HAW HAMBURG

BACHELOR THESIS
Olga Nesterova

Transaktionen und Resilienzstrategie in Microservices-Architekturen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Olga Nesterova

Transaktionen und Resilienzstrategie in Microservices-Architekturen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 06. Mai 2025

Olga Nesterova

Thema der Arbeit

Transaktionen und Resilienzstrategie in Microservices-Architekturen

Stichworte

Microservices, Transaktionen, ACID, Resilienz, verteilte Systeme, Eventual Consistency

Kurzzusammenfassung

Die Umsetzung von Transaktionen in Microservices-Architekturen ist herausfordernd. Das traditionelle Konzept der ACID-Eigenschaften wird in diesem Kontext in Frage gestellt, da Services unabhängig voneinander agieren, ihre eigenen Daten verwalten und hauptsächlich über Netzwerke kommunizieren.

Die Arbeit behandelt das Problem, dass Transaktionen in Microservices-Architekturen aus verschiedenen Perspektiven betrachtet werden können. Jeder Microservice verwaltet seine eigenen lokalen Transaktionen. Manchmal müssen auch serviceübergreifende Transaktionen durchgeführt werden. Bei der Umsetzung solcher Transaktionen werden die traditionellen ACID-Eigenschaften herausgefordert.

Man soll dabei auch mögliche Probleme mit Netzwerkstörungen und Systemausfällen in Betracht ziehen, um die Systemstabilität sicherzustellen. In dieser Arbeit haben wir auch untersucht, wie wir stabile, skalierbare und resiliente Systeme entwickeln können, die serviceübergreifende Transaktionen durchführen können.

Um all diese Herausforderungen anzugehen, wird zuerst nach vorhandenen Strategien und Mustern recherchiert. Danach wird ein Prototyp konzipiert und umgesetzt, um zu zeigen, wie durch den Einsatz passender Muster Koordination und Fehlertoleranz in einer tatsächlichen Microservice-Architektur umgesetzt werden können. Im Anschluss wird der Prototyp auch evaluiert. Die Ergebnisse zeigen, dass gewisse Muster eine Umsetzung von stabilen Abläufen über verschiedene Services hinweg ermöglichen. Es wäre dabei ratsam, nicht nur blind strenge Konsistenz anzustreben, sondern bei Bedarf bereit zu sein, Konsistenzgarantien abzuschwächen. Im Endeffekt verlassen wir uns darauf, dass unser System irgendwann später konsistent sein wird.

Zum Schluss werden die praktischen Ergebnisse besprochen und Möglichkeiten für zukünftige Experimente genannt.

Olga Nesterova

Title of Thesis

Transactions and Resilience Strategies in Microservices Architectures

Keywords

Microservices, Transactions, ACID, Resilience, Distributed Systems, Eventual Consistency

Abstract

Implementing transactions in microservice architectures could be viewed as a non-trivial task. The traditional ACID concept is called into question because services act independently, manage their data, and communicate mainly over networks.

This thesis addresses the problem that transactions in microservice architectures can be viewed from several angles. Each individual microservice is capable of managing its own local transactions; however, cross-service transactions are occasionally required. Implementing such transactions challenges the classical ACID guarantees.

To maintain system stability, we must consider potential issues caused by network disruptions and system failures. We therefore investigate how to build stable, scalable, and resilient systems that can execute cross-service transactions.

To tackle these challenges, we first survey existing strategies and patterns. Next, we design and implement a prototype that demonstrates how appropriate patterns can deliver coordination and fault tolerance in a real microservice architecture. The prototype is then evaluated. The results indicate that certain patterns enable reliable workflows across services. Rather than blindly pursuing strict consistency, it is advisable to weaken consistency guarantees when necessary and rely on the system to become eventually consistent.

Finally, we discuss the practical outcomes and propose ideas for future experiments.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

2PC 2-Phase-Commit.

ACID Atomicity, Consistency, Isolation, Durability.

BASE Basically Available, Soft state, Eventually consistent.

C2C Customer to customer.

CI/CD Continuous Integration / Continuous Deployment.

DBMS Database Management System.

DDD Domain-Driven Design.

DevOps Development and Operations.

DLT Dead-Letter-Topic.

RPS Requests Per Second.

SOA Service-Oriented Archiecture.

Listings

1 Einleitung

1.1 Motivation und Hintergrund

In Microservices-Architekturen stellt die Umsetzung von Transaktionen eine Herausforderung dar. Insbesondere wird das klassische Konzept der ACID-Eigenschaften in verteilten Systemen infrage gestellt, da einzelne Services unabhängig voneinander laufen, eigene Daten separat verwalten und primär über die Netzwerke miteinander kommunizieren. Die vorliegende Arbeit befasst sich mit dem Problem, dass Transaktionen in Microservices-Architekturen von unterschiedlichen Seiten betrachtet werden können.

Einerseits verwaltet jeder Microservice seine eigenen lokalen Transaktionen. Andererseits ist es auch manchmal erforderlich, serviceübergreifende Transaktionen durchzuführen. Und bei der Umsetzung solcher Transaktionen werden die klassischen ACID-Eigenschaften in Frage gestellt. Darüber hinaus sind potenzielle Probleme mit Netzwerkstörungen und Systemausfällen zu berücksichtigen, um die Systemstabilität zu gewährleisten. In dieser Arbeit haben wir uns mit Mechanismen auseinandergesetzt, die es uns erlauben, stabile, skalierbare und resiliente Systeme zu entwickeln, die auch in der Lage sein sollen, serviceübergreifende Transaktionen durchzuführen.

Um alle diese Probleme zu adressieren, wird zunächst eine Recherche bestehender Strategien und Muster durchgeführt. Anschließend wird ein Prototyp entwickelt und implementiert, um zu demonstrieren, wie durch den Einsatz geeigneter Muster Koordination und Fehlertoleranz in einer realen Microservice-Architektur erreicht werden können. Die Ergebnisse zeigen, dass es bestimmte Muster gibt, die eine Implementierung serviceübergreifender robuster Abläufe ermöglichen. Es wäre aber notwendig, von starker Konsistenz Abstand zu nehmen und unsere Konsistenzgarantien gegebenenfalls abzuschwächen. Letztlich verlassen wir uns darauf, dass unser System zu einem späteren Zeitpunkt konsistent sein wird.

1.2 Ziel der Arbeit

In der vorliegenden Arbeit wird der Frage nachgegangen, auf welche Art und Weise Transaktionen in einer Mikroservice-Architektur optimal implementiert werden können. Es muss sichergestellt werden, dass die Datenkonsistenz gewahrt bleibt. Der Fokus liegt dabei auf der Umsetzung einer zuverlässigen und funktionierenden Musterlösung. Im Idealfall soll dabei das fehlerbehaftete Verhalten eines Microservices während des Ablaufs der Transaktion keine negativen Auswirkungen haben. Das System soll dabei sowohl vor der Ausführung der Transaktion als auch danach in einem konsistenten Zustand bleiben. Es wird die Anforderung formuliert, dass die Microservices resilient gegen Ausfälle sein sollen. Die zentrale Fragestellung lautet daher, auf welche Art und Weise dies erreicht werden kann.

1.3 Umfang und Einschränkungen der Arbeit

Im Fokus dieser Arbeit stehen Transaktionen, deren systemübergreifende Implementierung und ausgewählte Resilienzstrategien. Dabei bleiben Kosten-, Effizienz- oder Security-Aspekte explizit ausgeklammert. Die Erkenntnisse des praktischen Anteils gelten primär für Systeme mit vergleichbarem Technologie-Stack und erheben keinen Anspruch auf allgemeine Gültigkeit.

1.4 Struktur der Arbeit

Zunächst werden im theoretischen Teil die Aspekte der Transaktionen behandelt, die u. a. dazu führen, dass das klassische ACID-Modell in Microservices an seine Grenzen stößt. Im weiteren Verlauf werden alternative Konsistenzmodelle, Koordinationsmechanismen und relevante Aspekte der Resilienz präsentiert. Auf Basis dieser Erkenntnisse entwickeln wir einen lauffähigen Prototyp für ein Microservices-System, das asynchrone Kommunikation über Kafka sowie Koordination via Saga umfasst. Zusätzlich werden im System Resilienz-Mechanismen integriert. Im letzten Schritt werden die Ergebnisse ausgewertet. Auf dieser Grundlage werden konkrete Verbesserungsmöglichkeiten sowie Ansatzpunkte für weiterführende Arbeiten skizziert.

2 Grundlagen der Microservices-Architektur

2.1 Definition und Eigenschaften

Microservices sind ein relativ neuer Trend, der sich Mitte der 2010er Jahre im Zusammenhang mit agiler Entwicklung und Development and Operations (DevOps)-Praktiken verbreitete. Modulare Software wird aus einer Reihe stark entkoppelter Services aufgebaut, im Gegensatz zu herkömmlichen Methoden, bei denen ein einziger Monolith entsteht. Die Hauptidee besteht darin, dass wir eine gewisse Menge von losgekoppelten Services haben, die unabhängig voneinander laufen, separat deployt werden und ihre eigenen Domänenmodelle verwalten.

2.2 Architektur-Einsätze im Überblick

Eine der wichtigsten Aufgaben im modernen Software-Engineering besteht in der Auswahl der Architektur, die in einem definierten Umfeld effizient wäre. Das spielt eine kritische Rolle in der Entwicklung hochbelasteter, skalierbarer und flexibler Softwaresysteme. Die Systemarchitektur definiert nicht nur die Art und Weise der Komponenten-Zusammenarbeit. Sie legt auch fest, wie schnell die Software-Entwicklung gehen wird, sowie Wartbarkeit, Skalierbarkeit und die Möglichkeit der Integration von neuen Features oder Technologien in langfristiger Perspektive.

Zu den weitest verbreiteten Softwarearchitekturlösungen werden meistens Monolith- und Microservice-Architekturen zugeordnet. Die beiden Ansätze haben bestimmte Vor- und Nachteile, und die Auswahl hängt von vielen Faktoren ab, inklusive Businessanforderungen, Entwicklungskapazitäten und Teamgröße, vorgesehener Last und zukünftigen Wachstumsaussichten des Softwareprodukts.

2.2.1 Monolithische Architektur

Monolithische Architektur kann als traditioneller Weg des Software-System-Engineerings definiert werden. Dabei wird die Anwendung als eine einzelne Einheit entwickelt und deployt. Alle Systemkomponenten, dazu gehören Businesslogik, Datenbank, Benutzeroberfläche und Zusatzmodule, sind miteinander eng verbunden und funktionieren im selben Prozess. Dieser Ansatz war seit Jahrzehnten weit verbreitet und wird zurzeit auch oft in vielen Projekten verwendet, wo Datenintegrität und einfaches Systemdeployment besonders kritisch sind [22].

2.2.2 Vorteile der Monolithen

Zuerst möchten wir kurz darauf eingehen, welche Vorteile ein monolithischer Ansatz mit sich bringt:

- Einfache Entwicklung – am Anfang werden wenige Infrastrukturlösungen benötigt und Software kann schneller im Einsatz gebracht werden;
- Datenkonsistenz gesichert – da alle Teile der Software-Systeme im selben Environment interagieren, besteht nicht das Problem der verteilten Datenspeicherung;
- Einheitliches Deployment – das System wird als einzelner Service deployt, was CI/CD und Deployment wesentlich vereinfacht [22].

Aber mit zunehmender Komplexität und Benutzerwachstum der Anwendung wird ein Monolith langsam zu einer Einschränkung. Zu wesentlichen Nachteilen der monolithischen Architektur gehören:

- Komplexität der Changes – jedes Update benötigt ein Neu-Deployment des ganzen Systems, was die Entwicklung neuer Features verlangsamt und das Systemtesten erschwert;
- Eingeschränkte Skalierbarkeit – wenn die Last auf ein Teilsystem zunimmt (etwa Zahlungsabwicklungs-Module), muss das gesamte System skaliert werden;
- Code-Komplexitätssteigerung – Codebase wird umfangreicher, was zu Support-Problemen führt und die Weiterentwicklung des Systems verlangsamt[36].

