices behandelt.

2.5.1 Vorteile

Technologieunabhängigkeit

Microservices werden als eigenständige Einheit betrachtet. Dadurch können die einzelnen Services verschiedene Technologien verwenden. Greift ein Service auf eine Datenbank zu und ein anderes Service führt Shell-Skripte aus, beeinflussen sich die beiden Services nicht und können jederzeit ausgetauscht werden, ohne Einfluss auf andere Services zu nehmen. Performance ist in diesem Zusammenhang ebenfalls sehr wichtig. Wird eine Technologie zu langsam, kann sie ohne Probleme durch eine neue ersetzt werden [Sam Newman, 2015, Kapitel 1].

Skalierung

Wie erwähnt, muss bei Monolithen die gesamte Software als Ganzes deployt und skaliert werden. Dies bedeutet viel Overhead. Bei Microservices können einzelne Teile hoch- und runterskaliert werden. Cloud-basierte Technologien, wie z.B. OpenShift, bieten diese Möglichkeit an. Es kann aber auch ein Microservice auf einer besseren Hardware laufen. Die anderen Services bleiben auf der billigeren Hardware. Monolithen müssten als Ganzes auf der besseren Hardware laufen. Dies verringert die Kosten der Skalierung bei Microservices drastisch [Sam Newman, 2015, Kapitel 1].

Austauschbarkeit

Bei Microservices ist es sehr einfach, den bestehenden Code zu ändern oder auszutauschen. Da diese unabhängig sind, sind keine anderen Teile des Systems betroffen. Bei Monolithen stellt dies

oft ein großes Risiko dar. Der Code ist oft sehr verwoben und kleine Änderungen haben große Auswirkungen [Sam Newman, 2015, Kapitel 1].

Entwicklerteam

Entwickler greifen bei monolithischen Architekturen oft auf die selben Teile des Codes zu. Bei Versionierungsprogrammen, wie GIT oder Subversion, führt dies oft zu Konflikten. Microservices können getrennt voneinander bearbeitet und der Code ohne Probleme zusammengefügt werden [Sam Newman, 2015, Kapitel 1].

2.5.2 Nachteile

Hohe Latenzzeiten

Microservices reden über das Netzwerk miteinander. Dieser Kommunikationsweg ist natürlich viel langsamer, als wenn diese in derselben JVM miteinander reden würden. Dies führt zu hohen Latenzzeiten. Diese müssen bei der Architektur von Microservices berücksichtigt werden [Sam Newman, 2015, Kapitel 1].

Wiederverwendbarkeit von Code

Microservices sind unabhängige und eigenständige Einheiten. Sie haben im besten Fall keine Abhängigkeiten zueinander. Dadurch verringert sich die Wiederverwendbarkeit von Code. Dies führt auch zu einer Codeverdopplung. Wird dasselbe Codestück in zwei verschiedenen Services benötigt, muss es in beiden vorhanden sein [Sam Newman, 2015, Kapitel 1].

Deployment

Jedes Service muss einzeln deployt werden. Dazu wird ein eigener Deployment-Prozess pro Service benötigt. Dies bedeutet Aufwand für den Entwickler und es muss eine eigene Infrastruktur geschaffen werden, was zusätzlich Kosten generiert [Sam Newman, 2015, Kapitel 1].

Kapitel 3

Containerisierung mit Docker ... 6 Seiten

3.1 Docker ... 4 Seiten

3.2 Notwendigkeit von Containerisierung ... 2 Seiten

Kapitel 4

OpenShift ... 10.5 Seiten

4.1 Beschreibung von OpenShift ... 2 Seiten

4.2 Komponenten von Kubernetes ... 3.5 Seiten

4.3 Die OpenShift-Umgebung ... 3 Seiten

4.4 Fabric8 ... 2 Seiten

Kapitel 5

Partnerdatenbank ... 14.5 Seiten

5.1 Grundaufbau und Zweck der Partnerdatenbank ... 2.5 Seiten

5.2 Backend-Beschreibung ... 4 Seiten

5.3 Beschreibung der einzelnen Services ... 4 Seiten

5.4 Frontend-Beschreibung ... 4 Seiten

Kapitel 6

Design der Partnerdatenbank ... 11 Seiten

6.1 Microservice-Architektur ... 3 Seiten

6.2 Beschreibung der verwendeten Microservice-Technologien ... 6 Seiten

6.3 Design in OpenShift .. 2 Seiten

Kapitel 7

Implementierung der Partnerdatenbank ... 17 Seiten

7.1 Microservice-Architektur ... 2 Seiten

7.2 Automatisierte Test-, Build- und Deployment-Pipelines mit Jenkins ... 3 Seiten

7.3 Fehlerbehandlung mit Microprofile ... 2 Seiten

7.4 REST-Schnittstellenbeschreibung mit Swagger ... 2 Seiten

7.5 Tracing mit Jaeger ... 1 Seite

7.6 Einsatz von Docker zur Containerisierung der Anwendung ... 1 Seite

7.7 Konfiguration und Deployment-Deskriptoren von OpenShift ... 4 Seiten

7.8 Deployment in OpenShift mit Fabric8 ... 2 Seiten

Kapitel 8

Evaluierung der Anwendung ... 7 Seiten

8.1 Evaluierung des Frontends ... 1.5 Seiten

8.2 Whitebox-Tests ... 2 Seiten

8.3 Blackbox-Tests ... 1.5 Seiten

8.4 Architekturevaluierung ... 2 Seiten

Kapitel 9

Zusammenfassung ... 2-3 Seiten

9.1 Resümee

9.2 Ausblick

Quellenverzeichnis

Literatur

[Jangla, 2018] Kinnary Jangla. *Accelerating Development Velocity Using Docker*. Apress, Berkeley, CA, 2018.

[Posta, 2016] Christian Posta. *Microservices for Java Developers*. O'Reilly Media Inc., 2016.

[Richards, 2016] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media Inc., 2016.

[Sam Newman, 2015] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2015.

[Sharma, 2017] Umesh Ram Sharma. *Practical Microservices*. Packt Publishing Limited, 2017.

[V, 2016] Rajesh R V. *Spring Microservices*. Packt Publishing Limited, 2016.

Online-Quellen

[Tilkov, 2015] Stefan Tilkov. *Don't start with a monolith*. 2015. URL: https://martinfowler.com/articles/dont-start-monolith.html (besucht am 02. 01. 2020).