Inhaltsverzeichnis

1 Einleitung **2**

1.1 Motivation . . . 2
1.2 Zielsetzung dieser Arbeit . . . 2
1.3 Aufbau dieser Arbeit . . . 2

2 Grundlagen **3**

2.1 Grundlagen . . . 3
2.1.1 System allgemein . . . 3
2.1.2 System in der Softwareentwicklung . . . 3
2.1.3 Zustand von Systemen . . . 3
2.1.4 Transaktionen . . . 3
2.1.5 ACID Eigenschaft von Transaktionen . . . 4
2.1.6 Konsistenz auf Datenbankebene . . . 4
2.2 Methoden für die Sicherstellung von Konsistenz . . . 5
2.2.1 Transaktionen auf Datenbankebene . . . 5
2.2.2 Transaktionen auf Anwendungsebene . . . 5
2.2.3 Grenzen der Konsistenz in Verteilten Systemen . . . 5
2.2.4 2 Phasencommit . . . 5
2.3 Saga Pattern . . . 6
2.3.1 Was sind die Probleme, die in verteilten ACID Transaktionen auftreten? 6
2.3.2 Woher kommt es? . . . 6
2.3.3 Bestandteile des Musters . . . 6
2.3.3.1 Vereinfachtes Grundprinzip . . . 6
2.3.3.2 Anforderungen an Ts und Cs . . . 6
2.3.3.3 Saga Execution Component . . . 8
2.3.3.4 Formalisierung eines Saga-Zustandsautomaten als DEA . . 8
2.3.3.4.1 Formale Darstellung eines DEA . . . 8
2.3.3.4.2 Saga als formale Sprache . . . 8
2.3.3.4.3 Überführung einer Saga in einen DEA . . . 8
2.3.3.4.4 Konfiguration . . . 9
2.3.3.5 Betrachtung des Zustands nach Erfolg/Misserfolg . . . 10
2.3.3.5.1 Endzustand q_{f1} . . . 10
2.3.3.5.2 Endzustand q_{f2} . . . 11
2.3.3.5.3 Endzustand q_{f3} . . . 11
2.3.3.6 Unterschiede des Saga-Modells zu Hector Garcia-Molinas Definition . . . 11
2.3.3.7 Recovery-Mechanismen . . . 11
2.3.3.7.1 Backward Recovery . . . 12
2.3.3.7.2 Forward Recovery . . . 12

2.3.3.7.3 Voraussetzung für Forward-Recovery 14
2.3.3.8 Implementierungsformen des Patterns 14
2.3.3.8.1 Orchestration 15
2.3.3.8.2 Choreografie 15
2.3.3.8.3 Kommunikationsstrategien 15
2.3.4 Anwendungsgebiete des Patterns - Welche Usecases erlauben die Verwendung dieses Patterns? Welche nicht? 17
2.3.4.1 Langlebige Transaktionen - LLT 17
2.3.4.2 Bezug auf den Geschäftsprozess 17
2.3.4.3 Verteilte Systemlandschaft 17
2.3.4.4 Reaktion auf verschiedene Antwortmöglichkeiten in der Geschäftslogik 17
2.3.4.5 Fehlerfälle - Geschäftslogik und Ausfälle 17

3 Versuchsvorbereitung 18
3.1 Problemstellung 18
3.2 Zielstellung 18
3.3 Ausgangspunkt 19
3.4 Methodik 19
3.4.1 Schritt 1 - Entwurf und Implementierung eines GP 19
3.4.2 Schritt 2 - Messung der verschiedenen Implementierungen 20
3.4.3 Schritt 3 - Analyse der Messdaten 21

4 Implementierung des Saga-Systems 22
4.1 Services 22
4.2 Transaktionen 22
4.3 Fachliche Kontextabgrenzung 23
4.3.1 Frontend 23
4.3.2 ArticleService 23
4.3.3 StockService 23
4.3.4 BankingServices 24
4.3.5 OrderService 25
4.4 Technische Kontextabgrenzung 25
4.4.1 ArticleService 26
4.4.1.1 Datenbankschicht 26
4.4.1.2 Schnittstellen 26
4.4.2 StockService 26
4.4.2.1 Datenbankschicht 26
4.4.2.2 Schnittstellen 27
4.4.3 BankingService 28
4.4.3.1 Datenbankschicht 28
4.4.3.2 Schnittstellen 29
4.5 Darstellung als DEA 29
4.5.1 Ergebnisse aller Transaktionen 29
4.5.2 Ergebnisse aller Kompensierungen 30
4.5.2.1 Durchlauf einer erfolgreichen Saga 31

4.5.2.2 Durchlauf einer gescheiterten, kompensierten Saga 31
4.5.2.3 Durchlauf einer gescheiterten, nichtkompensierten Saga . . 31

5 Implementierung **I**
5.1 Abänderung der Implementierung I

Abbildungsverzeichnis **II**

Literaturverzeichnis **III**

1 Einleitung

1.1 Motivation

Wieso, Weshalb, Warum?

1.2 Zielsetzung dieser Arbeit

These: Mittels Saga-Pattern implementierte langlebige Transaktionen (LLT) ermöglichen Konsistenz in Microservicesystemen, die per Request-Response Pattern kommunizieren.
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 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.

2.1.5 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.

2.1.6 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

2.3 Saga Pattern

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

Im vorherigen Abschnitt wurden die mit dem ACID-Konsistenzprinzip einhergehenden Probleme dargestellt, wenn man sich in einem verteilten System befindet:

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

2.3.2 Woher kommt es?

Referenzierung des Papers 1987 Garcia

2.3.3 Bestandteile des Musters

2.3.3.1 Vereinfachtes Grundprinzip

Das Saga-Pattern ist ein Entwurfsmuster, um eine verteilte Transaktion abzubilden. Eine Transaktion wird hier auch durch eine Menge von auszuführenden Operationen gebildet. Auch für das Saga-Pattern gehört die Gewährleistung der ACID-Eigenschaft zu den Anforderungen. Die Art und Weise, wie diese Eigenschaften erreicht werden unterscheidet sich jedoch vom 2-Phasencommit. Anstatt alle teilnehmenden Operationen durch einen Befehl gleichzeitig auszulösen, werden alle Operationen sequentiell ausgeführt. Eine Operation ist im Saga-Pattern eine lokale Transaktion, die in sich geschlossen ist und ebenfalls die ACID-Eigenschaft erfüllen muss. Für eine solche Operation wird gleichzeitig eine Schnittstelle angeboten, die die Veränderungen rückgängig macht. Somit besteht die Möglichkeit eine Operation zu neutralisieren. Es wird also die Voraussetzung an den Entwickler gestellt, für jede angebotene Operation eine Umkehroperation bereitzustellen, die selbst eine lokale Transaktion darstellt. Im Paper werden lokale Transaktionen, die eine Operation der Transaktion darstellen, als Ts bezeichnet. Die dazugehörigen lokalen Transaktionen werden als Cs bezeichnet. Die Transaktion kann nun also ausgeführt werden. Jedes teilhabende T wird nacheinander ausgeführt. Tritt bei der Ausführung eines Ts ein Fehler auf, können alle bereits ausgeführten Operationen in ihren Ursprungszustand zurückgesetzt werden, indem in der umgekehrten Reihenfolge die notwendigen Cs aufgerufen werden. Im Fehlerfall wird der Ausgangszustand in allen Services wiederhergestellt und die Atomarität der Transaktion ist gewährleistet. Sind alle Operationen erfolgreich, wird nach Ausführung aller Ts der Endzustand erreicht und die Transaktion hat einen neuen Zustand hergestellt. Sowohl im erfolgreichen als auch im kompensierten Endzustand ist die Konsistenz gewahrt.

2.3.3.2 Anforderungen an Ts und Cs

Die einzelnen Operationen, die ausgeführt werden müssen, müssen also kompensiert werden können. Eine solches T wird als lokale Transaktion betrachtet, die in einem anderen Service