Diese Einschränkungen treten besonders wesentlich in hochbelasteten Systemen wie Marktplätzen, Finanzdienstleistungsplattformen (z. B. Banking) oder Streaming-Services auf. In solchem Umfeld kann traditionelle Monolith-Architektur nicht immer das erwartete Niveau von Flexibilität und Skalierbarkeit gewährleisten, was einen Übergang zu Microservice-Architektur sinnvoller macht.

2.2.3 Microservice-Architektur

Microservice-Architektur bietet einen Alternativansatz des Software-System-Designs. Das basiert auf der Grundidee der Verteilung des Systems auf möglichst unabhängige Services. Jeder Service wäre dann für bestimmte Businesslogik verantwortlich, wird separat (autonom) entwickelt und verwendet einen eigenen Datensatz, soweit das nötig ist. Die Kommunikation mit anderen Services erfolgt via Netzwerk.

Die Philosophie der Microservice-Architektur kann gut mit dem „Teile und herrsche"-Prinzip beschrieben werden – das ermöglicht den Aufbau von flexiblen und gut skalierbaren Softwaresystemen, indem jedes Subsystem unabhängig von den anderen weiterentwickelt werden kann. Durch dieses Vorgehen werden System-Updates wesentlich vereinfacht und die Code-Basis wird effizient verteilt. Außerdem können so nur die Teilsysteme skaliert werden, die besonders kritisch sind und am meisten belastet werden [26].

Im Weiteren dieser Arbeit wollen wir nicht in eine detaillierte Betrachtung des Unterschieds zwischen Service-Oriented Archiecture (SOA) und Microservices eintauchen, aber es lohnt sich zu erwähnen, dass Microservices trotz bestimmter Ähnlichkeiten in gewissem Sinne einen etwa granulareren Spezialfall von SOA darstellen [23], [26].

2.2.4 Vorteile der Microservice-Architektur

Zu den Vorteilen dieses Ansatzes kann man Folgendes zählen:

- Einer der wichtigsten Vorteile der Microservice-Architektur ist die Möglichkeit, die Services unabhängig zu aktualisieren und deployen. Das löst eines der Hauptprobleme einer monolithischen Anwendung, wo jede einzelne Code-Änderung ein Re-Deployment des ganzen Systems erfordert [22];

- Technologische Flexibilität – verschiedene Technologien, Frameworks, Datenbanken und Programmiersprachen können im selben System für unterschiedliche Services benutzt werden;
- Horizontale Skalierbarkeit – mehr Kapazitäten können für die besonders belasteten Services separat gebucht werden;
- Fehlertoleranz – Ausfall eines Services bedeutet nicht den Ausfall des gesamten Systems. Andere Services bleiben im Einsatz [36], [26], [23].

Ein zentrales Ziel des Designs einer Microservices-Architektur besteht darin, Anwendungen leichter anpassen und warten zu können, indem sie das Deployment und die Weiterentwicklung einzelner Services unabhängig voneinander ermöglicht [15]. Für eine Microservice-Architektur werden jedoch eine wesentlich anspruchsvollere Infrastruktur/-Lösungen (und deren Pflegebedarf) benötigt, wie:

- Koordination der Services Zusammenarbeit – Es muss festgelegt werden, wie einzelne Services miteinander kommunizieren werden (REST, gRPC oder Event-Driven-Ansatz).
- Monitoring und Debugging – weil das System aus mehreren unabhängigen Services besteht, die über das Netzwerk kommunizieren, kann es schwierig sein, die Fehler zu erfassen und zu verfolgen. Dazu werden unterschiedliche Monitoring-/Logging-Tools eingesetzt, um die relevanten Informationen erst zu sammeln und danach zu analysieren/debuggen; [37].
- Sicherstellung der Datenkonsistenz – da jeder Service grundsätzlich mit einer eigenen Datenbank unabhängig von den anderen arbeiten kann, wird Inanspruchnahme der „Eventual Consistency“ vorangetrieben; [26];
- Man kann auch schnell den Überblick über einzelne Services und deren Versionen verlieren, wenn man keine klare Strategie für die Dokumentation verfolgt, oder man kann durch eine gemeinsame Oberfläche ungewünschte Abhängigkeiten reinbauen [29].

Wenn es aber um skalierbare und dynamisch entwickelte Software geht – sehen wir die Microservice-Architektur trotz dieser Challenges als eine gute Lösung. Als Beispiel kann man Marktplätze betrachten, bei denen eine Vielzahl von Benutzerinteraktionen und Anfragen verarbeitet werden müssen und sowohl Änderungsflexibilität als auch Fehlertoleranz gewährleistet sein müssen.

2.2.5 Grundprinzipien der Microservice-Architektur

Wie bereits dargestellt wurde, weisen herkömmliche Monolith-Architekturen gewisse Einschränkungen auf, die die Skalierbarkeit und das Deployment beeinträchtigen und die Entwicklungsflexibilität einschränken. Bei solchen Systemen, die normalerweise im Retail, Banking oder in der Logistik zum Einsatz kommen, sind hohe Belastbarkeit, Fehlertoleranz und die Fähigkeit zur dynamischen Skalierung von entscheidender Bedeutung. Aus diesem Grund erscheint ein monolithischer Ansatz in diesem Kontext nicht so ganz optimal.

Die Microservice-Architektur zeichnet sich durch eine Reihe von Eigenschaften aus, die eine effektive Verwaltung komplexer verteilter Systeme ermöglichen, Abhängigkeiten zwischen Komponenten reduzieren und eine hohe Systemverfügbarkeit gewährleisten. Es geht genauer um folgende Merkmale von Microservices:

- Unabhängiges Deployment;
- Lose Kopplung;
- Einzelverantwortung [23].

Aus diesen Prinzipien bildet man die Grundlage einer Microservice-Architektur. So werden Flexibilität, Fehlertoleranz und Skalierbarkeit des Systems gewährleistet. Im Weiteren werden diese Prinzipien genauer betrachtet.

Unabhängiges Deployment

Einzelne Services im Microservice-Einsatz betrachten wir als voneinander separierte Prozesse. Jeder davon kann dann ohne Einfluss auf andere Teile des Systems deployt werden. Welche Vorteile bringt uns das? Folgendes lässt sich definieren:

- Die Möglichkeit separater Updates, die das Fehlerrisiko reduzieren und die Entwicklung beschleunigen
- Jeder Service kann aktualisiert werden, ohne dass das Gesamtsystem ausfällt.
- Die Änderungen kann man unabhängig von anderen Systemkomponenten einspielen und testen.

Aber die Realisierung des unabhängigen Deployments braucht eine speziell ausgestattete Infrastruktur (Docker, Kubernetes, automatisierte Continuous Integration / Continuous Deployment (CI/CD)-Pipeline). [36].

Es ist wichtig zu sagen, dass unabhängiges Deployment bei Systemen mit relativ hoher Auslastung besonders wichtig wäre - weil die Development-Teams dann parallel arbeiten können, ohne sich gegenseitig zu stören. [26]. Beispielsweise kann in einer Retail-Anwendung das Team, das für das Accountingsystem verantwortlich ist, neue Features unabhängig von dem Team umsetzen, das am Produktkatalog oder an der Werbungsintegration arbeitet.

Dieses Prinzip gewährleistet Flexibilität, Update-Geschwindigkeit und Ausfallsicherheit des Software-Systems.

Lose Kopplung

Die nächste wichtige Eigenschaft der Microservices ist ihre inhärent lose Kopplung. Wenn es um einen Monolithen geht, sind alle Systemteile eng miteinander gekoppelt. Das erschwert manchmal eine Änderung/Ersatz einzelner Teile der Anwendung [23]. Im System von Microservices entwickelt man jeden Service als unabhängige Komponente. Diese Komponente interagiert dann mit anderen durch strikt definierte Schnittstellen. Ein wichtiger Vorteil wäre auch, dass für die verschiedenen Aufgaben die Technologien optimal selektiert werden können – genau diejenigen, die speziell für definierte Aufgaben entwickelt wurden. Etwa Python für Datenanalyse, Java für Zahlungsabwicklung und Node.js für Schnittstellen [26].

Es kann die lose Kopplung zwischen Microservices u. A. auf die Art und Weise gewährleistet werden:

- asynchrone Kommunikationsmechanismen wie Kafka-Message-Queue oder Rabbit MQ anstatt der harten Schnittstellenanfragen
- möglichst muss jeder Microservice eine separate Datenbank/Datenbankschema besitzen [23].

Services müssen so designed werden, dass Änderungen an einem Service keine Änderungen an anderen Services erfordern [36]. Die Services können von Entwicklern separat aktualisiert werden, ohne andere Teilsysteme zu beeinflussen. Dabei bedeutet der Ausfall

eines Services nicht den des Gesamtsystems. Aber lose Kopplung kommt auch nicht ohne Kosten, da Datenkonsistenz und Constraints zu bewahren über mehrere Services hinweg nicht so trivial sind [17]. Aber genauer auf dieses Problem werden wir im nächsten Kapitel eingehen.

Prinzip der Einzelverantwortung

Das Prinzip der Einzelverantwortung bedeutet, dass jeder Microservice für eine einzelne Business-Funktion verantwortlich ist. Das hilft, das System effizient, transparent und flexibel zu gestalten [26].

Die Service-Grenzen und Service-Aufgaben müssen klar konzipiert werden. Um das zu erreichen, kann man z. B. Domain-Driven Design (DDD) dafür einsetzen, um den passenden Services-Schnitt und Boundaries zu definieren, um hohe Kohäsion und lose Kopplung zu erreichen [23], [36]. In Monolithen kann es vorkommen, dass ein Modul für mehrere Aufgaben verantwortlich ist. Dies kann unter Umständen die Wartbarkeit und Skalierbarkeit des Systems beeinträchtigen. Die Microservice-Architektur bietet in diesem Zusammenhang einen entscheidenden Vorteil, so dass jeder Service für eine bestimmte Aufgabe verantwortlich ist. Die Services müssen maximal unabhängig voneinander designed werden, um sicherzustellen, dass die Änderungen in einem Service die Funktionalität des Gesamtsystems nicht beeinträchtigen.

2.3 Synchrone vs. asynchrone Kommunikation

In Microservices-Architekturen ist eine effektive Kommunikation zwischen den einzelnen Services entscheidend für den Aufbau skalierbarer und widerstandsfähiger Systeme. Im Weiteren möchten wir diese Ansätze genauer anschauen, sowie ihre Vorteile, Nachteile und Anwendungsgebiete.

Generell unterscheidet man zwischen einem synchronen und einem asynchronen Kommunikationsansatz [23] [26], wobei die Auswahl von einem oder anderem Ansatz systemspezifische Anforderungen an Latenz, Fehlertoleranz oder Kopplungseinschränkungen beeinflussen kann [26].

Bei der synchronen Kommunikation handelt es sich um blockierende Aufrufe zwischen Services. Der Client-Dienst sendet Request und wartet auf Response, bevor er die weiter

aufnehmen kann. Dieser Ansatz wird üblicherweise mithilfe von folgenden Technologien umgesetzt:

- REST über HTTP: am weitesten verbreiteter Ansatz aufgrund seiner Einfachheit und Kompatibilität mit Webstandards. Allerdings entsteht eine zeitliche Kopplung zwischen Services und das kann zu „cascading issues“ führen. Dabei gilt: Je mehr Calls hintereinander zu einer Aufrufkette hinzugefügt werden, desto gefährlicher sind die Konsequenzen bei Fehlern [23];
- gRPC: ist ein High-Performance-Framework, das HTTP/2 und Protocol Buffers zur effizienten Datenserialisierung verwendet. gRPC unterstützt Streaming und ist leistungsfähiger als REST, erfordert jedoch zusätzliche Tools [26].

Während die synchrone Kommunikation einfache direkte Interaktionen ermöglicht, kann sie die Systemstabilität beeinträchtigen, da sich Fehler in abhängigen Services ausbreiten und so Ausfallzeiten oder Leistungseinbußen verursachen können.

