Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Wieso, Weshalb, Warum?

1.2 Zielsetzung dieser Arbeit

These: Mittels Saga-Pattern implementierte lang andauernde Transaktionen (LLT) ermöglichen (k)eine ausfallsichere Konsistenz in Microservicesystemen.
Leitfragen: Wie kann mithilfe von BASE-Eigenschaft und Softstate ausfallsichere Konsistenz hergestellt werden?
(Wie) kann der 2 Phasencommit in eine Saga-Architektur integriert werden?

1.3 Aufbau dieser Arbeit

- Kapitel 1: Theoretische Grundlagen: In diesem Kapitel sollen dem Leser die Grundlagen vermittelt werden, die er benötigt, um den Rest der Arbeit zu verstehen. Dazu gehören Begriffe und Konzepte aus der Welt der Microservices.
- Kapitel 2: Methodik: Ziel des Kapitels: Entwurf, Implementierung, Messen und Bewertung des Systems. Fokus auf Saga-Pattern, Konsistenz, Ausfallsicherheit -> Der Systementwurf und der Bewertungsprozess aus diesem Kapitel sollen in Kapitel 3 umgesetzt werden.
  - Bewertungskriterien
  - Entwurf und Implementierung eines reinen Saga-Systems
  - Abänderung der Implementierung durch
    - Saga-Werkzeuge (Checkpoints, Recovery Mechanismen)
    - 2 Phasencommit
- Kapitel 3: Ergebnisse aus Kapitel 2: Die Ergebnisse dieses Kapitels sollen in Kapitel 4 verwendet werden, um die anfängliche These und die Leitfragen zu beantworten.
- Kapitel 4: Diskussion - Bewerten der Ergebnisse in Bezug auf die These und der Leitfragen; Beantwortung der These und der Leitfragen

2 Grundlagen

2.1 Grundlagen

Um das in dieser Arbeit betrachtete Saga-Pattern zu verstehen, sollen zuerst einige Grundlagen erläutert werden. Besonders die im Titel der Arbeit enthaltenen Begriffe *System* und *Konsistenz* sollen in diesem Abschnitt erläutert werden.

2.1.1 System allgemein

Ein System beschreibt einen abgegrenzten Bereich der objektiven Realität. Außerhalb dieses Bereichs liegt die Umgebung, die somit nicht zum System gehört. Zwischen des Systems und seiner Umgebung befindet sich der Systemrand.

2.1.2 System in der Softwareentwicklung

In der Softwareentwicklung besteht ein System aus einer Menge miteinander interagierenden Softwarekomponenten. Diese Komponenten arbeiten an einem gemeinsamen Ziel. Neben der Software und deren Quellcode gehören auch Nutzerhandbücher, Tests, Bestandteile für die Instandhaltung sowie Spezifikationen und Konzepte zum System.

2.1.3 Zustand von Systemen

Ein Softwaresystem befindet sich zu jedem Zeitpunkt in einem Zustand. Der Wechsel eines Zustands ist die Folge von Nutzerinteraktionen und festgelegten Routinen. Damit das System reibungslos funktionieren kann, darf es nur zwischen gültigen Zuständen wechseln.

2.1.4 Sichten auf Systeme

Softwaresysteme können unter verschiedenen Sichten betrachtet werden. TODO
- mathematische Sicht: Endlicher Automat - Endliche Menge von Zuständen - Endliches Alphabet - Übergangsfunktion von Zustand x Alphabet -> Zustand - Startzustand - endliche Menge von Endzuständen

2.1.5 Transaktionen

Ein Übergang von einem Zustand in den nächsten Zustand im Modell des Zustandsautomats ist eine einzige Operation. Häufig äußert sich eine solche Operation jedoch in einer Folge von auszuführenden Operationen. Eine solche Folge von Operationen wird als Transaktion bezeichnet. Teil einer Transaktion kann beispielsweise eine oder mehrere

Datenbankoperationen oder der Aufruf einer externen Schnittstelle sein. Transaktionen können auf verschiedenen Ebenen betrachtet werden. Auf Datenbankebene werden Transaktionen anders umgesetzt als auf Ebene der Geschäftslogik.