stattfindet. Der aufrufende Service soll als S1 und der T durchführende Service als S2 bezeichnet werden. Der Effekt der Transaktion ist häufig eine Veränderung in der Datenbank von S2. Es wird nun betrachtet, welche verschiedenen Effekte T in der Datenbank haben kann und welchen Effekt das entsprechende C haben muss.
Äußert sich der Effekt von T in einem Insert, dann ist der kompensierende Effekt von C ein Delete. Somit ist dieses Element für andere Prozesse nicht mehr auffindbar. Der Zustand ist also identisch mit dem Zustand vor Durchführung von T. Anstatt das Element zu löschen kann eine Markierung des zuvor eingefügten Elements vorgenommen werden. Diese Markierung kann ein zusätzliches Feld sein oder eine zusätzliche Tabelle, die die gelöschten Elemente auflistet. Muss ein Prozess nun mit den gültigen Elementen der Tabelle arbeiten, wird zusätzlich diese Markierung geprüft. Es ist somit möglich, den Zustand vor Durchführung von T zu reproduzieren.
Wenn der Effekt von T ein ein Update ist, ist die Kompensierung unter Umständen schwieriger. Wenn der Effekt von T eine Erhöhung eines Feldes um n auslöst, dann muss der Effekt das selbe Feld um n verringern. Eine solche Paarung von T und C ist leicht zu implementieren.
Ist der Effekt von T ein idempotentes Update, dann muss S2 eine Historytabelle bereithalten, um den Ausgangszustand reproduzieren zu können. Angenommen das vom Update betroffene Feld hat anfangs den Wert n und der Effekt von T setzt den Inhalt des Feldes auf n-Strich, dann muss innerhalb einer lokalen Transaktion der Wert von n in einer Historytabelle abgespeichert werden und der neue Wert kann gesetzt werden. Wenn nun der Effekt von T kompensiert werden soll, wird der historische Wert n per Select ermittelt und im tatsächlichen Feld gesetzt werden.
Es ist außerdem möglich, dass ein T wiederum externe Schnittstellen aufruft. In diesem Fall kann die Kompensierungslogik nicht verallgemeinert werden. Es ist möglich, den Prozess so zu gestalten, dass ein solches T als Sub-Saga TSub modelliert wird. Dabei muss jedoch gewährleistet werden, dass bei Aufruf von C alle Kompensierungen von TSub aufgerufen werden.
Es ist außerdem hervorzuheben, dass der Effekt eines Ts neben Änderungen in der Datenbank oder Aufrufe von anderen Schnittstellen auch reale Geschäftsprozesse auslösen können. Ein solcher Prozess kann unter Umständen nicht kompensierbar sein. Hier kann auch weiter differenziert werden.
Ist der Effekt von T die Versendung eines Briefs, so kann diese Versendung nicht kompensiert werden. Ein Folgebrief kann jedoch als Kompensierung angesehen werden, die den ausgelösten Effekt neutralisiert. Im Folgebrief können beispielsweise Anweisungen stehen, die den Empfänger informieren, dass der vorherige Brief als ungültig angesehen werden kann. Der Effekt von T kann hier als kompensiert angesehen werden.
Es gibt jedoch auch Effekte, die nicht kompensierbar sind und im Scheitern einer Saga resultieren. In solchen Fällen kann das System in einen inkonsistenten Zustand überführt werden. Dieses Verhalten tritt immer dann auf, wenn der Effekt einer Transaktion in einer endgültigen Aktion resultiert.

2.3.3.3 Saga Execution Component

In Garcia-Molinas Paper wird eine Saga Execution Component beschrieben, welche alle ausgeführten Aktionen einer Saga protokolliert und die nächste auszuführende Aktion bestimmt. Grundlegend besteht diese Komponente aus einem Transaktionslog und einem Zustandsautomaten.
Der Transaktionslog ist eine Liste von Transaktionen und Kompensierungen, die als Kommandos dargestellt werden. Jedes Transaktionslog enthält ein Kommando, welches den Start ($BeginnSaga$) und den Abschluss ($EndSaga$) der Saga darstellt. Neben den Ts und Cs zwischen $BeginnSaga$ und $EndSaga$ können weitere Kommandos definiert werden, um den Kontrollfluss zu beeinflussen.
Der Zustandsautomat bestimmt aus den bisherigen Ts und Cs den nächsten Schritt.

2.3.3.4 Formalisierung eines Saga-Zustandsautomaten als DEA

2.3.3.4.1 Formale Darstellung eines DEA Der Prozessablauf einer Saga kann als detemernistischer endlicher Automat angesehen werden. Ein DEA wird formal dargestellt als Tupel mit folgenden Elementen:

- Q: Zustandsmenge
- Σ: endliches Eingabealphabet
- $\delta : Q \times \sum \rightarrow Q$: Übergangsrelation
- $q_0 \in Q$: Startzustand
- $F \subseteq Q$: Menge an akzeptierenden Zuständen

2.3.3.4.2 Saga als formale Sprache Im vorherigen Abschnitt wurde die Saga Execution Component definiert als ein Tupel aus Transaktionslog und Zustandsautomat. Nun soll dieses Tupel in einen DEA überführt werden. Ein solcher DEA A_{Saga} akzeptiert die Sprache L_{Saga}, die alle gültigen Wörter enthält, die eine Saga darstellen.
Das Eintabealphabet Σ ist die Menge aller Elemente, die im Transaktionslog auftauchen können. Somit kann jedes Transaktionslog als Eingabewort aufgefasst werden.
Somit ist die von A_{Saga} akzeptierte Sprache $L_{Saga} = L(A_{Saga})$:

$$\forall w \in \Sigma^\star : w \in L(A_{Saga}) \iff w \in L_{Saga}$$

2.3.3.4.3 Überführung einer Saga in einen DEA Um eine Saga in einen DEA überführen zu können, müssen zuerst einige Definitionen vorgenommen werden. Die Unterscheidung zwischen Ts und Cs wird im Modell eines Zustandsautomaten per Zustand ausgedrückt. Es muss also eine Abstrahierung vorgenommen werden, die Ts und Cs vereinigt. Diese Abstrahierung wird im Folgenden als *Aktion A* bezeichnet. Eine solche Aktion a_n wird immer im entsprechenden Zustsand $q_n \in Q$ ausgeführt. In der folgenden Erläuterung kann die Zustandsmenge Q mit der Menge $T \cup C$ gleichgesetzt werden.

Das Eingabealphabet Σ drückt aus, welche möglichen Ergebnisse eine Aktion haben kann. Eine Aktion kann einerseits ein Aufruf einer externen Schnittstelle sein. Die Antwort dieser Schnittstelle kann das Ergebnis in unterschiedlichen Formen ausdrücken. Das können beispielsweise folgende Ausdrucksformen sein:

- Http-Statuscode
- Custom Http-Responsebody

Diese sind üblicherweise in einer Schnittstellendefinition aufgelistet. Im Folgenden wird davon ausgegangen, dass alle möglichen Antworten einer Http-Schnittstelle per Http-Statuscode ausgedrückt werden. Es wird ein Typ definiert, der für jede Aktion alle möglichen Http-Statuscodes enthält:

$$API - Ergebnis\ AE \in \{tn_{200}, tn_{201}, tn_{400}, tn + 1_{200}, tn + 1_{400}, tn + 1_{409}, ...\}.$$

Eine Aktion kann neben dem Aufruf einer Schnittstelle eine interne Verarbeitung sein. Das könnte beispielsweise eine Prüfung auf Vorhandensein eines Feldes in einer vorangegangenen Schnittstellenantwort sein. Eine solche Aktion wird definiert:

$$Internes\ Prozessergebnis\ IPE \in \{tn_{Success}, tn_{Failure}, tn + 1_{Success}, tn + 1_{Failure}, ...\}.$$

Ein Ergebnis einer Aktion wird also definiert als:

$$Ergebnis\ E = AE \cup IPE.$$

Das Eingabealphabet beinhaltet Elemente aus dem Ergebnistyp:

$$\Sigma = Ergebnis.$$

Ein Übergang von einem Zustand in den Folgezustand drückt somit aus, dass die Saga eine Aktion ausgeführt hat und dem Ergebnis entsprechend einen Zustandswechsel durchgeführt hat.
Der Startzustand q_0 ist die erste auszuführende Transaktion.
Ein Endzustand q_{f1} wird erreicht, nachdem die letzte auszuführende Transaktion erfolgreich beendet wurde. Ein weiterer Endzustand q_{f2} wird erreicht, nachdem die letzte Kompensierung erfolgreich beendet wurde. Der letzte Endzustand q_{f3} wird erreicht, nachdem die erste Kompensierung erfolglos beendet wurde.

2.3.3.4.4 Konfiguration Die Ausführung eines DEA kann mittels Konfigurationen dargestellt werden. Eine Konfiguration K ist definiert als:

$$K = (q, w) \in Q \times \Sigma^\star$$

Der Automat wechselt in einen Folgezustand, indem er ein Element aus dem Eingabewort abarbeitet und eine passende Übergangsrelation in δ findet. Somit gilt:

$$q1, q2 \in Q \land u \in \Sigma \land v \in \Sigma^\star : (q_1, u \circ v) \vdash (q_2, v) \implies \delta(q_1, u) = q_2$$

Außerdem können mehrere Konfigurationsübergänge mittels $\vdash^\star$ dargestellt werden:

$$K_1 \vdash^\star K_2 \implies K_1 = K_2 \lor \exists K : K_1 \vdash K \land K \vdash^\star K_2$$

2.3.3.5 Betrachtung des Zustands nach Erfolg/Misserfolg

Der Zustand des Systems soll nun in folgenden Fällen betrachtet werden:

1. Erfolgreicher Ablauf einer Saga
2. Scheitern der Saga nach n Schritten
3. Scheitern der Saga nach n Schritten und Scheitern der Kompensierung nach m Schritten