Asynchrone Kommunikation ermöglicht es Services, Nachrichten auszutauschen, ohne den Absender zu blockieren. Dieser Ansatz gewährleistet nicht nur lose Kopplung, sondern ermöglicht es auch, unabhängig zu arbeiten. Dieser Ansatz wird üblicherweise mithilfe von folgenden Technologien umgesetzt:

- Message-Brokers (Kafka, RabbitMQ): Diese erleichtern ereignisgesteuerte Kommunikation, indem sie es Services ermöglichen, Nachrichten auszutauschen. Dieser Mechanismus verbessert die Systembelastbarkeit und Skalierbarkeit, da Services nicht auf Antworten warten müssen [23].
- Ereignisgesteuerte Architekturen (engl.: Event-driven architecture): Services veröffentlichen Events, die von anderen Interessierten (Abonnenten) abgearbeitet werden können. Dieses Muster reduziert direkte Abhängigkeiten und verbessert die Reaktionsfähigkeit des Systems. Eine sorgfältige Verwaltung des Event-Schemas ist dabei erforderlich, um Inkonsistenzen und Inkompatibilitätsprobleme zu vermeiden [26].

Die asynchrone Kommunikation zwischen einzelnen unabhängigen Komponenten in einem System kann die Skalierbarkeit und Fehlertoleranz erheblich verbessern. Allerdings sind bei diesem Kommunikationsansatz besondere Herausforderungen im Hinblick auf Fehlerbehebung, die Einhaltung der Reihenfolge von Events und die Datenkonsistenz zu beachten. Die Entscheidung, ob man den anderen Ansatz benutzt, soll auf dem Grund von systemspezifischen Anforderungen getroffen werden:

- Wenn es für eine spezifische Art von Interaktionen entscheidend ist, dass diese in Echtzeit stattfinden, wie etwa bei Authentifizierung oder beim Datenabruf, kann man einen synchronen Ansatz verwenden.
- Wenn aber Verfügbarkeit und lose Kopplung im Vordergrund stehen, kann man sich für einen asynchronen Ansatz entscheiden.

Dabei wäre es auch durchaus denkbar, diese unterschiedlichen Kommunikationsansätze zu kombinieren, sowohl auf der Ebene des Gesamtsystems als auch auf der der einzelnen Services [23]. Letztendlich, wenn es dazu kommt, ein Kommunikationsprotokoll oder eine Technologie auszuwählen, soll man spezielle Systemanforderungen oder sogar konkrete Anwendungsfälle betrachten [23].

Zusammenfassung

Das war eine einigermaßen kurze Einführung in die Problematik. Dennoch ist es für den Zweck dieser Arbeit zunächst ausreichend, das Konzept von Microservices zu verstehen und ihre Unterschiede zu Monolithen zu erkennen. Wir haben festgestellt, dass Microservices durch solche Merkmale wie Modularisierung, die sich in unabhängigen Units für Deployment und Entwicklung manifestiert, sowie durch die Kommunikation über Netzwerke geprägt sind.

Die Microservices sind auch dadurch charakterisiert, dass den Entwicklern auf der Ebene der einzelnen Services ein erheblicher Handlungsspielraum zur Verfügung steht. Dies gilt sowohl für die Auswahl der Technologien als auch für die Festlegung der fachlichen Architektur. Die nach außen sichtbaren Schnittstellen müssen aber klar definiert werden, da über diese die Integration der einzelnen Microservices erfolgt.

Es bleibt aber tatsächlich nicht so viel übrig, was auf systemübergreifender Ebene vereinheitlicht werden soll. Zu diesen Punkten gehören: Definition von Schnittstellen, Logging, Tracing, Monitoring, Konfigurationen (falls eine zentrale Konfiguration gewünscht wird), Service Discovery und Security. Die Microservice-Architektur weist eine zunehmende Komplexität auf, die durch die Integration einer Vielzahl technischer Aspekte in die Gesamtarchitektur bedingt ist.

Microservices insgesamt bieten zwar viele attraktive Eigenschaften. Sie bringen aber auch erhebliche Herausforderungen mit sich, was wir in den nächsten Kapiteln noch sehen werden, insbesondere im Hinblick auf das Transaktionsmanagement und die Datenkonsistenz

im Gesamtsystem. Das Wichtigste, was man aber mitnehmen soll, ist, dass dieser Ansatz nicht besser oder schlechter ist als der monolithische, es ist nur halt anders.

3 Umsetzung von Transaktionen in Microservices

Wir wollen Transaktionen sowohl in einem einfachen, also nicht verteilten Kontext, um ihre grundlegenden Abläufe besser zu verstehen, als auch in einem verteilten Kontext betrachten. Anschließend werden wir unterschiedliche Vorgehensweisen bei der Implementierung von Transaktionen und deren Herausforderungen in dem verteilten Kontext analysieren.

3.1 Transaktionen und verteilte Systeme

3.1.1 ACID Transaktionen

Wir können eine Transaktion als einen Mechanismus definieren, der es einer Anwendung ermöglicht, mehrere Lese- und Schreiboperationen zu einer einzigen, atomaren Einheit zusammenzufassen. Dadurch wird sichergestellt, dass entweder alle Operationen erfolgreich sind (Commit) oder keine (Abbruch und Rollback) [14]. Dieser Ansatz vereinfacht die Fehlerbehandlung, indem er Teilfehler verhindert und es der Anwendung ermöglicht, die gesamte Transaktion bei einem Fehlschlag sicher erneut zu versuchen [15].

Einerseits helfen uns Transaktionen und erleichtern unser Leben, in dem Sinne, dass sie uns bestimmte Garantien gewährleisten, sodass wir uns um einige Fehlerszenarien gar nicht kümmern müssen, da Database Management System (DBMS) uns die fertigen Lösungen bereitstellt [15]. E. Brewer hat es in seinem Paper sehr schön ausgedrückt, dass „people love the ACID properties and hesitant to give them up“ [3].

Man kann eine Atomicity, Consistency, Isolation, Durability (ACID)-Transaktion als eine Abstraktion verstehen, die uns glauben lässt, die Welt sei perfekt – ohne Abstürze

(Atomicität), ohne Nebenläufigkeit (Isolation) und mit absolut verlässlichem Speicher (Dauerhaftigkeit) [15].

Atomicität in dem Sinne bedeutet, dass die Transaktion nur als Ganzes ausgeführt wird. Konsistenz soll weiter gewährleisten, dass Daten vor und nach der Transaktion sich in einem konsistenten Zustand befinden (d. h., alle Integritätseinschränkungen bleiben intakt). Und auf Isolation werden wir etwa genauer im Weiteren eingehen.

Isolation zielt darauf ab, unkontrollierte und unerwünschte Ergebnisse paralleler Zusammenarbeit an Objekten von mehreren Benutzern zu verhindern. Dies soll ermöglichen, dass im Fall eines Fehlers eine Transaktion mit minimalem Seiteneffekt zurückgerollt und das System in einem so konsistenten Zustand belassen werden kann, als hätte die Transaktion nie stattgefunden [14].

Unter Objekten verstehen wir dabei Ressourcen, die wir in der Datenbank manipulieren wollen. Dies können beispielsweise Zeilen in relationalen Datenbanken oder auch eine Menge an Zeilen sein, die von einem bestimmten Suchprädikat erfasst werden kann. Dabei sind die oben genannten unkontrollierten und ungewünschten Ergebnisse in der Literatur unter den Begriffen Parallelitätsprobleme, Parallelitätsbugs [15] oder auch Anomalien [2] oder auch Phänomene [2] bekannt. In dieser Arbeit werden wir für diese Zwecke entweder das Wort Anomalie oder das Wort Phänomen verwenden.

Es gibt tatsächlich viele Isolationsstufen, die unterschiedliche Grade an Isolation bieten (die schwächste dabei ist Read Uncommitted und die stärkste ist Serializable).

Es gibt auch verschiedene Gründe, warum man z. B. solche Phänomene wie Dirty Reads/-Dirty Writes verhindern will, und setzt mindestens Read Committed ein: Zum einen können parallele Transaktionen basierend auf den sichtbaren partiellen, aber bisher nicht committeten Ergebnissen von einer anderen Transaktion (was bei Read Uncommitted durchaus möglich ist) irgendwelche Entscheidungen treffen, und wenn z. B. die letztere Transaktion zurückrollt, entsteht logischerweise die Frage: Was soll man machen mit den Transaktionen, die schon Entscheidungen basierend auf diesen Daten getroffen haben und sogar schon committet haben [15]?

Um solche Konsistenzprobleme zu verhindern, soll man mindestens die Read Committed Isolationsstufe in Betracht ziehen. Tatsächlich gibt es eine Reihe von weiteren Anomalien, die weit über Read Committed hinausgehen, und um diese zu verhindern, müssen wir tatsächlich etwa stärkere Isolationsstufen in Betracht ziehen. Zu den weiteren Phänomenen zählt man u. A.:

- Lost Update tritt auf, wenn zwei Transaktionen dieselbe Zeile lesen, um sie zu aktualisieren, aber das erste Update vom zweiten Update überschrieben wird.
- Non-Repeatable Read liegt dann vor, wenn eine Zeile innerhalb einer Transaktion zweimal abgefragt wird und sich einige Attributwerte zwischen den beiden Abfragen unterscheiden.
- Phantom Read tritt dann auf, wenn eine Transaktion eine Suchabfrage macht, die ein bestimmtes Suchprädikat erfüllt, aber in der Zwischenzeit eine andere Transaktion eine neue Zeile hinzufügt, die ebenfalls das Suchprädikat erfüllt. Somit erhält die erste Transaktion bei den weiteren Leseabfragen eine zusätzliche Phantom-Zeile.
- Read Skew ist dem Non-Repeatable Read sehr ähnlich, betrifft aber mehr als eine Tabelle, so dass wir Inkonsistenzen beim Lesen mehrerer Tabellen beobachten können.
- Write Skew kann auftreten, wenn zwei Transaktionen die Beschränkung auf Domänenebene verletzen, d. h. in einer Situation, in der wir den Gesamtsaldo positiv halten müssen. Zwei Transaktionen aktualisieren unterschiedliche Zeilen, aber im Endeffekt, nachdem beide committet wurden, wird eine logische Integritätseinschränkung verletzt.

Alle diese Phänomene kann man beseitigen, indem die passende Isolationsstufe konfiguriert wird. Aber je höher die Isolationsstufe, desto weniger Spielraum bleibt übrig für die Nebenläufigkeit. Alternativ kann man auch überlegen, solche Anomalien wie Lost Update mithilfe von Sperren oder atomaren Operationen zu verhindern [15].

Die Abbildung 3.1 kann einen Gesamtüberblick über bekannte Anomalien und unterschiedliche Isolationsstufen verschaffen, die darauf gerichtet sind, jene andere Anomalie zu verhindern.

Jedoch können wir diese Garantien nicht einfach als selbstverständlich betrachten, wenn wir dem Bereich verteilter Systeme beitreten. Aus diesem Grund sollten wir alternative Strategien in Betracht ziehen, um Transaktionen zu handhaben.

3.1.2 Transaktionen ohne ACID

Alle diese ACID-Garantien funktionieren perfekt im Kontext „ein System – eine Datenbank“.

Isolation Level	P0 (Dirty Write)	P1 (Dirty Read)	P4C (Cursor Lost Update)	P4 (Lost Update)	P2 (Fuzzy Read)	P3 (Phantom)	A5A (R.Skew)	A5B (W.Skew)
Read Uncommitted	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
Read Committed	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
Repeatable Read	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Abbildung 3.1: Isolationsstufen nach [2]. Quelle: [6]

Wenn wir aber über verteilte Systeme reden, stoßen wir da auf die Grenzen, weil unsere Systeme physisch und oft geographisch verteilt sind. Wir müssen uns darüber im Klaren sein, dass alles aber nicht ohne Kosten kommt, und wenn wir feststellen, dass unser System die Transaktionen doch tatsächlich braucht, sollen wir diese Anforderung gegen die Kosten, die damit verbunden sind, abwägen [15].

Einerseits können wir uns immer für eine einfache Lösung entscheiden, sodass alle Services im System die gemeinsame Datenbank teilen. Services können einfach direkt auf alle nötigen Daten zugreifen und alle notwendigen Operationen durchführen, ohne dabei miteinander kommunizieren zu müssen. So bietet uns das „Shared Database“-Pattern gewünschte ACID Eigenschaften quasi „out of the box“.