2.1.6 ACID Eigenschaft von Transaktionen

ACID ist ein Akronym, welches häufig verwendet wird, um Eigenschaften auszudrücken, die eine Transaktion auf Datenbankebene aufweisen sollte. Die Buchstaben stehen für *A*tomicity (Atomarität), *C*onsistency (Konsistenz), *I*solation (Isolation) und *D*urability (Dauerhaftigkeit).
Atomarität beschreibt hier das Alles-Oder-Nichts-Prinzip. Eine Transaktion, also eine Folge von Datenbankoperationen, muss entweder alle Operationen ausführen oder in einem Fehlerfall alle Änderungen auf den ursprünglichen Zustand zurückrollen. Als illustratives Beispiel soll das Szenario einer Überweisung dienen. Eine Überweisung besteht aus dem Erhöhen des Kontostands des Debitors und dem Verringern des Kontostands des Kreditors. Eine solche Transaktion kann fehlschlagen. Dabei ist sicherzustellen, dass die initialen Kontostände wiederhergestellt werden. Wird die Atomarität nicht gewährleistet, könnte sich die Gesamtsumme des im System enthaltenen Systems verändern, Geld geht also verloren oder Geld wird aus dem Nichts generiert. Das ist natürlich nicht akzeptabel.
Konsistenz lässt sich mit widerspruchsfrei übersetzen. War das System vor Ausführung der Transaktion widerspruchsfrei, also konsistent, dann ist es das nach Ausführung der Transaktion ebenso. Um dies zu gewährleisten, sind verschiedene Mechanismen im Datenbanksystem im Einsatz. Dies werden im folgenden Abschnitt beschrieben.
Isolation stellt sicher, dass sich parallel in Ausführung befindliche Transaktionen nicht gegenseitig beeinflussen können. Das bedeutet, dass eine Transaktion Teile der Datenbasis sperren können muss. Eine solche Sperre führt jedoch zu blockierendem Verhalten und schränkt somit die nebenläufige Benutzung der Datenbank ein.
Dauerhaftigkeit von Transaktionen muss gewährleistet sein, damit bei einem Ausfall Transaktionsdaten nicht verloren gehen. Um Dauerhaftigkeit zu erreichen, wird ein Transaktionslog gefährt, welches im Ausfall verwendet werden kann, um die verlorenen Transaktionen nachzuspielen.

2.1.7 Konsistenz auf Datenbankebene

Auf Datenbankebene wird die Konsistenz durch drei Mechanismen sichergestellt: Bereichsintegrität, Entitätsintegrität, Referentielle Integrität und logische Konsistenz.
Die *Bereichsintegrität* stellt lediglich sicher, dass die Werte einer Spalte in einem definierten Wertebereich befinden.
Die *Entitätsintegrität* beschreibt die Eindeutigkeit eines Datensatzes und somit Fähigkeit, diesen Datensatz mittels Primärschlüssel eindeutig identifizieren zu können.
Die *Referentielle Integrität* gewährleistet, dass Relationen zwischen Datensätzen nicht auf nicht existierende Datensätze zeigen können. Ist ein Fremdschlüssel gesetzt, so muss der referenzierte Datensatz existieren.
Die in der Datenbank enthaltenen Daten drücken einen Sachverhalt aus. Die semantische Bedeutung dieser Daten kann nicht durch das Datenbanksystem sichergestellt werden. Es

liegt in der Verantwortung des Entwicklers, allen Nutzerinteraktionen mit dem System die korrekten Transaktionen zuzuordnen. Ein System ist *logisch konsistent*, wenn kein logisch inkonsistenter Zustand erreichbar ist.