Die Ausführung der Saga als DEA soll an folgendem Beispiel illustriert werden:
$Saga = (Q, \Sigma, \delta, q_0, F)$ mit

$$
\begin{aligned}
Q &= \{q_{t1}, q_{t2}, q_{t3}, q_{c1}, q_{c2}, q_{c3}, q_{f1}, q_{f2}, q_{f3}\} \\
\Sigma &= \{t1_{200}, t1_{400}, t2_{Success}, t2_{Failure}, t3_{200}, t3_{400}, c1_{200}, c1_{400}, c2_{Sucess}, c2_{Failure}, c3_{200}, c3_{400}\} \\
\delta &= \{((q_{t1}, t1_{200}), q_{t2}), ((q_{t2}, t2_{Sucess}), q_{t3}), ((q_{t3}, t3_{200}), q_{f1}), ((q_{t1}, t1_{400}), q_{f2}), \\
&\quad ((q_{t2}, t2_{Failure}), q_{c1}), ((q_{t3}, t3_{400}), q_{c2}), ((q_{c1}, c1_{200}), q_{f2}), ((q_{c2}, c2_{Sucess}), q_{c1}), \\
&\quad ((q_{c3}, c3_{200}), q_{c2}), ((q_{c1}, c1_{400}), q_{f3}), ((q_{c2}, c2_{Failure}), q_{f3}), ((q_{c3}, c3_{Failure}), q_{f3})\} \\
q_0 &= q_{t1} \\
F &= \{q_{f1}, q_{f2}, q_{f3}\}
\end{aligned}
$$

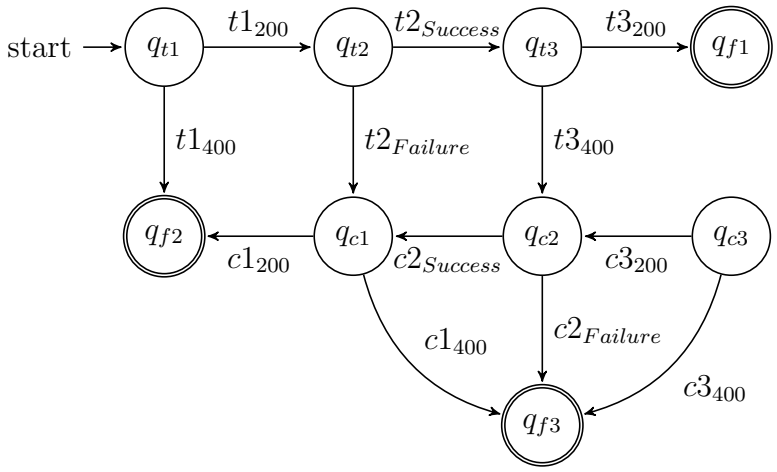


Abbildung 2.1: Saga als DEA

2.3.3.5.1 Endzustand q_{f1} Im Folgenden wird davon ausgegangen, dass die Aktionen der Zustände q_{t1}, q_{t2} und q_{t3} in einem erfolgreichen Ergebnis resultieren. Somit wird am Ende der Endzustand q_{f1} erreicht. Dieser Zustand drückt einen erfolgreichen Durchlauf einer Saga aus. Das Eingabewort $e_1 \in \Sigma^\star$ ist $t1_{200} \circ t2_{Success} \circ t3_{200} \circ \#$.
Die Konfigurationsübergänge für e_1 sind:

$$
\begin{aligned}
(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{200} \circ \#) \\
\vdash (q_{t2}, t2_{Success} \circ t3_{200} \circ \#) \\
\vdash (q_{t3}, t3_{200} \circ \#) \\
\vdash (q_{f1}, \#)
\end{aligned}
$$

2.3.3.5.2 Endzustand q_{f2} Es wird nun davon ausgegangen, dass bei der Aktion im Zustand q_{t3} ein Ergebnis $t3_{400}$ erfolgt. Ein solches Ergebnis führt dazu, dass der Zustand q_{c2} erreicht wird. Hier wird davon ausgegangen, dass die Aktionen q_{c2} und q_{c1} erfolgreiche Ergebnisse haben. Das Eingabewort $e_2 \in \Sigma^\star$ ist $t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200}$.
Die Konfigurationsübergänge für e_2 sind:

$$\begin{aligned}(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#) \\ \vdash (q_{t2}, t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#)) \\ \vdash (q_{t3}, t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#)) \\ \vdash (q_{c2}, c2_{Success} \circ c1_{200} \circ \#)) \\ \vdash (q_{c1}, c1_{200} \circ \#)) \\ \vdash (q_{f2}, \#))\end{aligned}$$

2.3.3.5.3 Endzustand q_{f3} Zuletzt soll der Zustand q_{f3} betrachtet werden. Dafür soll die Aktion in q_{t3} das Ergebnis $t3_{400}$ haben. Danach schlägt die Aktion q_{c2} fehl und liefert das Ergebnis $c2_{Failure}$. Das Eingabewort $e_3 \in \Sigma^\star$ ist $t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Failure} \circ \#$.
Die Konfigurationsübergänge für e_3 sind:

$$\begin{aligned}(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Failure} \circ \#) \\ \vdash (q_{t2}, t2_{Success} \circ t3_{400} \circ c2_{Failure} \circ \#) \\ \vdash (q_{t3}, t3_{400} \circ c2_{Failure} \circ \#) \\ \vdash (q_{c2}, c2_{Failure} \circ \#) \\ \vdash (q_{f3}, \#)\end{aligned}$$

2.3.3.6 Unterschiede des Saga-Modells zu Hector Garcia-Molinas Definition

2.3.3.7 Recovery-Mechanismen

Eine Saga, die in der Ausführung einer Transaktion fehlschlägt, wechselt nach der Definition in die entsprechende Kompensierung und versucht, alle bis dahin ausgeführten Transaktionen zu kompensieren. Somit wird der Anfangszustand des Systems wiederhergestellt. Dieses Verhalten wird als Backward-Recovery bezeichnet.
Neben der Backward Recovery wird ein weiteres Verhalten vorgeschlagen, welches Forward-Recovery genannt wird. Das Ziel der Forward Recovery ist es, seltener in einem erfolglosen Endzustand zu gelangen. Im Modell der hier aufgestellten DEA-Saga sind das Zustände q_{f2} und q_{f3}. Um dies zu erreichen, werden Save-Points definiert. Ein Save-Point stellt einen Zustand dar, von dem bei einem Systemabsturz oder einem erfolglosen Ergebnis die Ausführung weitergeführt werden kann. Es wird im Fehlerfall Backward-Recovery bis zum nächsten Save-Point ausgeführt. Wird dieser erreicht, werden alle noch fehlenden Ts ausgeführt, um zum erfolgreichen Endzustand zu gelangen. Das bedeutet, dass von der Kompensierungskette zurück auf die Transaktionskette gesprungen wird.

2.3.3.7.1 Backward Recovery Der DEA einer Saga, die Backward-Recovery implementiert, ist im vorherigen Abschnitt beschrieben.

2.3.3.7.2 Forward Recovery Forward-Recovery ist auf verschiedenen Wegen erreichbar. Der erste Ansatz beinhaltet die Verwendung eines Save-Points. Der DEA aus Abschnitt soll um einen Checkpoint und Forward Recovery ergänzt werden. Es wird ein weiterer Zustand eingeführt, der nach erfolgreichem Ergebnis von q_{t1} erreicht wird. Der Checkpoint wird hier dargestellt als ein interner Prozessschritt q_{sp1} und hat somit die möglichen Ergebnisse $\in \{sp1_{Success}, sp1_{Failure}\}$. Es ist zu sehen, dass dieser DEA eine mögliche Endlosschleift zulässt. Wenn q_{sp1} erreicht wird und in q_{t2} oder q_{t3} immer ein erfolgloses Ergebnis auftritt, darf im Zustand $qsp1$ nur endlich oft der Übergang $sp1_{Success}$ gewählt werden.
Die Funktion f die in q_{sp1} ein internes Prozessergebnis IPE berechnet, sieht so aus:

$f : \mathbb{N} \to IE$
$maxSavepointExecutionCount \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{sp1}\ während$
$der\ Ausführung\ der\ Saga$

$$f(x) = \begin{cases} IE_{Success}, x < maxSavepointExecutionCount \\ IE_{Failure}, else \end{cases}$$

Die Anzahl an Ausführungen beginnend bei q_{t2} ist begrenzt. Es wird also solange Forward Recovery versucht, bis die Saga erfolgreich ist oder das Oberlimit $maxSavepointExecutionCount$ erreicht wird. Wenn dieses Oberlimit erreicht ist, wird die Forward Recovery aufgegeben und in den Zustand q_{c1} gewechselt.