Aber man betrachtet Shared Database-Pattern eher als Anti-Pattern. Denn es führt zum einen zu hoher Kopplung, so dass strukturelle Änderungen am Dataschema die Arbeit der einzelnen Microservices verhindern können [32]. Und zum anderen besteht die Gefahr, dass mehrere Microservices im Endeffekt um gemeinsame Ressourcen kämpfen müssen [23]. Ferner fügen wir dadurch einen gewissen Grad an Kopplung (sog. „content coupling“ [23]) hinzu, und das alles steht im Gegensatz dazu, was wir mit der Microservice-Architektur erreichen wollen.

Deshalb gewinnen an Bedeutung weitere Ansätze, die uns ermöglichen, Daten separat zu halten. Und zwar gibt es folgende Ansätze für die Verwendung von Datenbanken in Microservice-Architekturen:

- private Tabellen pro Microservice;
- Schema pro Microservice;
- Datenbankserver pro Microservice [19].

In einer Studie wird dabei festgestellt, dass von den meisten Entwicklern die Idee hervorgehoben wird, den Zustand eines Microservices in seinem eigenen verwalteten Datenbankserver zu kapseln. Dafür werden solche Gründe angegeben, wie lose Kopplung, unabhängige Skalierbarkeit der Datenbank und Fehlertoleranz [17]. Genau diesen Ansatz wollen wir auch später in unserem praktischen Anteil dieser Arbeit verwenden.

3.1.3 ACID vs BASE

Mit der Skalierung unseres Systems wird die Beibehaltung von Konsistenz und Verfügbarkeit zu einer kritischen Herausforderung [3]. ACID gewährleistet starke Konsistenz („strong consistency“), geht jedoch auf Kosten einer geringeren Verfügbarkeit und Skalierbarkeit. 2-Phase-Commit (2PC) ist ein gängiger Ansatz dafür, hat jedoch erhebliche Auswirkungen auf die Systemverfügbarkeit und Sperrung von Ressourcen. Im Gegensatz dazu priorisiert Basically Available, Soft state, Eventually consistent (BASE) Verfügbarkeit und Skalierbarkeit und akzeptiert temporäre Inkonsistenzen, die im Laufe der Zeit durch „eventual consistency“-Mechanismen behoben werden [24].

Interessanterweise stellte Eric Brewer in einem seiner Vorträge die gängige Meinung in Frage, dass Finanzinstitute sich streng an ACID-Eigenschaften halten müssen. Er argumentierte, dass Finanzsysteme in der Vergangenheit auch ohne perfekte Konsistenz erfolgreich funktioniert haben und Verfügbarkeit tatsächlich stärker mit dem Umsatz korreliert als Konsistenz. Dies deutet darauf hin, dass selbst in kritischen Sektoren wie dem Bankwesen Spielraum für BASE-Eigenschaften gegeben ist, wenn Systeme entworfen werden, die Verfügbarkeit gegenüber strikter Konsistenz priorisieren [4]. Letztendlich hängt die Wahl zwischen ACID und BASE jedoch stark von den spezifischen Anforderungen der zu entwickelnden Anwendung ab.

3.2 Verteilte Ablaufsteuerung

Wie wir bereits gesehen haben, strukturieren Microservice-Architekturen unsere Anwendung als eine Menge von Services. Und manchmal müssen diese Services zusammenarbeiten, um eine Anfrage zu bearbeiten bzw. eine Aufgabe zu erledigen. In so einer Konstellation sprechen wir über Geschäftstransaktionen, die über die Grenzen der einzelnen Services hinausgehen. In diesem Zusammenhang wollen wir die folgenden Koordinationsmechanismen genauer betrachten: 2PC und Saga.

3.2.1 2-Phase-Commit

2PC ist eine gängige Lösung, um verteilte Transaktionen zu implementieren. Im Prinzip handelt es sich um einen Konsensalgorithmus (den Spezialfall davon – atomarer Commit), den wir in einer verteilten Umgebung laufen lassen wollen. Dies ermöglicht es, dass alle beteiligten Knoten sich auf ein bestimmtes Ergebnis (in unserem Fall wäre es die Entscheidung commit oder abort Transaktion) einigen können [15]. Dabei wird versucht, so nahe wie möglich an ACID-Garantien zu halten.

Wie man dem Namen entnehmen kann, besteht das Protokoll aus 2 Phasen, nämlich „prepare“ und „commit“. Während der „prepare“-Phase fragt der Koordinator alle beteiligten Teilnehmer an der Transaktion, ob diese bereit sind, einen Commit durchzuführen. Und nur in einer zweiten Phase wird tatsächlich ein Commit durchgeführt, es sei denn, mindestens einer von den Beteiligten hat sich gegen den Commit geäußert oder hat nichts geantwortet [30].

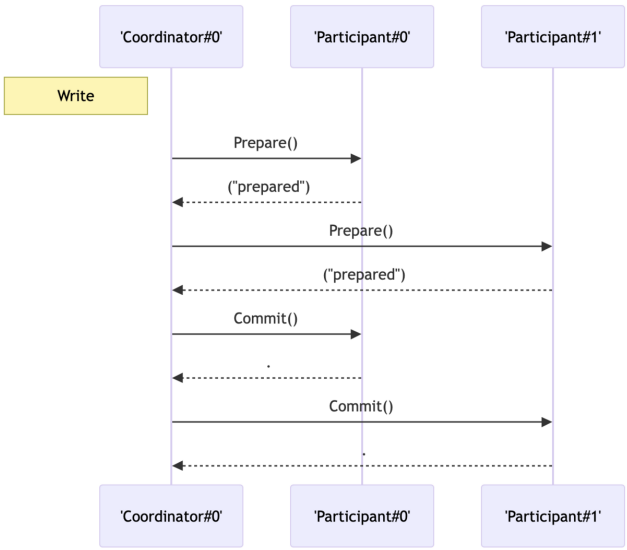


Abbildung 3.2: Zwei-Phasen-Commit-Protokoll. Quelle: Wikipedia

Wenn ein Teilnehmer sich für ein Commit entschieden hat und die entsprechende Nachricht an den Koordinator gesendet wurde, kann er seine Entscheidung während der Wartezeit nicht mehr ändern. Ab diesem Zeitpunkt ist er an seine Entscheidung gebunden und muss auf weitere Anweisungen des Koordinators für die zweite Phase warten. Der Koordinator wird zum „single point of failure“. Bei einem Ausfall des Koordinators müssen die Teilnehmer immer warten, bis dieser seine Arbeit wieder aufnehmen kann. Darüber hinaus müssen die Teilnehmer ihre Ressourcen sperren, bis diese eine Commit- oder Abort-Nachricht vom Koordinator erhalten (deshalb wird es auch „blocking commit protocol“ in der Literatur genannt [30]).

Es wird aber davon abgeraten, 2PC zu verwenden. Denn zum einen wird für ein Erfolgsszenario vorausgesetzt, dass alle beteiligten Parteien (Koordinator und Teilnehmer) verfügbar sind. Zum anderen wird dabei sehr stark auf synchrone blockierende Kommunikation gelegt [26]. Wir streben aber mit Microservices eher nach einer lose Kopplung von einzelnen Services und schätzen auch die Möglichkeit, asynchron zu kommunizieren. Deshalb wollen wir im Weiteren den anderen Ansatz genauer betrachten, und zwar die Implementierung von Geschäftsabläufen mit Saga-Muster [23] [35].

3.2.2 Sagas

Das Saga-Muster nutzt man, um sog. lange laufende Transaktionen zu koordinieren. Es wurde bereits 1987 eingeführt, und damals drehte sich die Idee um das Problem sogenannter langandauernder Transaktionen, die Ressourcen blockieren und dadurch schnellere Abläufe behindern können [12].

Kern dieses Ansatzes besteht darin, eine Kette von lokalen ACID-Transaktionen (Transaktionen, die Atomarität, Konsistenz, Isolation und Dauerhaftigkeit garantieren) zu bilden. Diese einzelnen lokalen Transaktionen werden unabhängig voneinander abgearbeitet, aber zusammen erfüllen sie die Anforderungen einer langen Transaktion, die über mehrere Services hinweg ausgeführt wird [23].

Die ursprüngliche Idee von Saga wurde später durch die Idee ergänzt, solche Transaktionen in einer Multi-Datenbankumgebung auszuführen. Dabei lohnt es sich, das Zusammenspiel von globalen und lokalen (sog. Subtransaktionen) Transaktionen zu betrachten, was auch im Kontext der Benutzung von mehreren DBMS von Bedeutung ist. So eine globale Transaktion beinhaltet normalerweise eine Reihe von autonomen Subtransaktionen [28].

Es gibt verschiedene Arten von Subtransaktionen: kompensierbare, wiederholbare und Pivot-Subtransaktionen.

- Kompensierbare Subtransaktionen sind Transaktionen, die nach dem Commit mithilfe einer kompensierenden Transaktion rückgängig gemacht werden können (sie sind rückwärts wiederherstellbar);
- Wiederholbare Subtransaktionen werden nach einer ausreichenden Anzahl von Übermittlungen garantiert festgeschrieben (sie sind vorwärts wiederherstellbar);
- Pivot-Subtransaktionen sind Transaktionen, die weder kompensierbar noch wiederholbar sind [28].

Das Resultat der globalen Transaktion hängt vom Ergebnis der Pivot-Subtransaktion ab. Bei einem Fehler in der Pivot-Subtransaktion wird die Kompensierungskette aller vorherigen Schritte angestoßen. Bei einer erfolgreichen Pivot-Transaktion werden unsere wiederholbaren Subtransaktionen bis zum Commit versucht [28] [1]. Man kann sich das so vorstellen, dass in einem Bestellprozess, nachdem die Bezahlung erfolgreich war, sich im

letzten Schritt eine Subtransaction darum kümmern muss, dass der Kunde eine entsprechende Nachricht bekommen soll. Diese letzte Transaktion kann beliebig oft wiederholt werden. Der Erfolg der gesamten Transaktion wird dadurch nicht beeinträchtigt.

Mit Saga (so wie es im orig. Paper [12] dargestellt wurde) wollen wir haben, dass es für jede sog. *T-Transaktion* eine entsprechende kompensierende *C-Transaktion* gibt. Die Letzteren sollen uns dabei helfen, den Zustand des Systems „semantisch“ wiederherzustellen. Es geht aber nicht darum, einen Rollback im Sinne von ACID zu machen, da sich der Datenbestand zwischen der ursprünglichen *T-Transaktion* und der späteren *C-Transaktion* bereits durch andere Prozesse verändert haben könnte[12].

Die grundlegende Idee des Saga-Musters in einer Mikroservice-Architektur besteht darin, umfassende Geschäftsprozesse, die über mehrere Services hinweg ausgeführt werden, so zu organisieren, dass sie als aufeinanderfolgende, voneinander unabhängige Transaktionen modelliert und von jedem beteiligten Service eigenständig ausgeführt werden können [20]. Im Prinzip bedeutet das, dass jede Saga aus einer Reihe von lokalen ACID-Transaktionen besteht. Im Erfolgsszenario werden alle ausgeführt und am Ende wird die Saga abgeschlossen. Im Fehlerszenario soll eine Reihe von kompensierenden Transaktionen die zuvor gemachten Änderungen ausgleichen (sog. „semantic rollback“ [23]).

Dabei sollen wir unterscheiden zwischen fachlichen und technischen Fehlern [8]. Mit Saga wollen wir die ersten adressieren. Dabei hat Saga damit gar nichts zu tun, wenn während des Aufrufs einer Methode in einem Service eine OptimisticLockingFailureException geworfen wird. Solche Fehler sollen nicht den gesamten Workflow rückabwickeln.