2.2 Methoden für die Sicherstellung von Konsistenz

Um einen konsistenten Zustand zu gewährleisten gibt es verschiedene Werkzeuge. Dazu gehören vor allem die Unterstützung von lokalen Transaktionen der Datenbank und der 2 Phasencommit. In diesem Abschnitt werden die (...) beschrieben.

2.2.1 Transaktionen auf Datenbankebene

Eine lokale Transaktion einer Anwendung wird oft durch eine Nacheinanderausführung von SQL-Statements realisiert. Zu Beginn wird dem DBMS signalisiert, dass die folgenden Operationen als atomare Transaktion zu interpretieren sind. Analog dazu gibt es ein Signal, welches das Ende einer solchen Folge markiert, den *Commit*. Tritt vor Erreichen des Commits ein Fehler auf, wird ein *Rollback* ausgeführt. Dieser Rollback überführt die Daten in den Zustand vor Beginn der Transaktion. Die Datenbank - Transaktion - Begin - Commit - Rollback - Aufruf aus der Anwendung

2.2.2 Transaktionen auf Anwendungsebene

- Operation ist ein Funktionsaufruf - Verkettung ist mittels try-catch Block behandelbar - Verantwortung des Entwicklers, dass alle Operationen erfolgreich sind - Verantwortung des Entwicklers, Fehler zu behandeln - Abwicklung der Transaktion per DBMS Transaktionen sinnvoll

2.2.3 Grenzen der Konsistenz in Verteilten Systemen

- Transaktion kann Aktion beinhalten, die eine Abhängigkeit aufruft (zB Aufruf einer externen Http-Schnittstelle) - Zentrales Problem: Wie stelle ich sicher, dass ein Aufruf angekommen ist? ..., dass der Aufruf erfolgreich war? Wie gehe ich vor, wenn eine Aktion einer Transaktion nicht geklappt hat? - Folge von abhängigen Operationen (= Substitution: Ergebnis eines Aufrufs ist Argument der nächsten Aufrufs)

2.2.4 2 Phasencommit

- 2 Phasen Commit als verteilte Umsetzung des Transaktionsvorgehens - zentraler Koordinator - Vorbereitungsphase: Alle teilhabende Akteure der Transaktion geben dem Koordinator die Bestätigung, dass die Operation ausgeführt werden kann. Damit sie dieses Versprechen halten können, beinhalten die Vorbereitungen oft Blockierungen auf Datenbankseite. - Commitphase: Koordinator gibt den Teilhabenden das Signal, ihre Operation auszuführen. Bei Erfolg werden auch alle Blockierungen aufgehoben. Die Transaktion wird als erfolgreich markiert. - Optionale Rollbackphase: Falls ein Teilhabender einen Fehler zurückgibt, werden alle ausgeführten Änderungen zurückgenommen (Rollback). Danach

wird die Transaktion als abgebrochen markiert. Alle Blockierungen müssen zurückgenommen werden. - Nachteile: sehr hohe Chattines, sehr langsam, blockierend, geringer Throughput, komplexe Implementierung

2.2.5 Saga-Pattern

- Fehlerbehandlungsstrategie für monolithische und verteilte Systeme - Auflösen der Atomarität der Transaktionen in einzelne lokale Transaktionen T - Definieren von Kompensationsaktionen C - jedes T hat ein C - sequentielle Ausführung der Ts - schlägt ein T fehl, kann ein entsprechendes C ausgeführt werden

2.2.6 Vergleich von 2 Phasencommit und Sagas

- In welcher Ebene wird der Fehler erkannt - Wird der Fehler zugelassen - In welcher Ebene wird der Fehler behoben - Gleichzeitig(atomare Operationen durch 2PC gleichzeitig auslösen) vs sequentiell (Atomarität wird ersetzt durch sequentielle Operationen, die einzeln behandelt werden können)

2.3 Saga Pattern

2.4 Was sind die Probleme, die in verteilten ACID Transaktionen auftreten?

Im Grundlagenkapitel wurden die mit dem ACID-Konsistenzprinzip einhergehenden Probleme dargestellt:

- Blockierung der Ressourcen
- Viele Netzwerkaufrufe - Chattiness
- Enge Kopplung der Dienste -

2.5 Woher kommt es?

Referenzierung des Papers 1987 Garcia

2.6 Bestandteile des Musters

2.6.1 Aufteilung der Transaktion in lokale Transaktionen, Nacheinanderausführung

2.6.2 Cs und Ts - Kompensierungen

2.6.3 Betrachtung des Zustands nach Erfolg/Misserfolg

2.6.4 Recovery-Mechanismen

2.6.4.1 Forward Recovery

2.6.4.2 Backward Recovery

2.6.4.3 Save-Points

2.6.5 Ausprägungen des Patterns

2.6.5.1 Orchestration

2.6.5.2 Choreografie

2.7 Anwendungsgebiete des Patterns - Welche Usecases erlauben die Verwendung dieses Patterns? Welche nicht?

2.7.1 Langlebige Transaktionen - LLT

2.7.2 Bezug auf den Geschäftsprozess

2.7.3 Verteilte Systemlandschaft

2.7.4 Reaktion auf verschiedene Antwortmöglichkeiten in der Geschäftslogik

2.7.5 Fehlerfälle - Geschäftslogik und Ausfälle

Hier soll der Unterschied zwischen Fehlern in der Geschäftslogik und Fehler aufgrund Ausfällen erläutert werden.

3 Methodik

3.1 Zielstellung

Der praktische Teil dieser Arbeit soll dem Ziel dienen, die These zu beantworten: (HIER WIRD DIE THESE WIEDERHOLT)
Dazu soll im ersten Schritt ein Microservicesystem nach Vorbild des Saga-Patterns entworfen, entwickelt und bewertet werden. Der zu verwendende Prozess soll einen Geschäftsprozess abbilden, der für die Umsetzung mittels Saga-Pattern geeignet ist. Der Prozess soll also:

- LLTs enthalten,
- im Kontext eine verteilten Microservicearchitektur umgesetzt werden können und
- auf verschiedene Fälle in der Geschäftslogik reagieren können.

Die Mindestvoraussetzung ist, dass das umgesetzte System die genannten Kriterien erfüllt. Neben der Reaktion auf Fehler in der Geschäftslogik soll untersucht werden, wie das System auf Ausfälle reagiert.
Der zweite Schritt des praktischen Teils soll die Kritikpunkte der ersten Implementierung verwenden und Lösungen für potentielle Probleme liefern.

3.2 Bewertungskriterien

In diesem Abschnitt werden die verschiedenen Bewertungskriterien für das zu entwerfende und zu implementierende System festgelegt.
konkreter Bewertungsprozess mit Punkteverteilung folgt.

3.2.1 Anzahl der Pfade

ganzzahlige Werte und Prozentwerte im Vergleich zur Gesamtanzahl der Programmpfade
Es soll die Anzahl an Pfaden gezählt werden, die die Saga koordinieren.

- Gesamtanzahl der Programmpfade
- Anzahl der Programmpfade, die das System in einem konsistenten Zustand hinterlassen
  - Anzahl der Programmpfade, die in einer erfolgreichen Saga resultieren
  - Anzahl der Programmpfade, die in einer erfolglosen Saga resultieren
- Anzahl der Programmpfade, die das System in einem inkonsistenten Zustand hinterlassen

3.2.2 Anzahl der Netzwerkaufrufe

ganzzahliger Wert

- im erfolgreichen Pfad
- im erfolglosen Pfad mit Kompensierung

3.2.3 Wahrscheinlichkeit für Ausfall

Wahrscheinlichkeit
Jeder am Prozess teilhabende Microservice läuft auf einem Server. Um die Gesamtausfallwahrscheinlichkeit des Systems zu messen wird von einer konstanten Ausfallwahrscheinlichkeit pro Service ausgegangen. Die Gesamtausfallwahrscheinlichkeit ist die Summe der Wahrscheinlichkeiten, in denen das System einen Programmpfad verwendet, der die Daten in einem inkonsistenten Zustand hinterlässt.

