Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Wieso, Weshalb, Warum?

1.2 Zielsetzung dieser Arbeit

These: Mittels Saga-Pattern implementierte lang andauernde Transaktionen (LLT) ermöglichen 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 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 Kapitel wurde die SEC 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 enthält 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:
$API - Response\ AR \in \{tn_{200}, tn_{201}, tn_{400}, tn + 1_{200}, tn + 1_{400}, tn + 1_{409}, ...\}$.
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:
$Interner\ Prozessschritt\ IP \in \{tn_{Success}, tn_{Failure}, tn + 1_{Success}, tn + 1_{Failure}, ...\}$.
Ein Ergebnis einer Aktion wird also definiert als:
$Ergebnis\ E = API - Response \cup Interner\ Prozessschritt$.
Das Eingabealphabet beinhaltet Elemente aus dem Ergebnistyp:
$\Sigma \in Ergebnis$.
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:
$q_1, q_2 \in Q \land u, v \in \Sigma$
$K = (q_1, u) \in Q \times \Sigma^\star$
Der Automat wechselt in einen Folgezustand, indem er ein Wort aus dem Eingabealphabet Σ abarbeitet und eine passende Übergangsrelation in δ findet. Somit gilt:
$(q_1, u) \vdash (q_2, v), falls\ \delta(q_1, u) = q_2$
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
$Q = \{q_{t1}, q_{t2}, q_{t3}, q_{c1}, q_{c2}, q_{c3}, q_{f1}, q_{f2}, q_{f3}\}$,
$\Sigma = \{t1_{200}, t1_{400}, t2_{Success}, t2_{Failure}, t3_{200}, t3_{400}, c1_{200}, c1_{400}, c2_{Sucess}, c2_{Failure}, c3_{200}, c3_{400}\}$,
$\delta = \{((q_{t1}, t1_{200}), q_{t2}), ((q_{t2}, t2_{Sucess}), q_{t3}), ((q_{t3}, t3_{200}), q_{f1}),$
$((q_{t1}, t1_{400}), q_{f2}), ((q_{t2}, t2_{Failure}), q_{c1}), ((q_{t3}, t3_{400}), q_{c2}),$
$((q_{c1}, c1_{200}), q_{f2}), ((q_{c2}, c2_{Sucess}), q_{c1}), ((q_{c3}, c3_{200}), q_{c2}),$
$((q_{c1}, c1_{400}), q_{f3}), ((q_{c2}, c2_{Failure}), q_{f3}), ((q_{c3}, c3_{Failure}), q_{f3})\}$,
$q_0 = q_{t1}$,
$F = \{q_{f1}, q_{f2}, q_{f3}\}$.

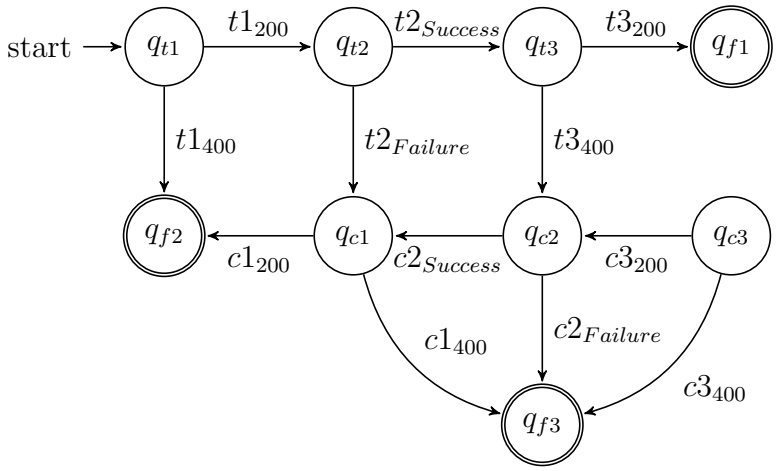


Abbildung 2.1: Saga als DEA

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

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:
$(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#)$
$\vdash (q_{t2}, t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#))$
$\vdash (q_{t3}, t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#))$

$\vdash (q_{c2}, c2_{Success} \circ c1_{200} \circ \#))$
$\vdash (q_{c1}, c1_{200} \circ \#))$
$\vdash (q_{f2}, \#))$

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

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} das Ergebnis des internen Prozessschritts IP berechnet, sieht so aus:

$f : \mathbb{N} \to Interner\ Prozessschritt$
$maxSavepointExecutionCount \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{sp1}$
$w\ddot{a}hrend\ der\ Ausf\ddot{u}hrung\ der\ Saga$

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

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.