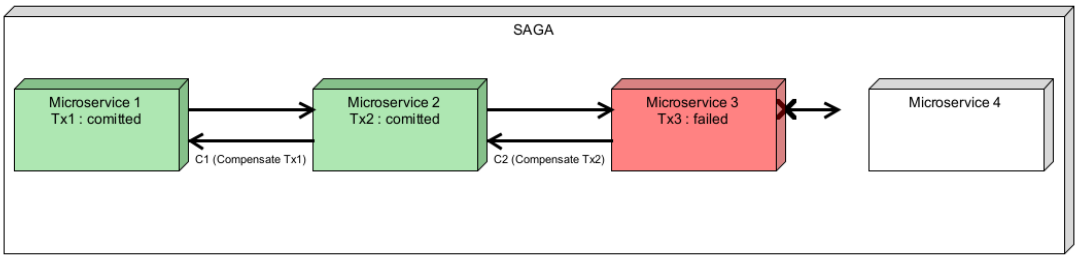


Abbildung 3.3: Allgemeine Abbildung von Saga

Wenn wir Saga im Kontext von ACID-Garantien betrachten, müssen wir tatsächlich feststellen, dass die Implementierung von Saga uns allerhöchstens nur ACD davon gewährleisten kann, d. h., es ist generell möglich, partielle Ergebnisse von einer bisher nicht abgeschlossenen Transaktion zu sehen [26] [31].

Eine mögliche Anomalie (Lost Update) wäre denkbar, wenn z. B. eine Saga ein von einer anderen Saga durchgeführtes Update überschreibt. Eine weitere Anomalie (Dirty Read) kann auftreten, wenn eine Saga die Daten liest, die gerade von einer anderen, bisher nicht abgeschlossenen Saga aktualisiert werden. Es gibt leider keine silberne Kugel, um diese Anomalie zu verhindern, wie es in einem nicht verteilten Kontext innerhalb einer Datenbank mit einer entsprechenden Isolationsstufe möglich wäre. Es liegt in der Verantwortung des Entwicklers, Sagas-Schritte so zu modellieren und zu implementieren, dass Anomalien entweder vermieden oder ihre Auswirkungen minimiert werden. Eine Möglichkeit wäre, semantische Sperrung zu verwenden, die auf Application-Level gesetzt werden soll [7]. Dabei wird es nicht von einem DBMS verwaltet, sondern es wird Teil einer Implementierungslogik.

Durchsetzung von „globalen“ Integritätseinschränkungen wird auf Application-Level gelagert, damit wird nicht DBMS dafür zuständig, von uns definierte Integritätseinschränkungen durchzusetzen und zu validieren, sondern Anwendungsentwickler selbst sollen sich darum in Code kümmern [24]. Die Fehler in der Validierung (oder deren Fehlen) auf Application-Level können dabei zu Inkonsistenzen der Daten im gesamten System führen [17]. Irgendwelche Workarounds dafür auszusuchen und zu implementieren, liegt in der Verantwortung der Anwendungsentwickler.

Es gibt zwei gängige Arten der Umsetzung, nämlich Choreografie und Orchestrierung – Saga. Dabei ist das Ziel des Patterns in beiden Ansätzen, eine Art von „koordiniertem Zusammenspiel“ zu schaffen, das die Gesamtaufgabe bewältigt, ohne dass einzelne Schritte dabei die beteiligten Services komplett blockieren. Koordinationslogik bei Orchestrierung-Saga bleibt beim zentralen Orchestrator, dabei kommunizieren einzelne Saga-Teilnehmer (egal ob synchron oder asynchron) nur mit dem Orchestrator; Bei Choreografie-Saga wird die Koordinationslogik zwischen einzelnen Services verteilt, die den jeweiligen Schritt der Saga ausführen sollen.

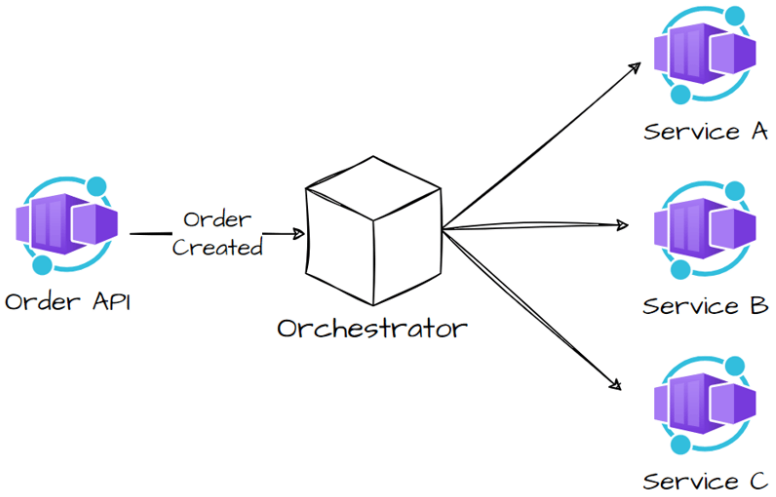


Abbildung 3.4: Orchestrierung-Saga (Bild aus dem Post im Blog The Saga Pattern, 29. Januar 2024).

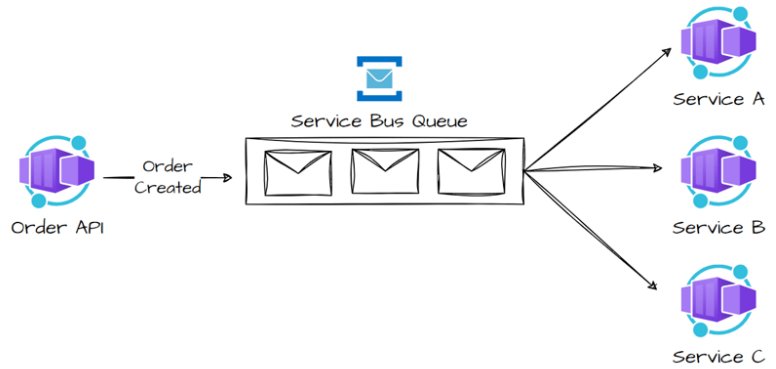


Abbildung 3.5: Choreografie-Saga (Bild aus dem Post im Blog The Saga Pattern, 29. Januar 2024).

Orchestrierung - Saga bietet mehr Einfachheit und Übersichtlichkeit, insbesondere bei komplexen Workflows. Dabei kann der Orchestrator Teil eines Services sein, der die Fachlogik beinhaltet, oder man kann ihn in ein separates Service auslagern. Außerdem ist es einfacher, die Sequenz von Schritten zu definieren und ihren Fortschritt von einem einzigen Kontrollpunkt aus zu überwachen. Wir sollten jedoch darauf achten, nicht zu viel Geschäftslogik, die eigentlich den Teilnehmern gehört, in den Orchestrator selbst zu über-

tragen. Als Faustregel gilt, dass man den Orchestrierungsansatz für komplexe Workflows anwenden soll [26].

Choreografie-Saga basiert auf der umfangreichen Nutzung asynchroner Kommunikation und Kommunikation über Events, was schwer zu erfassen und zu verstehen sein kann. Es ist schwieriger, unsere globalen „Business"-Transaktionen zu verfolgen und zu überwachen, aber auf der anderen Seite erreichen wir dadurch maximal lose Kopplung. Auch die Anzahl der Teams, die an Saga arbeiten, sollte in Betracht gezogen werden. Ist ein Team für eine Saga zuständig, kann es sinnvoll sein, einen stärker gekoppelten, orchestrierten Ansatz zu wählen. Falls Teams komplett isoliert voneinander arbeiten und diese Unabhängigkeit weiter behalten wollen, könnte eine Choreografie-Saga die sinnvollere Option sein [23].

Bei der Design- und Implementierung von Saga wäre es sinnvoll, die Schritte vorzuziehen, bei denen die Wahrscheinlichkeit eines Fehlers am höchsten ist. Wenn der Prozess früher scheitert, vermeiden wir es, später kompensierende Transaktionen durchzuführen (sog. „fail-fast" Prinzip).

Generell wird es dem Entwickler überlassen, welchen Ansatz für ein bestimmtes Problem er auswählt, und man hat relativ viel Spielraum bei der Implementierung. Zudem wäre es durchaus möglich, Choreografie- und Orchestrierungsansätze zu kombinieren und/oder zu verschachteln. Jedoch ist es wichtig, den Überblick darüber zu behalten, auf welcher Stage von Saga sich das System aktuell befindet; sonst kann man einfach die Kontrolle über die Geschäftstransaktion verlieren [23] [22].

Zuallerletzt können wir einen Schritt zurückmachen und noch einmal überlegen: Wenn unser System einen hohen Koordinationsaufwand erfordert und viel Kommunikation zwischen Services stattfindet, die die Ausführung von Transaktionen über mehrere Services hinweg erfordert, wäre es durchaus möglich, dass die Boundaries von Services (im Kontext von DDD) nicht so sauber definiert wurden [36]. In diesem Fall können Korrekturen der Architektur und Anpassungen der Servicegrenzen unnötigen Overhead beseitigen. Am Ende kann es sein, dass verteilte Transaktionen gar nicht erforderlich sind und die gesamte Logik und die notwendigen Daten innerhalb eines Bounded Contexts zusammenpassen. Wir könnten sogar behaupten, dass dies als ein separater Ansatz zur Lösung unseres ursprünglichen Problems betrachtet werden könnte.

Zusammenfassung

Mit ACID-Transaktionen können wir uns auf Fachlogik konzentrieren und müssen uns nicht um die Fehlererkennung, Wiederherstellung und die komplexen Lösungen von Nebenläufigkeitsproblemen bei gemeinsam genutzten Daten kümmern. In vielen Fällen steht uns eine Lösung zur Verfügung, die keine aufwendige Anpassung erfordert. Wenn wir jedoch den Bereich der Monolithen verlassen und zu Microservices wechseln, gibt es einige Aspekte, die wir berücksichtigen müssen. In Microservice-Architekturen scheinen ACID-Eigenschaften für serviceübergreifende Transaktionen nicht wirklich erreichbar zu sein. Es existieren jedoch alternative Ansätze, die bei der Architekturplanung in Betracht gezogen werden sollten, allerdings mit ihren jeweiligen Besonderheiten.

Und zwar, wenn wir Transaktionen in so einem verteilten System benötigen, die über mehrere Services in unserem System hinausgehen, haben wir im Prinzip zwei Möglichkeiten: Eine davon wäre, das 2PC-Protokoll zu verwenden, in der Hoffnung, die gewünschten ACID-Eigenschaften doch zu erreichen, wobei einige Nachteile dieses Ansatzes in Betracht gezogen werden müssen, oder die andere Möglichkeit wäre, das Konzept der „Eventual Consistency“ zu akzeptieren und einen lockeren Ansatz in Bezug auf Transaktionen und Consistency in Systemen zu wählen, indem die Idee von Saga verwendet wird. Das bedeutet aber nicht, dass ACID-Transaktionen nun komplett irrelevant wären. Die sind immer noch relevant, aber eher im Kontext von einzelnen Services (innerhalb unserer Bounded Contexts).

4 Absicherung von Microservices gegen Ausfall

4.1 Bedeutung von Resilienz für Microservicearchitektur

Wie wir es früher festgestellt haben: Idealerweise sollen einzelne Services so unabhängig wie möglich voneinander sein (sowohl technisch als auch logisch), sodass der Ausfall eines Microservices keine negativen Auswirkungen auf andere Microservices haben soll. Dabei sollen auch Fehler in einem Microservice keine negativen Auswirkungen auf die anderen haben. Wir sagen erst ganz allgemein, dass wir wollen, dass unsere Microservices resilient sind. Was aber genau hinter dieser Idee steckt, wollen wir in diesem Kapitel näher untersuchen.

Das Thema Resilienz ist ein weites und umfassendes Forschungsgebiet, und es ist mit Sicherheit einer gründlichen Untersuchung würdig, was definitiv weit über den Rahmen dieser Arbeit hinausgeht. In dieser Arbeit aber werden wir es primär im Rahmen des Aufbaus einer resilienten Architektur für Microservices untersuchen. Wenn wir von Resilienz im weitesten Sinne sprechen, können wir uns auf die Definition von Uwe Friedrichsen beziehen, die umfangreich und dennoch selbsterklärend ist und wie folgt lautet: „Resilience means that a system can ideally withstand adverse external influences completely or at least recover from them quickly“ [9].

Es wird auch geklärt, dass das System aus der Definition viele Formen annehmen kann, darunter Einzelpersonen, IT-Infrastruktur, Organisationen, Gesellschaften oder Ökosysteme. Die Herausforderungen reichen von Stress und technischen Ausfällen bis hin zu wirtschaftlichen Veränderungen und (Natur-)Katastrophen [33].