3.2.4 Konsistenzerhaltung im Falle eines Ausfalls der Nichtkoordinierenden Services

ja oder nein
Es soll betrachtet werden, ob sich das System in einen konsistenten Zustand zurückfinden kann, falls einer der Services ausfällt, die vom Koordinator zu einer lokalen Transaktion aufgefordert werden können.

3.2.5 Konsistenzerhaltung im Falle eines Koordinatorausfalls

ja oder nein
Es soll betrachtet werden, ob sich das System in einen konsistenten Zustand zurückfinden kann, falls der koordinierende Service ausfällt.

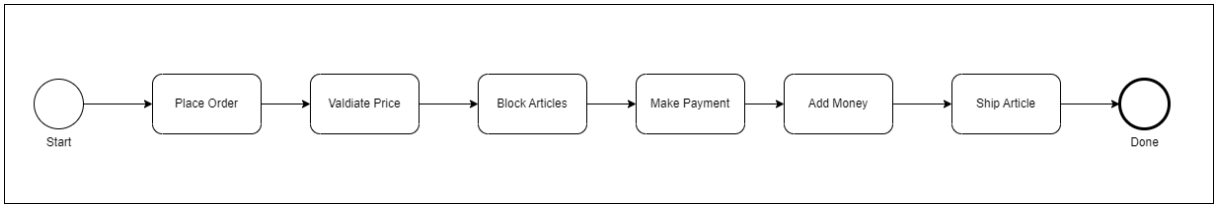
4 Erstes Saga-System

4.1 Entwurf des abzubildenden Geschäftsprozesses

Das zu implementierende System soll mittels Microservicearchitektur umgesetzt werden. Die einzelnen Services sollen das Saga-Pattern verwenden, um miteinander zu kommunizieren. Fehler in der Geschäftslogik sollen kompensiert werden. Für jeden auszuführenden Schritt soll es also einen kompensierenden Schritt geben.

4.1.1 Geschäftsprozess

Als abzubildender Geschäftsprozess soll ein Bestell- und Liefervorgang eines Online-Shops dienen. Der Bestellvorgang soll durch das Platzierung einer Bestellung ausgelöst werden. Die Benutzeroberfläche gehört nicht zum Scope des umzusetzenden Systems.
Als Ausgangspunkt soll folgender Geschäftsprozess dienen:



Die zum Prozess gehörenden Schritte sind folgende:

1. Entgegennehmen der Bestellung: Die Bestellung wird über ein imaginäres Frontend entgegengenommen. Dieses Frontend baut einen Request auf und sendet diesen per Http-Schnittstelle an das Backend. Dort wird der Request entgegengenommen und muss alle für die Abwicklung der Bestellung erforderlichen Daten enthalten. Dazu gehören der bestellende Nutzer, die geforderten Artikel und die Zahlungsinformationen. Beim Entgegennehmen wird die Bestellung initialisiert.

2. Validierung des Preises: Der Bestellungsrequest enthält eine Liste von den gewünschten Produkten und dem bekannten Preis pro Produkt. Um zu überprüfen, ob der dem Nutzer (dem Frontend) bekannte Preis mit dem aktuellen Preis übereinstimmt, muss dieser validiert werden.

3. Blockieren der Artikel: Die geforderten Artikel sollten für diese Bestellung reserviert werden, bis der Bestellvorgang abgeschlossen ist. In einem Online-Shop wird angezeigt, wieviele Artikel auf Lager vorrätig sind. Beim Blockieren der Artikel wird dieser Betrag verändert. Somit sehen andere Nutzer nach Ausführung dieses Schrittes den aktuellen Wert der vorrätigen Artikel.