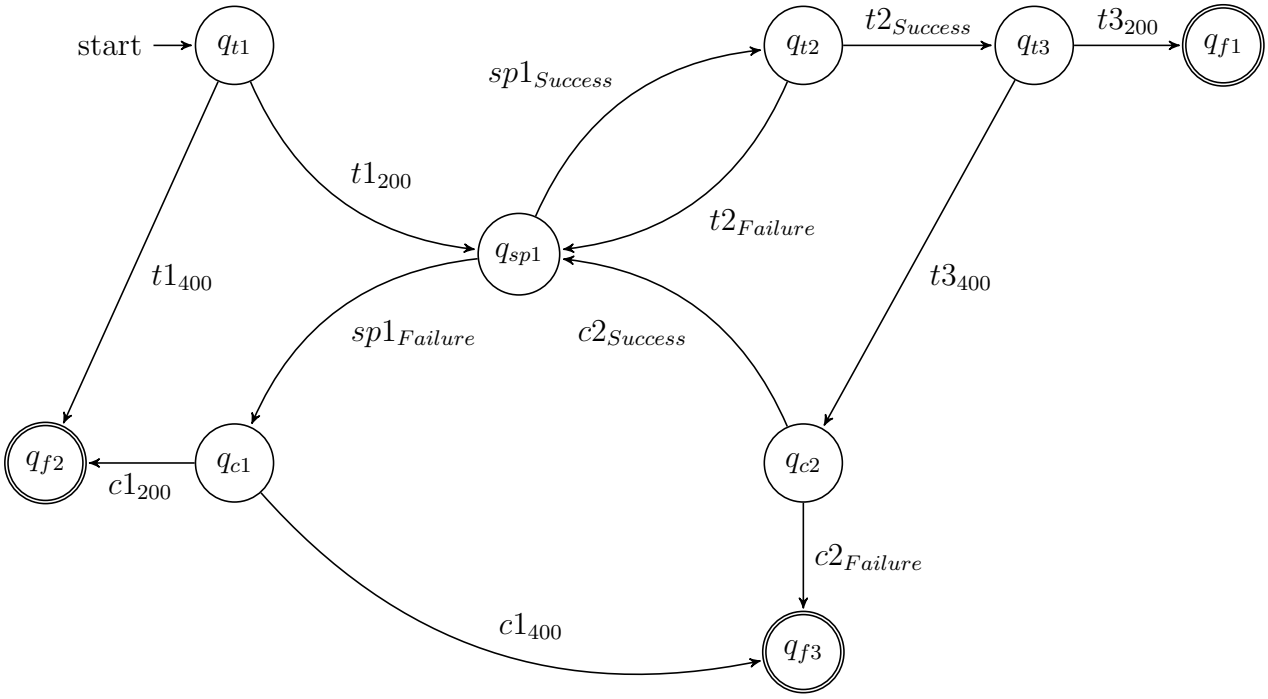


Abbildung 2.2: Saga als DEA

Forward-Recovery kann alternativ auch als Retry interpretiert und somit ohne Save-Points realisiert werden. Einen solcher Retry kann sehr einfach in jedem Zustand ergänzt werden. Dazu wird eine Kante hinzugefügt, die im gleichen Zustand bleibt. Die Kante, die zuvor ein erfolgloses Ergebnis ausgedrückt hat, drückt nun ein Scheitern oberhalb des Retrylimits aus.
Der Typ Ergebnis wird dafür definiert als:

$$Ergebnis\ E = \{t1_{Success}, t1_{Failure}, t1_{FinalFailure}, ...\}$$

Die Funktion fn_{AE}, die in dem jeweiligen Zustand q_{tn} das entsprechende API-Ergebnis AE berechnet, ist:
$fn_{AE} : \mathbb{N} \times AE \to E$
$maxSavepointExecutionCount_n \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{tn}$
$während\ der\ Ausführung\ der\ Saga$

$$fn_{AE}(x, y) = \begin{cases} En_{Success}, y = tn_{200} \\ En_{Failure}, y \neq tn_{200} \land x < maxSavepointExecutionCount_n \\ En_{FinalFailure}, else \end{cases}$$

Die Funktion fn_{IE}, die in dem jeweiligen Zustand q_{tn} das entsprechende interne Prozessergebnis IPE berechnet, ist:
$fn_{AE} : \mathbb{N} \times IPE \to Ergebnis$
$maxSavepointExecutionCount_n \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{tn}$
$während\ der\ Ausführung\ der\ Saga$

$$fn_{AR}(x, y) = \begin{cases} En_{Success}, y = tn_{Success} \\ En_{Failure}, y \neq tn_{Failure} \land x < maxSavepointExecutionCount_n \\ En_{FinalFailure}, else \end{cases}$$

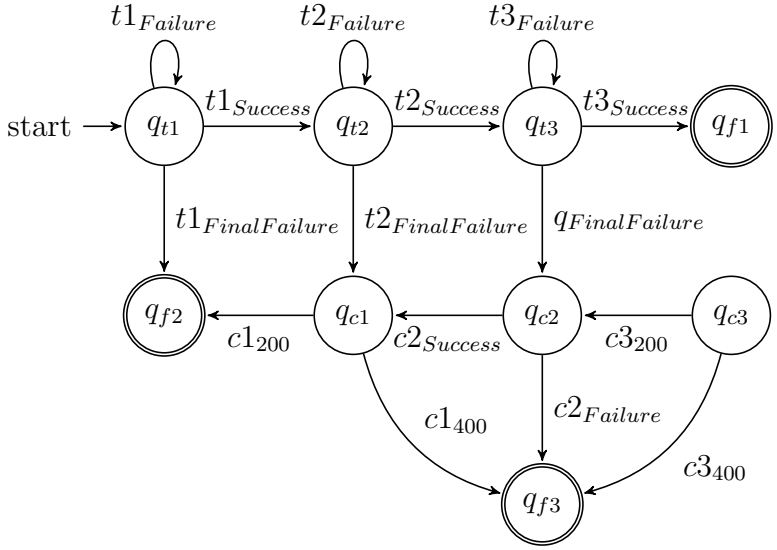


Abbildung 2.3: Saga als DEA

Es kann außerdem verboten werden, dass in einer Implementierung von Forward-Recovery der Fall verboten wird, der zu einer Backward-Recovery führt. Dabei wird erreicht, dass es nur einen gültigen Endzustand gibt. Dieser Endzustand drückt einen erfolgreichen Abschluss der Saga aus. Dabei ist zu beachten, dass das wiederholte Ausführen einer Aktion schlussendlich zu einem erfolgreichen Ergebnis führen muss.
Der DEA für dieses Verhalten sieht so aus:

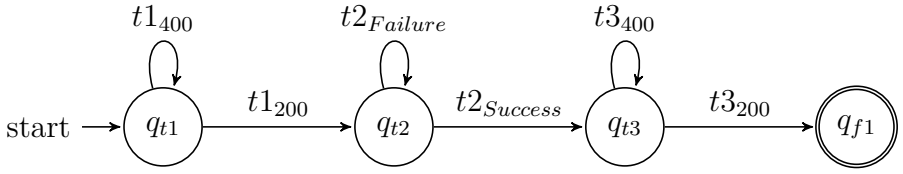


Abbildung 2.4: Saga als DEA

Es ist zu sehen, dass in diesem DEA keine Zustände enthalten sind, die eine Kompensierungsaktion ausdrücken. Somit geht in dieser Implementierung der Gedanke der Kompensierung verloren, der eine zentrale Rolle im Saga-Pattern innehat. Bei wiederholtem Auftreten eines erfolglosen Ergebnisses endet die Saga nie.

2.3.3.7.3 Voraussetzung für Forward-Recovery Damit eine Forward-Recovery sinnvoll ist, muss die Möglichkeit bestehen, dass ein gescheitertes T bei erneutem Ausführen ein erfolgreiches Ergebnis liefert. Das ist abhängig von der Semantik des Ergebnisses. Ist ein T beispielsweise ein Aufruf einer Schnittstelle zum Buchen eines Hotels, so könnten erfolglose Ergebnisse beispielsweise folgende Bedeutungen haben:

1. Hotel ist im angefragten Zeitraum ausgebucht
2. Hotel ist im angefragten Zeitraum im Betriebsurlaub

Im ersten Fall ist eine Forward Recovery möglich. Wenn ein andere Kunde seine Reservierung storniert, ist es es möglich, dass bei erneutem Anfragen eine Reservierung zustande kommt, die vorher abgelehnt wurde.
Im zweiten Fall ist Forward-Recovery ohne Effekt. Wenn eine Hotelbuchung für einen Zeitraum angefragt wird, in dem das Hotel im Betriebsurlaub ist, wird auch bei wiederholter Anfrage keine Buchung zustande kommen.

2.3.3.8 Implementierungsformen des Patterns

Um eine Saga als Microservice-System zu implementieren, gibt es zwei verschiedene Herangehensweisen. Die zwei Formen der Implementierung werden als Orchestrierung und als Choreografie bezeichnet. Beide Ausprägungen des Saga-Patterns verfolgen denselben Zweck: den Gedanken, eine globale verteilte Transaktion in einem verteilten System in lokale Teiltransaktionen aufzuteilen, die mittels passender Kompensierung zurückgerollt werden können.
Die zwei Ausprägungen unterscheiden sich hauptsächlich in der Softwarearchitektur. Es ist zu beachten, dass beide Implementierungen denselben Geschäftsprozess abbilden können und somit als äquivalent angesehen werden können.

Im Folgenden sollen die beiden Implementierungsansätze vorgestellt werden. Um die Unterschiede zu verdeutlichen, soll in den nachfolgenden Erläuterungen von einem Geschäftsprozess ausgegangen werden, der Ts enthält, die Teil einer verteilten, globalen Transaktion sind. Jedes T soll eine andere Schnittstelle aufrufen. Jedes T hat ein entsprechendes C zugeordnet. Sowohl die Ts als auch die Cs entsprechen den Anforderungen, die in XXX beschrieben sind.