Etwas technischer ausgedrückt: Wenn wir wollen, dass unser System resilient ist, müssen wir zunächst die Tatsache akzeptieren, dass Herausforderungen unvermeidlich sind, und dann einen Weg finden, mit ihnen umzugehen. Und in der Welt der verteilten Systeme

sind diese Herausforderungen allgemein als „8 fallacies of distributed computing“ [27] bekannt, die wir akzeptieren bzw. handhaben sollten. Es gibt sogar einen Ansatz namens „design for failures“ [18], der uns dazu bewegen soll, unser System mit dem Gedanken zu bauen, dass alles schiefgehen kann.

Eine kleine Anmerkung lohnt es sich an dieser Stelle zu machen, nämlich dass diese Arbeit nicht den Anspruch erhebt, alle möglichen Aspekte der Absicherung gegen Ausfälle umfassend abzudecken. Vielmehr haben wir uns primär auf bemerkenswerte Herausforderungen und Lösungsansätze konzentriert, die uns später beim Design und der Implementierung des Prototyps unterstützen sollen. Eine vertiefte und ausführliche Erklärung von Fehlertoleranzmustern bietet das Buch „Patterns for Fault Tolerant Software“ von Robert S. Hanmer, das eine systematische Sammlung von Architekturmustern für fehlertolerante Systeme bereitstellt.

Während der Recherche haben wir zudem festgestellt, dass viele Autoren nicht explizit zwischen zwei (verwandten, jedoch auch unterschiedlichen) Begriffen unterscheiden, nämlich Resilienz und Fehlertoleranz. Sie werden manchmal austauschbar oder parallel benutzt, ohne dass man einen klaren Unterschied zwischen den beiden Begriffen sehen kann (siehe [36], [21], [18]), sodass man den Eindruck bekommen kann, dass diese Begriffe tatsächlich eine Art Synonyme sind (siehe auch [16]).

Sowohl Resilienz als auch Fehlertoleranz tragen dazu bei, die reibungslose Funktionalität des Systems zu gewährleisten. Dennoch ist es wichtig zu beachten, dass die Reichweite von Resilienz über Fehlertoleranz hinausgeht. Die erste umfasst nicht nur die Fähigkeit, Fehler (sowohl bekannte als auch unbekannte) zu widerstehen und sich von ihnen zu erholen (Resilienz im engeren Sinne), sondern auch, sich an veränderte Umstände und potenzielle Systemausfälle anzupassen [10]. Obwohl es Überschneidungen gibt, kann man Resilienz auch als einen breiteren Begriff betrachten, während Fehlertoleranz ein engeres Spektrum potenzieller (und meist bekannter) Probleme und dazugehöriger Lösungen im System abdeckt [10].

Für den Zweck dieser Arbeit haben wir uns primär auf die Resilienzstrategien im engeren Sinne konzentriert, die darauf gerichtet sind, vorhersehbare Szenarien bzw. Fehler zu handhaben. In diesem und dem darauf anschließenden praktischen Teil wird daher primär auf Muster und Lösungen eingegangen, die für die Lösung weit verbreiteter Probleme verwendet werden können.

4.2 Fehlertoleranz als Subdomäne der Resilienz

4.2.1 Einführung in die Fehlertoleranz

Unter Fehlertoleranz versteht man eine der grundlegenden und wichtigsten Eigenschaften von Softwaresystemen. Dadurch kann erreicht werden, dass das System trotz teilweiser Ausfälle weiter im Betrieb bleibt [15]. In der heutigen Software-Engineering wird es immer wichtiger, das entwickelte System fehlertolerant zu designen. Für viele Bereiche wie Logistik, Fintech oder Retail spielt die Sicherstellung der Systemzuverlässigkeit und -verfügbarkeit eine superkritische Rolle.

Fehlertoleranz setzt voraus, dass die Fehler nicht zu vermeiden sind, insbesondere wenn es um umfangreiche und komplexe Software geht. Also müssen die Ansätze überlegt werden, die Fehlerauswirkungen minimal zu machen. Eine fehlertolerante Software muss dann so designed werden, dass sie in der Lage ist, weiter korrekt ihre Aufgaben zu erfüllen – auch wenn eine oder mehrere Teilsysteme nicht mehr funktionieren (ausfallen). Um das zu ermöglichen, müssen bei der Software-Design- und -Entwicklung bestimmte Arten von Fehlern, Störungen und Ausfällen betrachtet werden sowie Mechanismen der Fehlererkennung und Fehlerkorrektur implementiert werden.

Es ist aber notwendig, den Unterschied zwischen Fehlern (faults), Störungen (errors) und Ausfällen (failures) klar zu verstehen. Ein Fehler ist ein Defekt im System, z. B. ein Bug oder eine Fehlfunktion der Hardware. Eine Störung zeigt die Auswirkungen des Fehlers auf das Systemverhalten. Und ein Ausfall liegt erst dann vor, wenn das System nicht mehr in der Lage ist, seine spezifizierten Aufgaben zu erfüllen und das Problem nach außen sichtbar ist (also $Fehler \rightarrow Strung \rightarrow Ausfall$) [13]. Hier ist wichtig zu notieren, dass das erwartete Verhalten erstmal durch eine Systemspezifikation definiert sein soll, ansonsten ist es fast unmöglich festzustellen, ob irgendwas nicht in Ordnung ist und ob ein Ausfall vorliegt. Also werden die Ausfälle strikt mit Systemanforderungen und Benutzer-Experience verbunden.

Systemstörungen sind auch nach ihren Formen zu unterscheiden. Einige davon werden sich durch falsche Werte äußern und führen zu Inkonsistenzen – bei den Berechnungen oder bei der Datenspeicherung. Andere werden durch endlose Schleifen oder Probleme mit der Synchronisation entstehen, die den Systembetrieb beeinträchtigen. Systeminstabilität kann auch wegen unzureichender Verarbeitung der Überlastungen, Inkonsistenzen zwischen Speicherorten oder Protokollverletzungen entstehen. Sogenannte byzantinische

Ausfälle scheinen besonders problematisch zu sein, da bei diesen eine Diagnose und Korrektur erschwert sind [13].

4.2.2 Metriken der Zuverlässigkeit und Verfügbarkeit in fehlertoleranten Systemen

Die Wirksamkeit eines fehlertoleranten Software-Systems wird durch den Einsatz von mehreren Metriken bewertet. Zuverlässigkeit (Reliability): Die Wahrscheinlichkeit, dass das System über einen bestimmten Zeitraum fehlerfrei arbeitet:

$$reliability = e^{-(\frac{1}{MTTF})} \quad [13]$$

wo die Ausfallrate ($\frac{1}{MTTF}$) angibt, wie oft es pro Zeiteinheit zu einem Ausfall kommt. Diesen Zeitraum misst man auf spezielle Art und Weise. Im Beispiel oben wird dafür MTTF benutzt [13].

Die Verfügbarkeit (availability) wird als Zeitanteil betrachtet, in dem das System betriebsbereit bleibt:

$$availability = \frac{MTTF}{MTTF + MTTR} \quad [13]$$

Die Verfügbarkeit wird als Prozentsatz dargestellt und ist besonders relevant für die Software-Systeme, in denen die Ausfallzeiten schwierige Konsequenzen bringen können. So wird z. B. für kritische Infrastrukturen die Verfügbarkeit gefordert, die sehr häufig bei einer Verfügbarkeit von 99,999 % liegt [25]. Das wird auch als sogenannte „Five-Nines“ [25] bekannt. Zuverlässigkeit und Verfügbarkeit werden zwar in der Regel in einem Zusammenhang gesehen, sind aber unterschiedliche Eigenschaften. Ein System kann nicht besonders zuverlässig sein, aber gleichzeitig eine hohe Verfügbarkeit haben. Voraussetzung wären dann die effizienten Mechanismen der schnellen Wiederherstellung.

In umfangreichen Software-Systemen werden die Zuverlässigkeit und hohe Verfügbarkeit durch sehr feines Management der konkurrierenden Anforderungen erreicht. Dazu gehört etwa das Gleichgewicht zwischen Kosten, Komplexität des Systems und Entwicklungsaufwand. Interessant wäre auch, die praktische Seite der MTTF und MTTR zu vergleichen. Ein klassisches Beispiel, in dem hohe MTTF besonders kritisch ist, sind die Weltraummissionen, weil die Reparaturen vor Ort fast unmöglich sind [13]. In Businesssystemen wird aber häufiger ein relativ niedriger MTTR vorausgesetzt – wichtig wäre die Möglichkeit, das System schnell wiederherzustellen [25]. Es ist wichtig, im Hinterkopf

zu behalten, dass MTTF nicht unbegrenzt groß sein kann. Anderenfalls riskieren wir, in „100% availability trap“ [11] zu fallen. Die einfachste auf den ersten Blick Strategie, MTTF kontinuierlich zu erhöhen und MTTR zu ignorieren, ist keine gute Option, wenn wir akzeptieren, dass Fehler in einem verteilten System nicht zu vermeiden sind und die MTTF daher eine Obergrenze haben soll [11].

4.2.3 Prinzipien des fehlertoleranten Systemdesigns

Fehlertolerante Ansätze der Systementwicklung

Die frühzeitige Identifikation der Fehlerquellen und die proaktive Umsetzung der Maßnahmen, die die Auswirkungen mindern, werden beim Design der fehlertoleranten Software-Systeme als grundlegende Prinzipien definiert. Solche fehlertolerante Designweise wird durch systematische Analyse der potenziellen Fehler in jeder Entwicklungsphase ausgedrückt. Dazu gehören alle Aspekte des Software-Lebenszyklus – von der Anforderungsdefinition bis zum Betrieb und zur Wartung.

Eine der zentralen Aufgaben beim Design von fehlertoleranter Software liegt in der Abwägung zwischen zwei Aspekten – Zuverlässigkeit und Verfügbarkeit. Die Anforderungen variieren je nach Business oder Branche.

Minimaler MTTR wird besonders wichtig eingestuft, wenn es um Telekommunikation geht. Da braucht man eine schnelle Wiederherstellung nach einem Ausfall. Eine hohe MTTF spielt eine sehr große Rolle in der Logistik sowie in der Luft- und Raumfahrt, da die Folgen eines Softwarefehlers katastrophal sein können. Beides (MTTR und MTTF) je nach Einzelgeschäft in verschiedenen Proportionen wird im Fintech erforderlich – um Unterbrechungen im Dienst und Dateninkonsistenzen zu vermeiden. Um Design fehlertolerant zu gestalten, werden folgende Prinzipien definiert [13]:

- Simplicity (Keep It Simple, KIS): Systemkomplexität möglichst minimal gestalten, so wird das Fehlerrisiko reduziert
- Iterative Verbesserung: Das System wird regelmäßig analysiert und optimiert. So können auch die komplexeren Probleme schon früh erkannt werden. Das erhöht die Stabilität des Systems.
- Defensive Software-Entwicklung: Die Software wird auf Unerwartetes so vorbereitet, dass kleinere Fehler nicht zu kritischen Problemen auswachsen können.

- Optimierung der Datenstrukturen: Auf diese Weise werden Datenkorruption und Inkonsistenzen vermieden.

Mechanismen der Fehlertoleranz

Wie erreicht man eigentlich das spezifizierte Niveau der Fehlertoleranz? Wenn es um mögliche Lösungen geht, spielt Redundanz eine der entscheidenden Rollen. Man unterscheidet dabei grundsätzlich drei Arten [13]:

- Räumliche Redundanz (spatial redundancy): Mehrere Komponenten können potentiell dieselbe Funktionalität übernehmen. So wird das System ausfallsicher.
- Zeitliche Redundanz (temporal redundancy): Kritische Berechnungen führt man mehrmals durch. Die Richtigkeit der Ergebnisse stellt man so sicher.
- Speicherredundanz (storage redundancy): Es geht hier hauptsächlich um Backups. Zum Beispiel werden die Daten in der Cloud gespeichert oder auch auf mehrere „on-prem“-Speicher verteilt. So verringert man das Datenverlustrisiko.