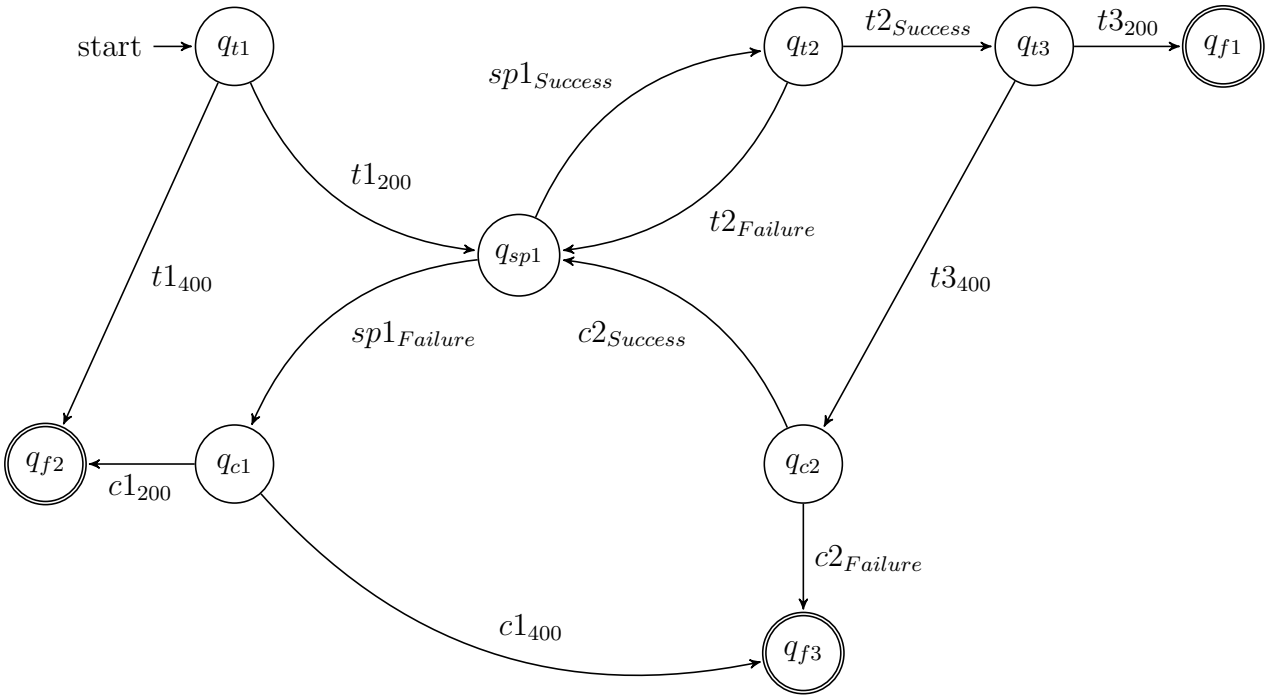


Abbildung 2.2: Saga als DEA

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_{AR}, die in dem jeweiligen Zustand q_{tn} das entsprechende Ergebnis für eine API-Response berechnet, ist:

$fn_{AR} : \mathbb{N} \times (API - Response) \to Ergebnis$
$maxSavepointExecutionCount_n \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{tn}$

während der Ausführung der Saga

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

Die Funktion fn_{IP}, die in dem jeweiligen Zustand q_{tn} das entsprechende Ergebnis für einen internen Prozessschritt berechnet, ist:
$fn_{AR} : \mathbb{N} \times (Interner\ Prozessschritt) \rightarrow Ergebnis$
$maxSavepointExecutionCount_n \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{tn}$
während der Ausführung der Saga

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

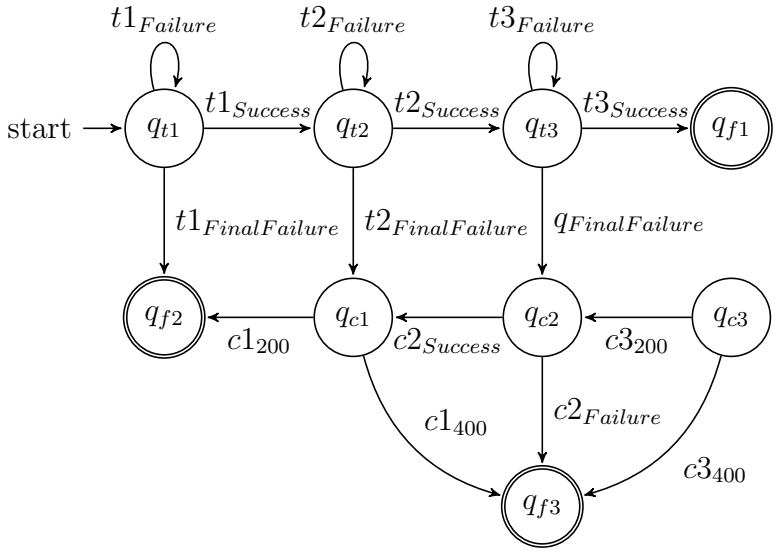


Abbildung 2.3: Saga als DEA

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:

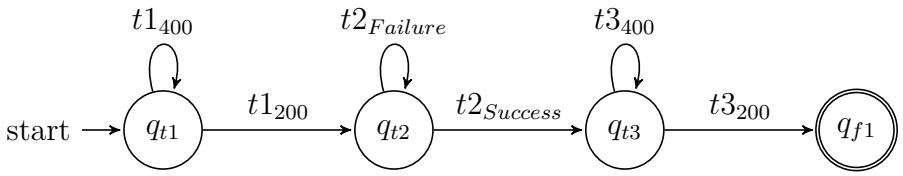


Abbildung 2.4: Saga als DEA

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 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.

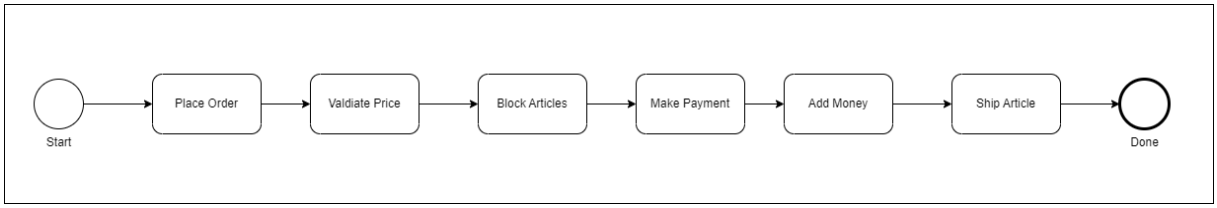
4 Erstes Saga-System

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