2.3.3.8.1 Orchestration Die Orchestrierung zentralisiert die Logik für eine Saga in einem einzigen Service. Dieser Service wird als Koordinator oder Orchestrator bezeichnet. Der Koordinator ist verantwortlich für die Einhaltung der Transaktionsanforderungen. Er ruft aktiv die restlichen teilhabenden Services auf und muss die Ergebnisse der Aufrufe auswerten. Die teilhabenden Services haben nur Verantwortung für die Korrektheit der Prozessierung ihre eigenen Servicegrenzen. Ein solcher vom Koordinator aufgerufener Service hat keine Kenntnis vom ablaufenden Geschäftsprozess.
Der Orchestrator stellt einen Prozessmanager dar. Als solcher muss dieser Service garantieren, dass eine gestartete Saga nicht abbricht. Damit ein Absturz des Orchestrators dies gewährleisten kann, muss der Zustand der gestarteten Saga persistiert werden. Häufig wird das Transaktionslog in einer Datenbank gespeichert und erlaubt damit die Weiterführung der Saga auch nach Absturz der Anwendung.

2.3.3.8.2 Choreografie Bei der Choreographie gibt es keinen koordinierenden Service. Alle teilhabenden Services kennen den Ablauf des Geschäftsprozesses. Die Logik ist über alle Services verteilt.
Ein Service ist auch hier für die Korrektheit der Prozessierung innerhalb der eigenen Servicegrenzen verantwortlich. Zusätzlich muss jeder Service nach der Prozessierung den Prozess weiterführen. Dazu gehören sowohl mögliche weitere Transaktionen als auch mögliche Kompensierungsaufrufe.

2.3.3.8.3 Kommunikationsstrategien Die Orchestration unterstützt sowohl synchrone als auch asynchrone Kommunikation mit den teilhabenden Services.
Bietet ein an der globalen Transaktion teilhabender Service eine synchrone Schnittstelle zur Verfügung, muss der Koordinator warten, bis der aufgerufene Service eine Antwort liefert und ist solange blockiert. Bei einem Ausfall des aufgerufenen Services hat der Koordinator keine Möglichkeit, die Transaktion fortzufahren. Die Verfügbarkeit aller Services zum Aufrufzeitpunkt ist Voraussetzung für den erfolgreichen Abschluss einer orchestrierten Saga. Dafür ist dem Koordinator in einem solchen Fall die Unerreichbarkeit des Services bekannt und kann entsprechend reagieren.
Des weiteren kann ein Service eine asynchrone Schnittstelle zur Verfügung stellen. Diese Schnittstelle kann eine Implementierung des asynchronen Request-Response Musters sein (Polling Pattern, Callback Pattern). Um eine asynchrone Request-Response Schnittstelle zu verwenden muss der Orchestrator das entsprechende Protokoll des Musters einhalten. Der Vorteil in der Verwendung asynchroner Kommunikation liegt darin, dass der Orchestrator nicht blockiert. In der Zeit zwischen der Platzierung der Anfrage und dem Erhalt der Antwort kann der Orchestrator die Prozessierung der aktuellen Saga pausieren und mit

der Verarbeitung anderer Anfragen fortfahren. Der Vorteil dieser Implementierungen ist die Entkopplung von Request und Response. Das zahlt sich in Fällen aus, in denen die Verarbeitung der Anfrage einen längeren Zeitraum in Anspruch nimmt.
Die Implementierung eines asynchronen Request-Response Musters ist wesentlich komplizierter als die Entwicklung einer synchronen Schnittstelle. Deshalbt sollte dies als Implementierung einer lokalen Transaktion unter Verwendung einer Orchestrierung nur in Szenarien gewählt werden, die die Entkopplung von Anfrage und Antwort voraussetzen.
Schlussendlich bietet die Orchestrierung die Möglichkeit, asynchrone Messaging-Komponenten zu verwenden. Anstatt direkt miteinander zu kommunizieren platziert der Koordinator die Anfrage als Event in einer Messaging-Middleware und kann mit der Prozessierung der Saga pausieren. Der angefragte Service erhält dieses Event und kann eine beliebig lang andauernde Verarbeitung ausführen. Nachdem die Verarbeitung abgeschlossen ist, kann die Antwort wiederum als Event in der Middleware platziert werden. Der Koordinator erhält dieses Event und kann darin das Ergebnis ablesen.
Um eine Saga mittels Choreographie zu implementieren, sollte asynchrones Messaging verwendet werden. Da die Geschäftslogik über alle Komponenten verteilt ist, ist selten ein Service am Ergebnis des nächsten Transaktionsschrittes interessiert. Ein Service S_1 verarbeitet seinen Teil der Transaktion und informiert den nächsten Service S_2 über den Erfolg der Berechnung. S_2 ist so implementiert, dass er die Logik für seine eigenen Berechnungen kennt. Somit muss S_1 nicht über den Erfolg informiert werden. Ein Erfolg von der in S_2 ablaufenden Transaktion endet in einem Event für einen nachfolgenden Service S_3. Die Kommunikation ist hier nicht auf ein Request-Response Muster ausgelegt, es werden Einweg-Nachrichten genutzt. Die Ausnahme ist ein erfolgloses Ergebnis in S_2. In diesem Fall wird S_3 nicht per Event informiert. Es wird lediglich S_1 mit einem erfolglosen Ergebnis benachrichtigt. Als Reaktion auf dieses Event kann S_1 mit Forward- oder Backward-Recovery reagieren.
Die Implementierung einer Choreographie per Request-Response Muster ist nicht unmöglich. S_1 ruft S_2 per synchroner oder asynchroner Request-Response Schnittstelle auf. Daraufhin erhält S_1 eine Antwort mit dem Ergebnis von der Berechnung von S_2. Bei einem Erfolg findet in S_1 jedoch keine Reaktion statt. Lediglich bei einem Misserfolg muss S_1 Kenntnis vom Ergebnis der Transaktion in S_2 haben. Somit hat die Verwendung einer Response nur einen Nutzen, falls ein Misserfolg vorliegt.
Des Weiteren hat die Verwendung einer synchronen Kommunikation in der Implementierung der Choreographie den Nachteil, dass es zu Blockierungen aller teilhabenden Services führt. Auch S_2 ruft S_3 synchron auf. Somit muss S_2 warten, bis die Response in S_3 erfolgt. Erst danach kann S_2 die Response für S_1 absenden. Dieses Verhalten wird als Aufrufkaskadierung bezeichnet und wirkt sich sowohl auf den Fall eines Erfolgs als auch den eines Misserfolgs aus.
Aus den genannten Gründen ist es zu empfehlen, bei der Implementierung einer Saga per Choreographie eine eventbasierte Architektur mit asynchronen Messaging-Komponenten zu verwenden.

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

2.3.4.1 Langlebige Transaktionen - LLT

2.3.4.2 Bezug auf den Geschäftsprozess

2.3.4.3 Verteilte Systemlandschaft

2.3.4.4 Reaktion auf verschiedene Antwortmöglichkeiten in der Geschäftslogik

2.3.4.5 Fehlerfälle - Geschäftslogik und Ausfälle

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

3 Versuchsvorbereitung

3.1 Problemstellung

In 4 wurde das Saga-Pattern als ein Implementierungsmuster vorgestellt, welches ermöglichen soll, die ACID-Anforderungen in einem verteilten System nachzubilden.
Ein verteiltes System steht immer vor der Herausforderung von Netzwerkfehlern. Verwenden Transaktionsteilnehmer eines Systems die Request-Response-Kommunikation, so besteht immer die Möglichkeit, dass einzelne Nachrichten ihr Ziel nicht erreichen. Wenn eine solche Kommunikation deterministischer Natur ist, kann der Sender seinen Request ohne Gefahr wiederholen. Die im Saga-Pattern verwendeten lokalen Transaktion stellen jedoch keine deterministische Abfrage dar, sondern verfolgen das Ziel eines Zustandswechsels des Empfängers. Wiederholt der Sender seine Requests, führt dies zu ungewünschten Nebenwirkungen.
Es stellt sich die Frage, ob in Saga-Systemen, die Request-Response-Kommunikation verwenden, Konsistenz gewährleistet werden kann.

3.2 Zielstellung

Die folgenden Kapitel dienen dem Zweck, das Saga-Pattern hinsichtlich Systemkonsistenz zu untersuchen. Dabei wird davon ausgegangen, dass jegliche Kommunikation per Request-Response-Protokolle abläuft.

1 Es ist die Frage zu beantworten, unter welchen Bedingungen ein Microservice-System, welches mittels Saga-Pattern implementiert wurde, eine LLT abbilden kann.

2 Es sind Fehlerquellen und Fehlertypen zu identifizieren, die einen inkonsistenten Systemzustand verursachen können. Es sollen Lösungen im Rahmen des Saga-Patterns formuliert, implementiert und evaluiert werden.