4.2.4 Hauptansätze zur Fehlertoleranz

Phasen der Fehlerverarbeitung

Wie gehen wir vor, wenn die Aufgabe darin besteht, ein System fehlertolerant zu designen? Man betrachtet dabei vier wichtige Schritte [13]:

1. Fehlererkennung (error detection): Das Problem (Anomalie, Fehler) identifizieren wir erstmals mit unterschiedlichen Prüfungen. Dazu gehören Protokolle, Assertions oder auch automatisierte Tests.
2. Fehlerkorrektur (error recovery): Gedacht ist, dass wir das System in einen korrekten Zustand wiederbringen. Es lassen sich zwei Wege unterscheiden: Ein Fehler kann nachträglich korrigiert werden (vorwärtsgerichtete Korrektur) oder wir stellen den Stand vor dem Fehler wieder her (rückwärtsgerichtete Wiederherstellung).
3. Mitigation des Fehlers (error mitigation): Fehler wird isoliert und dadurch verringert man die Eskalation.

4. Fehlerbehandlung (fault treatment): Die Ursachen des Fehlers beseitigt man mit Software-Updates.

Forward und Backward Recovery

Im Kontext von Long-Running-Transactions wird zwischen zwei zentralen Ansätzen für Wiederaufnahme unterschieden [12], [5].

1. Backward Recovery (Rückwärtswiederherstellung)
   - Zurücksetzen des Systems auf korrekten früheren Zustand.
   - Erfolgt unter Nutzung von Checkpoints (Checkpointing) oder durch Protokoll der Ausführung (Audit-Trail).
   - Der Log über durchgeführte Aktionen und das Wissen, wie sie rückgängig gemacht werden können, sind die Voraussetzung.
2. Forward Recovery (Vorwärtswiederherstellung)
   - Es geht nicht um das Zurücksetzen des Systems, sondern der „Ist-Zustand“ wird korrigiert und es ist dann wieder konsistent.
   - Forward Recovery wird meistens durch Exception Handling umgesetzt.
   - Besonders geeignet im Falle, dass schon ausgeführte Aktionen nicht (oder sehr schwierig) rückgängig gemacht werden können.

Im Kontext von verteilten Systemen scheint Backward-Recovery im klassischen Sinne relativ problematisch zu sein, da nicht alle Services unbedingt ein Rollback unterstützen. Etwa eine Banküberweisung oder Produktversand – sie können nicht einfach rückgängig gemacht werden. Da wird eine kompensierende Gegenaktion benötigt [5]. Also ist so eine Kompensation nicht als ein Rollback im klassischen Sinne zu betrachten, sondern es geht eher um sogenanntes semantisches Rollback, das die Auswirkungen des Fehlers korrigiert.

4.3 Kompensierende Transactionen

4.3.1 Challenges der Fehlertoleranz in verteilten Systemen

Die Zusammenarbeit der mehreren Services macht die Wiederherstellung nach möglichen Fehlern sehr umständlich. Ein klassisches Problem dabei ist der sogenannte Domino-Effekt. Das bedeutet, dass der Fehler eines Services auch auf andere Services propagieren kann.

Beispielablauf:

- Ein Service überweist Geld auf Konto A.
- Ein zweiter Service zieht das Geld von Konto A ab und überweist es auf Konto B.
- Falls die erste Überweisung fehlschlägt, meldet der zweite Service einen Fehler wegen unzureichender Mittel.

In verteilten Systemen werden statt harter Rollbacks in diesem Fall die kompensierenden Transaktionen eingesetzt. Kompensierende Transaktionen haben wir schon in Kapitel 3 im Kontext von Saga kennengelernt, in diesem Abschnitt wollen wir sie aber eher als separates Muster im Resilienzkontext betrachten.

4.3.2 Anwendung der kompensierenden Transaktionen

Rollback vs. Kompensation

Wie bereits im Rahmen dieser Arbeit gesagt, scheint ein traditioneller Rollback in verteilten Systemen nicht besonders praktikabel zu sein. Es gibt dafür mehrere Gründe, und zwar: Es gibt keine zentrale Zustandskontrollstelle, einige Aktionen sind schwer zu reversieren (klassisches Beispiel: Banküberweisungen). Außerdem ist eine rückwirkende Abwicklung im Fall eines asynchronen Systems normalerweise problematisch.

Von Colombo und Pace [5] wird der Unterschied zwischen Kompensation und Rollback so definiert: Wenn es um einen Rollback geht, werden alle Transaktionsergebnisse vollständig aus dem System gelöscht und das System ist danach wieder in den Zustand „vor dem Fehler“ zurückgesetzt. Durch eine Kompensation wird nichts wirklich gelöscht, sondern es wird eine korrigierende Gegenaktion durchgeführt, um den Fehlereffekt zu fixen. Zum

Beispiel kann eine bereits versandte Bestellung nicht „zurückgesetzt“ werden, sondern es wird eine Rücksendung und schließlich eine Rückerstattung organisiert.

Kompensation in Long-Running-Transaktionen

Das klassische ACID-Modell wäre für Long-Running-Transaktionen nicht geeignet [5]. Zum einen müssen wir die Ressourcen über relativ lange Zeiträume blockieren, was in verteilten Systemen unpraktisch und problematisch ist. Zum anderen werden die externen Services involviert und erfolgreiche Commits können nicht rückgängig gemacht werden. Wie bereits erwähnt, versuchen Microservices, eine lose Kopplung zu erreichen, unter anderem dadurch, dass jeder Service seine eigene Datenbank hat. Würde eine lokale Transaktion irgendwo halbwegs während der Saga-Ausführung scheitern, können wir nicht alle bisher committeten Transaktionen zurückrollen, und zwar sollen wir dort kompensieren.

4.3.3 Praktische Aspekte der Kompensation

Einsatzbereich von Kompensation

Es werden folgende Bedingungen für Kompensation [5] definiert:

- Wenn eine explizite Stornierung der bestimmten Aktionen erforderlich ist (eine Reisebuchung mit Rückerstattung).
- Wenn eine klare Beziehung zwischen Aktionen und deren Kompensation besteht.
- Wenn die notwendigen kompensierenden Aktionen zur Laufzeit dynamisch generiert und ausgeführt werden können.

Kompensation vs. Reparatur

Es ist auch wichtig, den Unterschied zwischen Kompensation und Reparatur klar zu verstehen. Unter Reparatur versteht man einen Zustand, der dem erfolgreichen (richtigen) Transaktionsabschluss entspricht. Wenn es aber um eine Kompensation geht, wird das System in den Zustand vor dem Start der Transaktion zurückgesetzt. In der Praxis werden diese beiden Ansätze auch oft und gern kombiniert.

Anwendungsfälle der Kompensation

In Long-Running-Transaktionen sind die Kompensationen besonders nützlich, wenn es beispielsweise um folgende Use-Cases geht [5]:

- Reisebuchungen: Eine Flugstornierung bedeutet oft auch eine Hostelstornierung.
- Retail und besonders Online-Handel: Da es oft mehrere Beteiligte wie Bank und Versandunternehmen gibt.
- Integration von externen Systemen: Um Datenkonsistenz sicherzustellen.

4.4 Stabilitätsmuster

Der Fokus lag bisher auf dem Muster „kompensierende Transaktion“. Es ist jedoch nur eines von vielen, um die Resilienz von Microservices und mögliche Fehlerszenarien zu adressieren.

Die Auseinandersetzung mit allen möglichen Stabilitätsmustern würde den Rahmen dieser Arbeit sprengen, daher werden im Folgenden weitere Stabilitätsmuster mit den entsprechenden Szenarien, in denen sie potenziell angewendet werden können, erwähnt. Basierend auf der Studie [21] wurden folgende sieben Fehlerszenarien mit dazugehörigen Mustern identifiziert:

Fehlerszenario	**Empfohlene Muster/Lösungen**
Totalausfall oder partielle Netzwerkausfälle	Circuit Breaker, Fail Fast, Bulkheads, Correlation ID
Paketverluste im Netzwerk	Retry, Cache, Circuit Breaker
Hohe Latenzzeiten durch Systemüberlastung	Throttling, Timeout, Circuit Breaker, Bulkheads
Inkonsistente Daten in Transaktionen	Saga Pattern, Clearing Transactions
Sicherheitsbedrohungen (z. B. DDoS, Credential Stuffing)	Security by Design, Rate Limiting, Timeouts, Circuit breaker
Lange Antwortzeiten	Asynchronous Messaging, Caching
API-Inkompatibilität nach Änderungen	Version Identifier, Wish List Pattern

Tabelle 4.1: Fehlerszenarien und empfohlene Muster für fehlertolerante Microservices (basierend auf [21])

Dementsprechend gibt es eine Reihe an Mustern, die uns dabei auf dem Weg zu Resilienz unterstützen können. Die gute Nachricht ist, dass die meisten gängigen Bibliotheken eine sehr gute Unterstützung leisten und es für manche Muster out-of-the-box-Lösungen gibt. Das einzige, was man machen soll, ist, sie richtig zu konfigurieren. Dabei sind einige Resilienzmuster (wie Bulkhead) in der Natur von Microservices verankert und die anderen (z. B. Timeout, Circuit Breaker, Fail Fast) soll man explizit implementieren bzw. konfigurieren. [36].

Die richtige Auswahl von Mustern kann uns dabei helfen, die Widerstandsfähigkeit des Systems bzw. Resilienz einzubauen. So kann man durch den richtigen Einsatz von solchen Mustern wie Retry, Circuit Breaker, Bulkhead, SAGA und Asynchronous Messaging die Auswirkungen von Fehlern auf unser System minimieren.

Beim Entwurf von Microservices sollten wir jedoch immer die Möglichkeit von Fehlern im Hinterkopf behalten. Die Fähigkeit der Anwendung, sich von Fehlern zu erholen, hilft

uns nicht nur, die Herausforderungen der verteilten Natur unseres Systems zu mildern, sondern erhöht auch die Zuverlässigkeit unseres Systems.

Zusammenfassung

Es gibt bestimmte Muster und Strategien, die uns auf dem Weg zu Resilienz unterstützen. Dabei ist zu bedenken, dass mit zunehmender Komplexität der Resilienzstrategie auch das Risiko von unbemerkten Fehlern steigt. Resiliente Systeme sind nicht in der Lage, sämtliche Fehler zu verhindern, jedoch verfügen sie über Mechanismen, um sich von einigen dieser Fehler zu erholen und angemessen darauf zu reagieren.

Das System ist so zu entwerfen, dass es auch im Falle von Fehlern in einen konsistenten Zustand überführt wird (beispielsweise durch Retry-Mechanismen, Idempotenz und kompensierende Transaktionen). Es soll an dieser Stelle aber darauf hingewiesen werden, dass „resiliency is more than just implementing a few patterns“ [22].

Kompensierende Transaktionen sind z.B. nicht einfach ein weiteres Muster im Resilienz-Bereich. Sie sind auch von erheblicher Bedeutung bei der Umsetzung von Saga und werden dort eingesetzt, um zum einen bereits durchgeführte Aktionen rückgängig machen zu können und zum anderen, um eine verteilte Fehlerbehandlung zu ermöglichen. So tragen Sagas durch kompensierende Transaktionen zur Resilienz von Microservice-Systemen bei, wobei der Beitrag auch indirekt erfolgt.

5 Implementierung des Prototyps

Im Rahmen dieser Arbeit haben wir uns mit dem Konzept, der Architektur sowie der praktischen Implementierung eines Backend-Prototyps für Customer to customer (C2C)-Marktplatz auseinandergesetzt. Der Fokus liegt dabei auf dem Backend-Teil des Microservices-Systems. Unser letztendliches Ziel besteht darin, das spezifische Muster zur Implementierung unserer systemübergreifenden Transaktionen umzusetzen und zu bewerten.

Wichtig zu erwähnen ist, dass das System als ein Prototyp zu betrachten ist und nur rudimentäre Features einer C2C-Marktplatz-Anwendung implementiert wurden. Deshalb werden bestimmte Aspekte wie Sicherheit (Authentifizierung und Autorisierung) oder gesetzliche Vorschriften (besonders wenn es um Bearbeitung/Speicherung der Personaldaten geht) absichtlich außer Acht gelassen. Der Schwerpunkt liegt daher auf der Entwicklung eines einfach zu bedienenden Prototyps sowie der Implementierung von Saga, nicht jedoch auf der Erstellung einer vollständig funktionalen, kommerziellen Softwarelösung.