4. Zahlungsabwicklung: Der berechnete Preis der Bestellung muss vom Konto des Kunden abgebucht und auf das Konto des Händlers gutgeschrieben werden. Die Konten des Kunden und des Online-Shop-Besitzers müssen nicht bei derselben Bank liegen. In diesem Schritt muss also eine verteilte Transaktion stattfinden.

5. Auslösen der Lieferung: Die blockierten Artikel werden versendet. Dieser Prozess dauert einen längeren Zeitraum an.

6. Abschluss der Lieferung: Die Saga ist abgeschlossen.

4.1.2 Services

Aus der Beschreibung des Geschäftsprozesses lassen sich folgende Services ableiten:

Name des Services	Aufgabe
Frontend	GUI, Anzeige der Produkte, Aufnahme der Bestellung, Platzieren der Bestellung
OrderService	Entgegennehmen der Bestellung, Koordinierung des Bestellprozesses
ArticleService	API für die angebotenen Produkte und Preise
StockService	Informationen über Lagerstand, Auslösen des Lieferprozesses
BankingServices	Schnittstellen für das Erhöhen und Verringern von Geldbeträten eines Kontos

4.1.3 Transaktionen

Sieht man den gesamten Geschäftsprozess als Transaktion, wären folgende lokale Transaktionen Teil der globalen Transaktion, die durch das Platzieren der Bestellung ausgelöst werden:

1. T_1: OrderService - Initialisieren der Bestellung

2. T_2: OrderService, ArticleService - Abfragen und Validieren des Preises für jeden geforderten Artikel

3. T_3: StockService - Blockieren der Artikel

4. T_4: BankingService des Kundenkontos - Verringern des Geldbetrages des Kundenkontos

5. T_5: BankingService des Händlerkontos - Erhöhen des Geldbetrages des Händlerkontos

6. T_6: StockService - Lieferung auslösen

7. T_7: StockService - Lieferung bestätigen

Nach der Funktionsweise des Saga-Patterns muss für jede lokale Transaktion eine Kompensierung angeboten werden:

Transaktion	Kompensierung
T_1	-
T_2	-
T_3	C_3: StockService - Freigeben der Artikel
T_4	C_4: Erhöhen des Geldbetrages des Kundenkontos
T_5	C_5: Verringern des Geldbetrages des Händlerkontos
T_6	-
T_7	-

4.2 Spezifikation der Services

Für die Realisierung des Microservicesystems im Rahmen dieser Arbeit wurde die Orchestrierung gewählt. Die Rolle des Koordinators übernimmt der OrderService. Der OrderService übernimmt die Annahme des Bestellprozesses und löst somit die Saga aus.

4.2.1 Frontend

4.2.1.1 Funktionalitäten

In einem Online-Shop interagiert der Kunde per Frontend mit der Anwendung. Das Frontend soll übernimmt die grafische Schnittstelle zwischen Backend und dem Nutzer. Dazu gehört vor Allem die Darstellung der Artikel in einer Katalogansicht. Die darzustellenden Daten für eine solche Liste müssen zumindest Artikelbezeichnung und Artikelpreis enthalten. Diese Daten sollten aus einer API für Artikeldaten stammen. Darüber hinaus muss das Frontend einen Prozess unterstützen, in dem der Kunde ein Formular ausfüllt, welches die erforderlichen Daten für das Platzieren einer Bestellung enthält. Dazu gehört ein Warenkorbsystem sowie eine Authorisierung und Authentifizierung der Zahlungsidentität des Kunden. Die Bestellung kann also als Objekt mit folgenden Feldern zusammengefasst werden:

- Consument
  - BankId
  - UserId
- Liste der zu bestellenden Artikel
  - ArticleId
  - ArticlePrices
  - Amount

Dieses Objekt kann an das Backend gesendet werden.