3 Es soll eine Antwort darauf gefunden werden, welche Kriterien eine Schnittstelle erfüllen muss, um an einer LLT teilnehmen zu können.

4 Es soll beantwortet werden, ob aufgrund Netzwerkpartitionen auftretende Fehler in das Saga-Pattern integrierbar sind.

3.3 Ausgangspunkt

Es soll ein Geschäftsvorgang mittels Saga-Pattern in einem Microservicesystem entworfen, implementiert und bewertet werden. Die gewählte Geschäftsvorgang soll als LLT aufgefasst werden und eine verteilte Transaktion abbilden.
Es soll eine Implementierungsform des Saga-Patterns gewählt werden. In der Zielstellung wird ein System gefordert, welches Request-Response-Kommunikation verwendet. Somit kann die Verwendung von Messagingkomponenten ausgeschlossen werden. Die auf diese Komponenten ausgerichtete Implementierung per Choreografie wird deshalb nicht gewählt. Als Implementierungsform des Saga-Patterns wird für diesen Versuch die Orchestrierung gewählt.

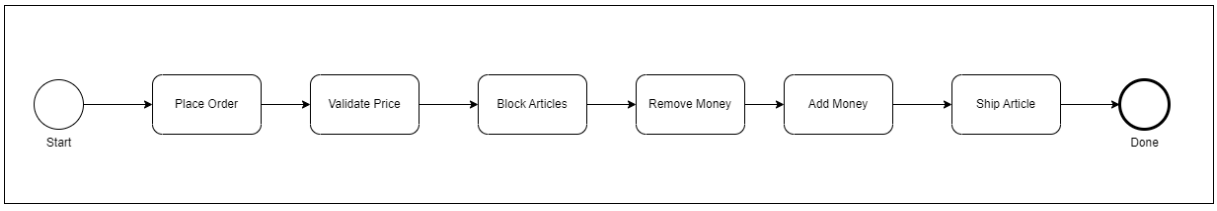
3.4 Methodik

In diesem Abschnitt soll das Vorgehen bei der Bearbeitung des Problems erläutert werden.

3.4.1 Schritt 1 - Entwurf und Implementierung eines GP

Entwurf eines Geschäftsprozesses Zur Untersuchung der Probleme soll ein Geschäftsprozess entworfen und als Saga-System implementiert werden. Der Durchlauf durch diesen Geschäftsprozess soll als LLT aufgefasst werden. Der Geschäftsprozess muss eine der in **??** beschriebenen Eigenschaften aufweisen, damit die Realisierung dieses Prozesses innerhalb einer Saga sinnvoll ist.
Es besteht die Anforderung an den zu entwerfenden Prozess, dass dieser durch eine Menge von lokalen Transaktionen abbildbar sein muss. Außerdem muss es möglich sein, Kompensierungen für jede dieser Transaktionen zu formulieren.

Imaginärer Bestell- und Lieferprozess 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.

Implementierung Nachdem der Geschäftsprozess definiert wurde, soll das Saga-System implementiert werden. In 2.3.3.4 wurde erläutert, wie die im Koordinator laufende SEC beschrieben werden kann. Die Implementierung soll durch diese Darstellungsform beschrieben werden können.
Für die Transaktionsteilnehmer gilt zu Beginn der Implementierung lediglich die Anforderung, dass die Schnittstellen per Request-Response-Muster aufgerufen werden. Dafür wird das Http-Protokoll verwendet.

Feedbackschleife

3.4.2 Schritt 2 - Messung der verschiedenen Implementierungen

Die Bewertung der Implementierungen soll auf Grundlage von Messdaten erfolgen. Es soll nun beschrieben werden, wie die Erfassung dieser Daten erfolgen soll.

Systemtest Die Messdaten erfolgen in einer produktionsähnlichen Umgebung im Rahmen von Systemtests. Das erwartete Ergebnis ist ein konkreter Endzustand, der erreicht werden soll. Dieser Endzustand kann mit dem erreichten Endzustand verglichen werden.

Testkonfiguration Ein solcher Systemtest wird unter einer bestimmten Konfiguration durchgeführt. Die Konfiguration setzt sich zusammen aus einem Testcase und einem Netzwerkszenario.

Testcase Ein Testcase stellt eine konkrete Interaktion mit dem System dar. Die Testcases übernehmen die Aufgabe, die verschiedenen Fälle der Geschäftslogik auf Korrektheit zu überprüfen.

Netzwerkszenarien Ein Netzwerkszenario ist ebenfalls Teil der Testkonfiguration. Wird ein Systemtest durchgeführt, so beschreibt das Netzwerkszenario das konkrete Netzwerkverhalten.

Messgegenstand Das Ziel der Messung ist, Aussagen über die Konsistenz des Systems zu treffen. In Abschnitt **??** wurden verschiedene Vorgehensweisen dargestellt, wie die Konsistenz eines Systems gemessen werden kann.
Die verwendete Implementierung per Orchestrierung setzt das Vorhandensein eines Koordinators voraus. Die darin befindliche Steuerung der LLT durch die SEC gibt Auskunft über die ausgeführten lokalen Transaktionen einer Saga. Es kann für jede lokale Transaktion gemessen werden, wie oft die SEC davon ausgeht, dass eine Transaktion ausgeführt wurde. Analog dazu kann die tatsächlich ausgeführte Anzahl an Transaktionen bestimmt werden, indem die Sicht der Transaktionsteilnehmer verwendet wird. Stimmen die korrespondierenden Werte aller lokalen Transaktionen in beiden Sichten überein, kann davon ausgegangen werden, dass die LLT keine Inkonsistenzen in das System eingeführt hat.
Ein weiterer Anhaltspunkt für Konsistenzanomalien ist der erreichte Endzustand einer Transaktion. In 2.3.3.4 wurde ein Endzustand für einen DEA definiert, der erreicht wird, nachdem eine Saga in einer kompensierenden Aktion eine Fehlerantwort erhält. In solchen Fällen ist der Koordinator nicht in der Lage die LLT abzuschließen und endet in einem weder erfolgreichen noch kompensierten Zustand.
Da dieser Zustand einen Fall aufzeigt, in dem die Atomarität der LLT verletzt wird, ist das Auftreten solcher Endzustände ein unmittelbares Zeichen für Inkonsistenz.

Metriken TODO Beschreibung, wie die Messdaten in eine (oder mehrere) Kennzahlen umgewandelt wird

3.4.3 Schritt 3 - Analyse der Messdaten

4 Implementierung des Saga-Systems

4.1 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

4.2 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	

4.3 Fachliche Kontextabgrenzung

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.

4.3.1 Frontend

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:

- Zahlungsinformationen des Kunden: BankId, UserId
- Liste der zu bestellenden Artikel, mit Artikel: ArticleId, ArticlePrice, Amount

Dieses Objekt kann an das Backend gesendet werden.

4.3.2 ArticleService

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 ArticleService eine Produktdatenbank. Da dieser Service ausschließlich die Produktdaten als Ressource behandelt, kann er RESTful implementiert werden.

4.3.3 StockService

Der Service soll den aktuellen Bestand an vorrätigen Artikeln abbilden. Es soll möglich sein, eine Menge an Artikeln für eine konkrete Bestellung zu reservieren. Eine Reservierung von einer Menge von Artikeln wartet auf die Auslösen der Lieferung.

Um dies zu erlauben, muss der aktuelle Lagerstand in einer Tabelle hinterlegt sein. Die Tabelle muss den aktuell verfügbaren Bestand pro Artikel ausdrücken.
Um eine Reservierung zu ermöglichen, muss es eine weitere Tabelle geben, die eine Menge von blockierten Artikeln für einen bestimmten Bestellprozess enthält. 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.
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. Diese Transaktion soll den physischen Prozess abbilden, die bestellten und für diese Bestellung blockierten Artikel aus dem Lager in das Transportfahrzeug und schließlich zum Kunden zu transferieren.
Die Übergabe der Ware an den Kunden stellt den finalen Schritt des Prozesses dar. Ist dies geschehen, gibt der Lieferant dem StockService die Bestätigung für die gelieferte Bestellung.
Die Blockierung eines Artikels muss kompensiert werden können, da sonst der reservierte 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.
Die Auslösung der Lieferung ist nur bedingt kompensierbar. Nachdem das Transportfahrzeug mit der Ware losgefahren ist und die Ware noch nicht übergeben hat, kann die Lieferung noch abgebrochen und somit kompensiert werden. Die Kompensierung muss also den Lieferant benachrichtigen und die Ankunft der Waren bestätigen. Nach der Saga-Definition soll die zu kompensierende Transaktion zurückgerollt werden. Deshalb werden die Waren aus der Lieferungstabelle zurück in die Reservierungstabelle geschrieben.
Der Abschluss einer Lieferung bildet die physische Warenübergabe an den Kunden ab. Eine Kompensierung ist hier nicht möglich. Da diese lokale Transaktion die letzte Transaktion ist und einen erfolgreichen Abschluss der Saga zur Folge hat, muss hier keine Kompensierung angeboten werden.

4.3.4 BankingServices

Im Geschäftsprozess wurde definiert, dass die Transaktion den Geldbetrag des Kundenkontos und des Händlerkontos in zwei separaten lokalen 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 muss Anfragen zum Geldabbuchung ablehnen, wenn die Verringerung den Kontostand in den negativen Bereich fallen lassen würde. In diesem Fall wird die Transaktion abgebrochen.
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.

4.3.5 OrderService

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. Die Tabelle, in der die Schritte gespeichert werden, stellt das Saga Execution Log dar. Außerdem ruft der Service nach Feststellung eines Misserfolgs die Backward-Recovery auf.
Neben der Schnittstelle zum Platzieren der Bestellung soll dem Nutzer ermöglicht werden, die Bestellung zu stornieren. Dieser zusätzliche Endpunkt nimmt die Vorgangsnummer der zuvor ausgelösten Bestellung entgegen. Falls die Ware noch nicht beim Kunden eingetroffen ist, kann hier die Bestellung abgebrochen werden. Eine solche Stornierung löst ebenfalls Backward-Recovery aus und soll den Initialzustand wiederherstellen. Diese Aktion stellt keine lokale Transaktion der globalen Transaktion dar; es ist eine Aktion, die von außen in den Bestellprozess eingreift. Damit ist die Stornierung kein T und hat somit kein zugehöriges C.

4.4 Technische Kontextabgrenzung

Bei der Modellierung der Datenbankschicht wurde das Database-per-Service Muster verwendet. Dieses Muster erlaubt eine sehr lose Bindung der Services und lässt die Datenbasis in der Verantwortung eines Services. Es ist möglich, ein Microservice-System zu implementieren, welches das Muster einer gemeinsamen Datenbank verwendet. Dies würde die Implementierung atomaren und konsistenten Transaktionen vereinfachen. Die

Voraussetzungen für eine solche Implementierungen sind jedoch nicht immer gegeben, da an der LLT teilhabende Services unter Umständen nicht in der Verantwortung des Entwicklerteams liegen könnten, welches die LLT zu implementieren hat. Aus diesem Grund wird im Rahmen der für diese Implementierung vorzunehmende Implementierung jeder Service als getrennte Komponente aufgefasst, die als alleinige Instanz auf ihre eigenen Daten zugreift und zugreifen kann.

4.4.1 ArticleService

4.4.1.1 Datenbankschicht

Der ArticleService soll lediglich Daten aus der Datenbanksicht zurückliefern. Deshalb wird für diesen Service lediglich eine Tabelle benötigt. Die Tabelle *article* enthält für jeden anzubietenden Artikel den zugehörigen Namen und Preis.

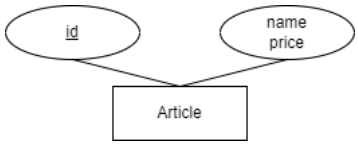


Abbildung 4.1: ER-Diagramm ArticleService

4.4.1.2 Schnittstellen

Die Produktdaten sollen von dem Frontend abgefragt und dargestellt werden. Dabei werden alle vorhandenen Produktdaten selektiert und zurückgegeben.
Zusätzlich soll der Koordinator eine gezielte Abfrage durchführen können, welche den Preis eines Artikels für eine gegebene ProduktId liefert. Beide Schnittstellen sind RESTful, da jeder Artikel als Ressource angesehen wird.

Endpunkt	Http-Methode	Argumente
/api/articles	GET	
/api/articles/{id}	GET	ArticleId

4.4.2 StockService

4.4.2.1 Datenbankschicht

Der StockService verwaltet vier Tabellen. Die zentrale Tabelle ist *articlestock* und stellt für jeden bekannten Artikel den auf Lager befindlichen Vorrat dar. Eine Reservierung für eine gegebene Vorgangsnummer wird in der Tabelle *blockedarticles* durch alle Tupel, bei denen diese Nummer auftritt. Die Lieferungen werden in der Tabelle *shipments* dargestellt. Auch hier gehört eine Vorgangsnummer zu den Spalten. Zusätzlich gibt es das Attribut *hasarrived*, was Auskunft über den Status der Lieferung gibt. Die zu einer Lieferung gehörenden Artikel und deren Menge sind in der Tabelle *shippedarticles* enthalten.

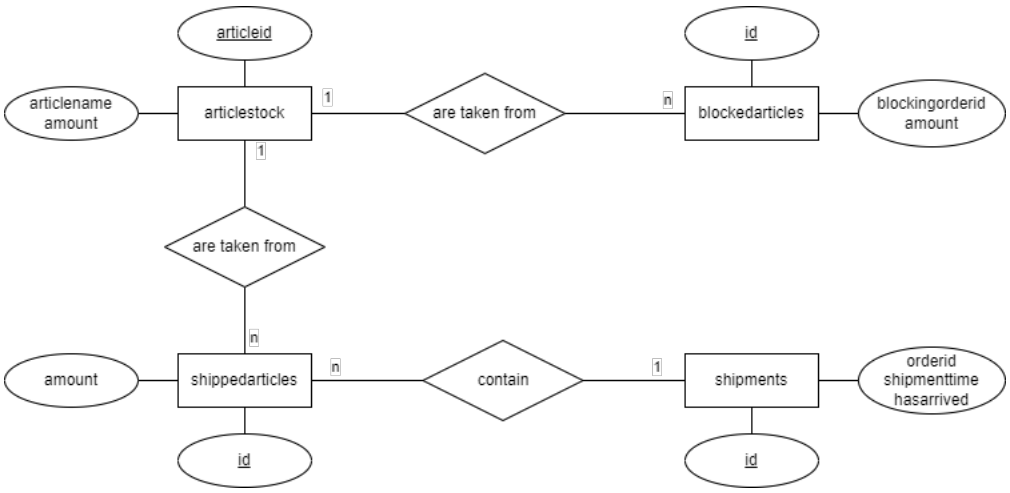


Abbildung 4.2: ER-Diagramm StockService

4.4.2.2 Schnittstellen

Der Endpunkt $/api/blocked-articles$ überträgt frei verfügbare Artikel aus der Tabelle *articlestock* in die Tabelle *blockedarticles*. Als Argumente werden die zu blockierenden Artikel-Ids und Artikel-Mengen sowie die Vorgangsnummer benötigt. Teil der lokalen Transaktion sind folgende Schritte:

1. Reduzieren jedes Artikelvorrats in der Tabelle *articlestock*
2. Einfügen eines Elements in der Tabelle *blockedarticles* für jede zu blockierende Artikel-Id

Der Endpunkt $/api/blocked-articles-compensation$ stellt die Kompensierung für die Schnittstelle $/api/blocked-articles$ dar. Als Argument wird hier lediglich die Vorgangsnummer benötigt. Notwendige Schritte der auszuführenden lokalen Transaktion sind:

1. Selektieren aller Elemente aus $blocked-articles$ mit der angefragten Vorgangsnummer
2. Löschen dieser Elemente
3. Erhöhen der Vorratsmengen in *articlestock* für jede Artikelblockierunge

Der Endpunkt $/api/start-shipment$ erwartet die Vorgangsnummer als Argument. Die reservierten Artikel werden in eine Lieferung umgewandelt und versendet. Die Schritte der ablaufenden lokalen Transaktion sind:

1. Initialisieren eines Elements in *shipments* mit dem Status $hasarrived = 0$
2. Selektieren und Löschen aller Elemente in *blockedarticles*, die die OrderId enthalten
3. Einfügen der Selektierungen in der Tabelle *shippedarticles*

Der Endpunkt $/api/finish-shipment$ wird ausschließlich vom Lieferanten verwendet und dient zur Bestätigung der Lieferung. Es wird lediglich der Status der entsprechenden ShipmentId in der Tabelle *shipments* auf 1 gesetzt.

Der Endpunkt $/api/shipments$ liefert lediglich den Status der angeforderten ShipmentId. Der Koordinator hat mit dem Aufruf von $/api/start-shipments$ die Lieferung ausgelöst. Der Koordinator hat mit der Schnittstelle $/api/shipments$ die Möglichkeit, den Status der Lieferung solange abzufragen, bis der Lieferant per Aufruf von $/api/finish-shipment$ den Lieferabschluss bestätigt. Die Kombination von $/api/start-shipment$, $/api/finish-shipment$ und $/api/shipments$ können als Polling-Implementierung eines asynchronen Request-Response Musters aufgefasst werden.
Der letzte Endpunkt des StockServices ist $/api/cancel-shipment$ und bietet dem Koordinator die Möglichkeit, auf eine Stornierung zu reagieren. Bis die Lieferung abgegeben und bestätigt wurde, kann diese Schnittstelle verwendet werden, um die Lieferung abzubrechen. Als Argument wird die ShipmentId benötigt.