4.2.2 ArticleService

4.2.2.1 Funktionalitäten

Dieser Service ist ein Service zum reinen Lesen der Produktdaten. Er soll eine Schnittstelle zur Verfügung stellen, die dem Frontend ermöglicht, den Produktkatalog abzufragen und

darzustellen. Das Backend muss außerdem die Möglichkeit haben, die im Request enthaltenen Artikelpreise zu validieren. Dazu benötigt der Service eine Produktdatenbank. Da dieser Service ausschließlich die Produktdaten als Ressource behandelt, kann er RESTful implementiert werden.

4.2.2.2 Endpunkte

Diese Schnittstelle liefert eine Liste von Produkten.
GET /products
Diese Schnittstelle liefert für eine ProduktId das zugehörige Produkt.
GET /products/productId

4.2.2.3 Datenbanktabellen

Die Produktdatenbank benötigt lediglich eine Tabelle mit den Artikeldaten.

4.2.2.4 Kompensierung

Da dieser Service den Systemzustand nicht verändert, sondern lediglich lesend auf die Produktdaten zugreift, gibt es keine kompensierenden Endpunkte.

4.2.3 StockService

4.2.3.1 Funktionalitäten

Dieser Service soll dazu dienen, den Lagerbestand der vorhandenen Artikel zu verwalten. Dazu gehört die Reservierung von Artikeln, das Auslösen und der Abschluss einer Lieferung.

4.2.3.1.1 Darstellung des aktuellen Lagerbestands Um dies zu erlauben, muss der aktuelle Lagerstand in einer Tabelle hinterlegt sein. Die Tabelle muss den aktuell verfügbaren Bestand pro Artikel ausdrücken.

4.2.3.1.2 Reservierung von Artikeln Um eine Reservierung zu ermöglichen, muss es eine weitere Tabelle geben, die eine Menge von blockierten Artikeln für einen bestimmten Bestellprozess blockiert. Beim Reservieren verringert sich der Bestand in der Bestandstabelle und erhöht sich in der Reservierungstabelle. Um die Konsistenz zu gewährleisten, müssen beide Operationen in einer lokalen Transaktion ausgeführt werden. Die Summe der vorrätigen Artikel und der reservierten Artikel darf sich nicht verändern bis der Artikel geliefert wird und somit tatsächlich nicht mehr vorrätig ist.

4.2.3.1.3 Auslösen einer Lieferung Um das Auslösen und Abschließen einer Lieferung zu ermöglichen, muss es eine Tabelle geben, die den Inhalt einer Lieferung und einen Status enthält. Wenn eine Lieferung ausgelöst wird, werden die für diesen Vorgang reservierten Artikel aus der Reservierungstabelle entfernt und in der Lieferungstabelle eingefügt. Um Konsistenz zu gewährleisten, muss dies in einer lokalen Transaktion erfolgen.

4.2.3.2 Endpunkte

Die Schnittstelle zum Reservieren von Produkten empfängt einen Http-Body mit folgenden Daten:

- Vorgangsnummer
- Liste von Produkten
  - ArtikelId
  - Anzahl

$POST\ /blocked-articles$
Die Schnittstelle zum Auslösen einer Bestellung muss lediglich die Vorgangsnummer enthalten.
$GET\ /shipments/shipmentId$

4.2.3.3 Datenbanktabellen

articlestock, blockedarticles, shipments

4.2.3.4 Kompensierung

Die Blockierung eines Artikels muss kompensiert werden können, da sonst der blockierte Artikel nach Abbruch einer Bestellung nicht wieder freigegeben würde. Deshalt muss diese Kompensierung die Einträge aus der Blockierungstabelle entfernen und die Anzahl auf den Lagerbestand addiert werden. Dies soll ebenfalls in einer lokalen Transaktion ablaufen, um Konsistenz zu wahren.
Das Auslösen einer Lieferung ist nicht in dem Sinne kompensierbar. Das liegt nicht an dem System, sondern am repräsentierten Geschäftsprozess. Eine Stornierung ist im Rahmen dieser Umsetzung nicht vorgesehen.