Endpunkt	Http-Methode	Argumente
/api/blocked-articles	POST	ArticleId, Amount, OrderId
/api/blocked-articles-compensation	POST	OrderId
/api/start-shipment	POST	OrderId
/api/finish-shipment	POST	ShipmentId
/api/shipments	GET	ShipmentId
/api/cancel-shipment	POST	ShipmentId

4.4.3 BankingService

4.4.3.1 Datenbankschicht

Zur Verwaltung des BankingServices gehören drei Tabellen. Die Tabelle *bankuser* enthält alle Nutzer, die Tabelle *bankusercredit* ordnet jedem Nutzer einen Kontostand zu und die Tabelle *bankusertransaction* stellt Veränderung des Kontostands der Tabelle *bankcredit* in der Tabelle *bankusertransaction* als Historie dar.

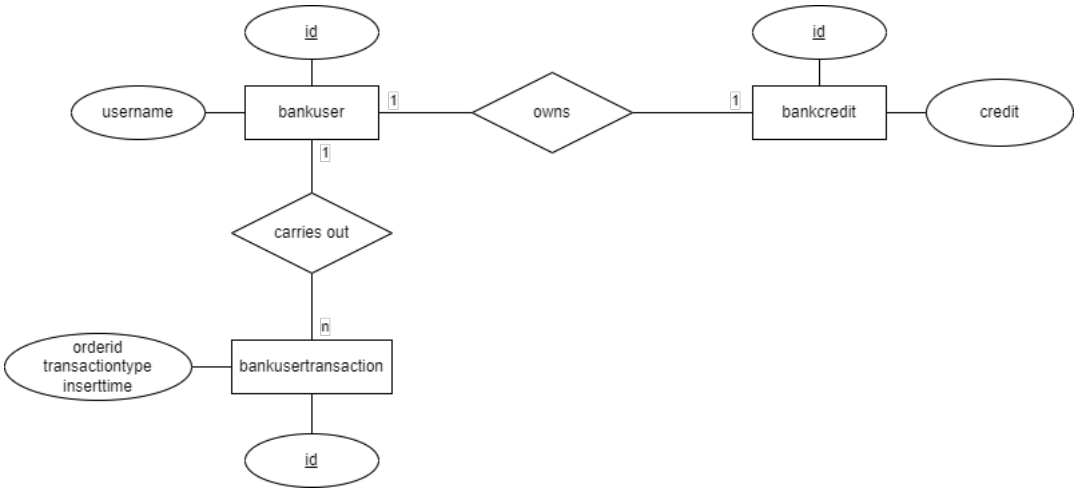


Abbildung 4.3: ER-Diagramm BankingService

4.4.3.2 Schnittstellen

Der BankingService bietet jeweils eine Schnittstelle zum Erhöhen und zum Verringern des Kontostandes an. Dabei wird lediglich in der Tabelle *bankcredit* das Credit-Attribut erhöht oder verringert. Die Spalte ist im Datenbankmanagementsystem mit einer Einschränkung versehen, die verhindert, dass der Wert der Spalte unter 0 fällt.
Die Veränderung des Kontostandes ist der erste Schritt der lokalen Transaktion. Der zweite Schritt ist das Einfügen eines neuen Elementes in der Tabelle *bankusertransaction*. In dieser Tabelle wird der Transaktionstyp festgehalten, der den Typ

Endpunkt	Http-Methode	Argumente
/api/add-money	POST	UserId, Amount, OrderId
/api/add-money-compensation	POST	UserId, Amount, OrderId
/api/remove-money	POST	UserId, Amount, OrderId
/api/remove-money-compensation	POST	UserId, Amount, OrderId

4.5 Darstellung als DEA

Der entworfene Prozess soll nun als DEA dargestellt werden. Dafür sind alle möglichen Ergebnisse zu erfassen, die jede lokale Transaktion aus Sicht des Koordinators liefern kann.

4.5.1 Ergebnisse aller Transaktionen

Transaktion	Ergebnis	Bedeutung
Initialize Saga	Success	Bestellung ist initialisiert
Get Article Data	200	Artikeldaten wurden vom ArticleService empfangen
	404	Artikel wurde nicht gefunden
Validate Price	Success	Preis aus Bestellung und aus dem System stimmen überein
	Failure	Preise aus Bestellung und aus dem System stimmen nicht überein
Block Articles	200	Produkte wurden reserviert
	208	Artikel wurden bereits blockiert
	409	Lagervorrat ist erschöpft
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Remove Money	200	Geldbetrag auf dem Konto wurde verringert
	208	Geldbetrag wurde auf vorherige Anfrage bereits verringert
	409	Lokale Transaktion ist fehlgeschlagen (Konto ist nicht gedeckt)
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Add Money	200	Geldbetrag auf dem Konto wurde erhöht
	208	Geldbetrag wurde auf vorherige Anfrage bereits erhöht
	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl

Start Shipment	200	Lieferung wurde ausgelöst
	208	Lieferung wurde auf vorherige Anfrage bereits gestartet
	404	Zur geforderten Bestellnummer wurde keine Reservierung gefunden
	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Check No Cancellation	Success	Es wurde keine Stornierung festgestellt
	Failure	Es wurde eine Stornierung festgestellt
Get Shipment Status	200	Status wurde vom StockService empfangen
	404	Lieferung existiert nicht
Check Shipment Status	Success	Lieferstatus signalisiert abgeschlossene Lieferung
	Failure	Lieferstatus signalisiert noch nicht abgeschlossene Lieferung

4.5.2 Ergebnisse aller Kompensierungen

Transaktion	Ergebnis	Bedeutung
Initialize Saga Compensation	Success	
Get Article Data Compensation	Success	
Validate Price Compensation	Success	
Block Articles Compensation	200	Reservierung wurde aufgehoben
	404	Reservierung wurde nicht gefunden
	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Remove Money Compensation	200	Geldbetrag auf dem Konto wurde erhöht
	208	Transaktion wurde auf vorherige Anfrage bereits kompensiert
	404	keine zugehörige kompensierbare Transaktion gefunden
	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Add Money Compensation	200	Geldbetrag auf dem Konto wurde erhöht
	208	Transaktion wurde auf vorherige Anfrage bereits kompensiert
	404	keine zugehörige kompensierbare Transaktion gefunden

	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Start Shipment Compensation	200	Lieferung wurde abgebrochen
	404	Lieferung existiert nicht
	409	Lokale Transaktion ist fehlgeschlagen
	410	Lieferung wurde nicht abgebrochen, da sie bereits abgeschlossen ist
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl

Unter Berücksichtigung der dargestellten Antwortmöglichkeiten jeder lokalen Transaktion kann folgender Zustandsautomat formuliert werden.

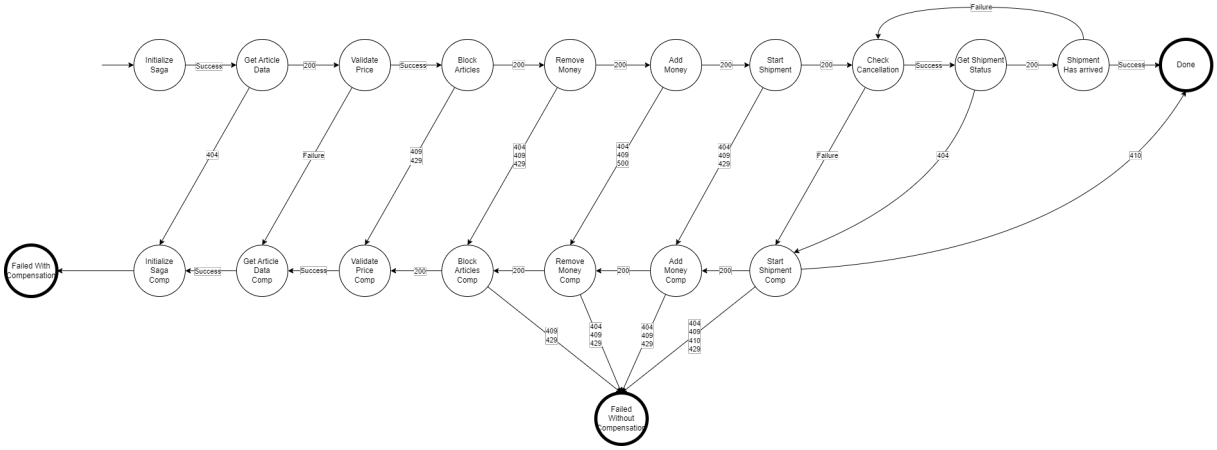


Abbildung 4.4: Saga als DEA

4.5.2.1 Durchlauf einer erfolgreichen Saga

4.5.2.2 Durchlauf einer gescheiterten, kompensierten Saga

4.5.2.3 Durchlauf einer gescheiterten, nichtkompensierten Saga

4.6

5 Implementierung

5.1 Abänderung der Implementierung

Abbildungsverzeichnis

2.1 Saga als DEA 10
2.2 Saga als DEA 12
2.3 Saga als DEA 13
2.4 Saga als DEA 14

4.1 ER-Diagramm ArticleService 26
4.2 ER-Diagramm StockService 27
4.3 ER-Diagramm BankingService 28
4.4 Saga als DEA 31

Literatur

[1] H. Garcia-Molina und K. Salem, "Sagas," *ACM SIGMOD Record*, Jg. 16, Nr. 3, S. 249–259, 1987, ISSN: 0163-5808. DOI: 10.1145/38714.38742.