4.2.4 BankingServices

4.2.4.1 Funktionalitäten

Im Geschäftsprozess wurde definiert, dass die Transaktion das Geldbetrag des Kundenkontos und des Händlerkontos in zwei separaten Transaktionen abwickeln können soll. Somit muss der BankingService jeweils eine Transaktion zum Erhöhen und zum Verringern des Geldbetrages anbieten. Der BankingService soll am Ende in zwei Instanzen laufen, die zwei verschiedene Banken darstellen sollen. Kunden- und Käuferkonto können, müssen aber nicht bei derselben Bank liegen.
Um dies zu ermöglichen benötigt der BankingService eine Tabelle, die seine Nutzer enthält. Zusätzlich benötigt der Service eine Tabelle, die den aktuellen Geldbetrag jedes Nutzers enthält. Außerdem sollten die einzelnen Transaktionen jedes Nutzers in einer separaten Tabelle gesichert werden. Für die reine Implementierung dieser Anwendung wäre dies nicht notwendig. Für den Nutzer eines BankingServices ist neben dem Kontostand auch die Liste an getätigten Transaktionen interessant, um die Ausgaben und Einnahmen zuordnen zu können. Im Rahmen dieser Implementierung wird die Tabelle zusätzlich für Analysezwecke verwentet werden.

Bei einer Anfrage, den Geldbetrag eines konkreten Nutzers zu erhöhen, wird in einer lokalen Transaktion der Betrag des Kontos in der UserCredit-Tabelle erhöht und die Differenz in der Transaktion-Tabelle eingetragen.
Der Service darf Anfragen zum Geld verringern ablehnen, wenn die Verringerung den Kontostand in den negativen Bereich fallen lassen würde. In diesem Fall wird die Transaktion abgebrochen.

4.2.4.2 Endpunkte

$POST\ /add-money$
$POST\ /remove-money$

4.2.4.3 Datenbanktabellen

user, usercredit, usertransaction

4.2.4.4 Kompensierung

Beide angebotenen Operationen benötigen eine zugehörige Kompensation, da sie den Datenbestand verändern. Die Verwendung des jeweils anderen Endpunktes ist semantisch bereits korrekt. Der Klarheit halber sollen zwei weitere Entpunkte eingeführt werden, die nur für die Kompensation verwendet werden sollen.
$POST\ /add-money-compensation$
$POST\ /remove-money-compensation$

4.3 OrderService

4.3.0.1 Funktionalitäten

Der OrderService übernimmt die Rolle des Koordinators im Orchestrator-Saga-Patterns. Die Bestellung wird entgegengenommen und vom OrderService initialisiert. Zur Initialisierung gehört die Generierung einer Vorgangsnummer sowie das Abspeichern der Bestellung in einer separaten Tabelle. Anhand dieser Tabelle wird persistiert, an welcher Stelle der Ausführung die Saga sich befindet, und in welchem Status die Bestellung ist. Die etwaigen ausgeführten Kompensationsschritte sind in ihrer eigenen Tabelle und werden der Vorgansnummer zugeordnet. Die gewünschten Artikel einer Bestellung sind in eine separate Tabelle ausgelagert und verweisen auf die Saga-Tabelle.
Als Koordinator hat dieser Service die Verantwortung, die an der Saga beteiligten Services korrekt aufzurufen. Die Reihenfolge und die getroffenen Entscheidungen repräsentieren die Geschäftslogik.
Nach jedem Schritt persistiert der OrderService den Erfolg oder Misserfolg. Außerdem ruft der Service nach Feststellung eines Misserfolgs die Kompensierungskette auf.

4.3.0.2 Endpunkte

$POST\ /order$

4.3.0.3 Datenbanktabellen

ordersaga, requestedarticle, ordersagacompensations

4.3.0.4 Durchlauf einer erfolgreichen Saga

4.3.0.5 Durchlauf einer gescheiterten, kompensierten Saga

4.3.0.6 Durchlauf einer gescheiterten, nichtkompensierten Saga

5 Implementierung

5.1 Abänderung der Implementierung

Abbildungsverzeichnis