5.1 Systemanforderungen

Die Entwicklung des C2C-Marktplatz-Prototyps wurde mit einer Analyse und Definition der funktionalen und nicht-funktionalen Anforderungen eingeleitet. Die Domäne, in der die Entwicklung durchgeführt wurde, umfasste verschiedene Funktionen des Marktplatzes, einschließlich Produktlisting, Produktsuche, Auftragsmanagement (Aufgabe der Bestellung, Stornierung der Bestellung), Zahlungsabwicklung, Benachrichtigungen und andere Hilfsaspekte. Neuanlegen des Kontos und Anmeldung gehören auch nicht zum Scope unseres Prototyps, da das System davon ausgeht, dass alle Benutzerinteraktionen bereits autorisiert sind.

5.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die businesskritische Funktionalität. In unserem Fall liegt der Fokus auf Produktsuche, Auftragsmanagement und Zahlungsabwicklung. Im Rahmen der Anforderungsanalyse wurden auch kleine Use Cases erstellt, die die grundlegende Funktionalität abdecken sollen (siehe dazu den Anhang A.1).

Die wichtigsten funktionalen Anforderungen in unserem System umfassen:

- Produktlisting und -verwaltung: Mit dieser Funktion können Verkäufer Produkte hinzufügen, aktualisieren oder löschen.
- Produktsuche: Käufer können Produkte nach ID, Kategorie und Preis suchen.
- Bestellerstellung und Benachrichtigung: Unterstützt den Käufer bei der Bestellerstellung und sorgt dafür, dass alle Beteiligten rechtzeitig informiert werden.
- Zahlungsabwicklung: Es werden Zahlungen über ein externes Zahlungsgateway innerhalb eines bestimmten Zeitraums abgewickelt.
- Stornierung der Bestellung und Rückerstattung: Käufer können Bestellungen innerhalb von 12 Stunden nach Zahlung ohne Grund stornieren. Sie erhalten das Geld automatisch zurück. Verkäufer werden davon entsprechend benachrichtigt.

5.1.2 Nicht-Funktionale Anforderungen

Die nicht-funktionalen Anforderungen in unserem System stellen Qualitätsmerkmale dar, die gewährleisten, dass das Prototyp-System wartbar, zuverlässig, skalierbar und belastbar ist. Während sich funktionale Anforderungen auf die tatsächliche Funktionalität des Systems und die abgedeckten Geschäftsaspekte konzentrieren, bestimmen nichtfunktionale Anforderungen die Qualität und Leistung der Systemarbeitsweise unter realen (oder auch realitätsnahen) Bedingungen.

Für unseren Marktplatz-Prototyp sind folgende nicht-funktionale Anforderungen als relevant zu betrachten:

- Leistung und Systemstabilität, um eine Vielzahl von gleichzeitigen Anfragen effizient zu verarbeiten.

- Zuverlässigkeit, um sicherzustellen, dass unser System auch bei Problemen in der Lage ist, weiterzuarbeiten oder sich davon zu erholen.
- Wartbarkeit und Erweiterbarkeit, um künftige Änderungen oder Anpassungen sowie modulare Verbesserungen zu ermöglichen.

Diese Anforderungen sollen im Weiteren sicherstellen, dass unser System stabil, robust und an zukünftige Anforderungen anpassbar bleibt.

5.2 Lösungsstrategie

5.2.1 Technologieentscheidungen

Java (21) [1] und Spring Boot [2] wurden aufgrund ihrer Stabilität, modernen Funktionen und eines umfassenden Ökosystems ausgewählt. Außerdem sind sie gut dokumentiert und verfügen über eine großartige Entwickler-Community, was auch gewissermaßen zur Erfüllung nicht-funktionaler Anforderungen beiträgt.

Maven [3] als Build-Tool: Maven übernimmt das Abhängigkeitsmanagement und automatisiert verschiedene Aufgaben wie Kompilieren und Verpacken.

Eureka [4] für Service Discovery: Die dynamische Erkennung von Diensten ist unerlässlich. Eureka bietet ein robustes Register, das die verfügbaren Dienste verfolgt und so eine nahtlose Skalierung und einfachere Lastverteilung ermöglicht.

OpenTelemetry [5] und Spring Boot Actuator [6]: OpenTelemetry bietet einheitliche Tools für die Ablaufverfolgung und Metriken über verschiedene Komponenten hinweg, während Actuator integrierte Endpunkte zum Überprüfen der Anwendungsintegrität, Metriken und Konfigurationsdetails bietet. Zusammen machen sie es viel einfacher, Probleme in einer verteilten Umgebung zu erkennen und zu beheben.

[1] https://www.oracle.com
[2] https://spring.io/projects/spring-boot
[3] https://maven.apache.org
[4] https://spring.io/guides/gs/service-registration-and-discovery
[5] https://opentelemetry.io
[6] https://spring.io/guides/gs/actuator-service

REST [7] für externe Kommunikation: Die Verwendung von RESTful-APIs ist aufgrund ihrer Einfachheit und breiten Akzeptanz eine natürliche Wahl für die externe Kommunikation.

Apache Kafka [8] für interne Kommunikation: Asynchrone Nachrichtenübermittlung über Kafka entkoppelt Microservices und ermöglicht die Kommunikation, ohne auf synchrone Antworten warten zu müssen. Dieser Ansatz erhöht die allgemeine Resilienz und Skalierbarkeit des Systems, indem direkte Abhängigkeiten reduziert werden.

GitHub Actions und GitHub-Packages [9]: GitHub Actions automatisieren solche Aufgaben wie Erstellen, Testen und Überprüfen von Codeänderungen. CI-Pipeline veröffentlicht dann im Anschluss Artefakte in GitHub-Packages und vereinfacht so die Versionskontrolle und das Artefaktmanagement.

Local Deployment mit Docker [10]: Docker bietet für jeden Service eine isolierte Umgebung, in der sie unabhängig voneinander ausgeführt werden können.

PostgreSQL [11] für Datenpersistenz: Jeder Microservice hat die Hoheit über seine eigenen Daten, die in einer PostgreSQL-Datenbank gespeichert sind.

5.2.2 Highlevel Architektur- und Designentscheidungen

Jeder Microservice im Prototyp kapselt seine eigene Geschäftslogik und persistiert die für ihn relevanten Daten. Dies fördert eine lose Kopplung und unterstützt eine inkrementelle Entwicklung – neue Funktionen können eingeführt oder erweitert werden, ohne dass das gesamte System wesentlich geändert werden muss.

Schichtenarchitektur innerhalb jedes Services: Innerhalb jedes Mikroservices trennt ein traditioneller Schichtansatz die Verantwortlichkeiten.

Saga-Muster: Um die Zusammenarbeit von mehreren Services zu koordinieren, wird die Konsistenz über alle Systeme hinweg durch Saga sichergestellt. Für den Zweck dieser Arbeit haben wir uns für Choreografie-Saga entschieden. Einerseits bietet uns dieser

[7] https://spring.io/guides/gs/rest-service
[8] https://kafka.apache.org
[9] https://github.com
[10] https://www.docker.com
[11] https://www.postgresql.org

Ansatz maximale Flexibilität und lose Kopplung, andererseits passt er auch zur Größe und Komplexität unseres Systems.

Outbox-Pattern: Durch das Outbox-Pattern wird ferner sichergestellt, dass die Speicher-Vorgänge in die Datenbank und das Event-Publishing in Kafka atomar erfolgen. Damit sollen die Szenarien vermieden werden, in denen die Datenbank aktualisiert wird, aber die Nachricht jedoch nie gesendet wurde. Dies gilt auch umgekehrt.

5.3 Architekturüberblick

Zunächst wird ein Überblick über die Architektur im Kontext der anderen Systeme gegeben. Unser System („Marketplace System“ in der Abbildung 5.1 als Black-Box dargestellt) übernimmt die zentrale Verantwortung für die Umsetzung der funktionalen Anforderungen sowie die Koordination zwischen den einzelnen Microservices. Für die Zahlungsabwicklung ist ein externes System (Stripe) angebunden, so dass sensible Zahlungsinformationen nicht von unserem System selbst verarbeitet werden müssen. Gleichzeitig nutzt das System einen SMTP-Server, um Nachrichten zuverlässig zu versenden. Der Nutzer kann über definierte Schnittstellen auf unser System zugreifen.

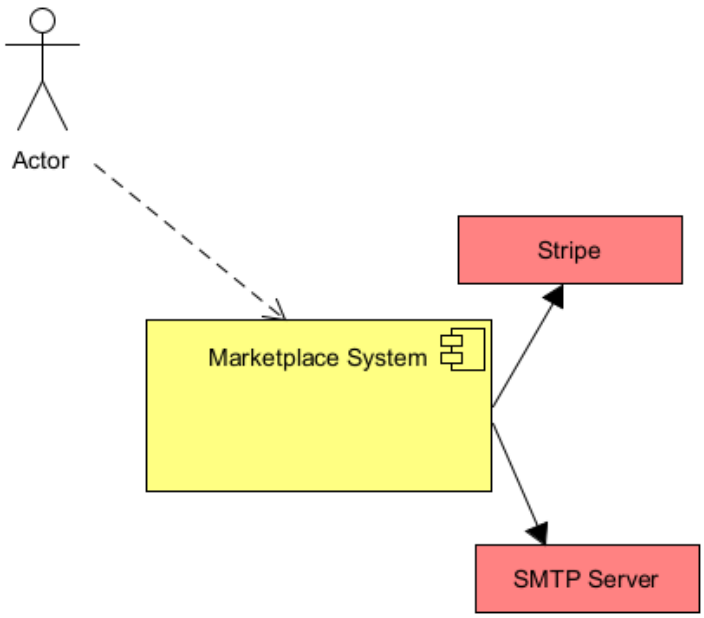


Abbildung 5.1: Kontextabgrenzung des Systems

Nachbarsystem / Benutzer	**Beschreibung**	**Protokoll**	**Format**	**Input**	**Output**
Actor	Der Endnutzer, der Bestellungen aufgibt und Zahlungen tätigt.	HTTP (REST API)	JSON	Bestellung erstellen, Zahlung abwickeln	Bestellbestätigung, Zahlungsstatus
Stripe API	Zahlungsdienstleister für Transaktionen im Marketplace System.	HTTPS (REST API)	JSON	`POST /v1/checkout/sessions` mit Betrag, Währung, Zahlungsmethode	JSON-Antwort mit PaymentIntent-ID, Zahlungsstatus
Stripe Webhooks	Asynchrone Benachrichtigungen über Zahlungsstatus.	HTTPS (Webhook)	JSON	Event (`checkout.session.expired`, `checkout.session.completed`)	Zahlungsbestätigung, Zahlungsstatus-Update
SMTP Server	E-Mail-Dienst, der Bestell- und Zahlungsbestätigungen versendet.	SMTP	MIME (Text/ HTML)	E-Mail-Inhalt (`TO`, `FROM`, `SUBJECT`, `BODY`)	Versandbestätigung, E-Mail-Zustellstatus

Tabelle 5.1: Kommunikation mit externen Systemen im Prototyp

Die Abbildung 5.2 stellt das System als „White-Box“ dar, in der jeder Service eine klar abgegrenzte Aufgabe zu erfüllen hat. Jeder Microservice hat eine klar definierte Verantwortung und verfügt über eine eigene Datenbank.

Die Kommunikation zwischen den einzelnen Services erfolgt asynchron über einen Kafka-Broker. Dadurch lassen sich komplexe Workflows zuverlässig und skalierbar abbilden. Das API-Gateway fungiert als zentraler Einstiegspunkt für externe Anfragen, leitet diese an die zuständigen Microservices weiter und abstrahiert dabei die interne Komplexität des Systems für die Nutzer. Das System kann man leicht erweitern, ohne dass man dafür die Gesamtstruktur ändern muss. Neue oder angepasste Services beeinträchtigen die Struktur nicht, die Verantwortlichkeiten bleiben klar getrennt und die Wartung der einzelnen Komponenten ist überschaubar.

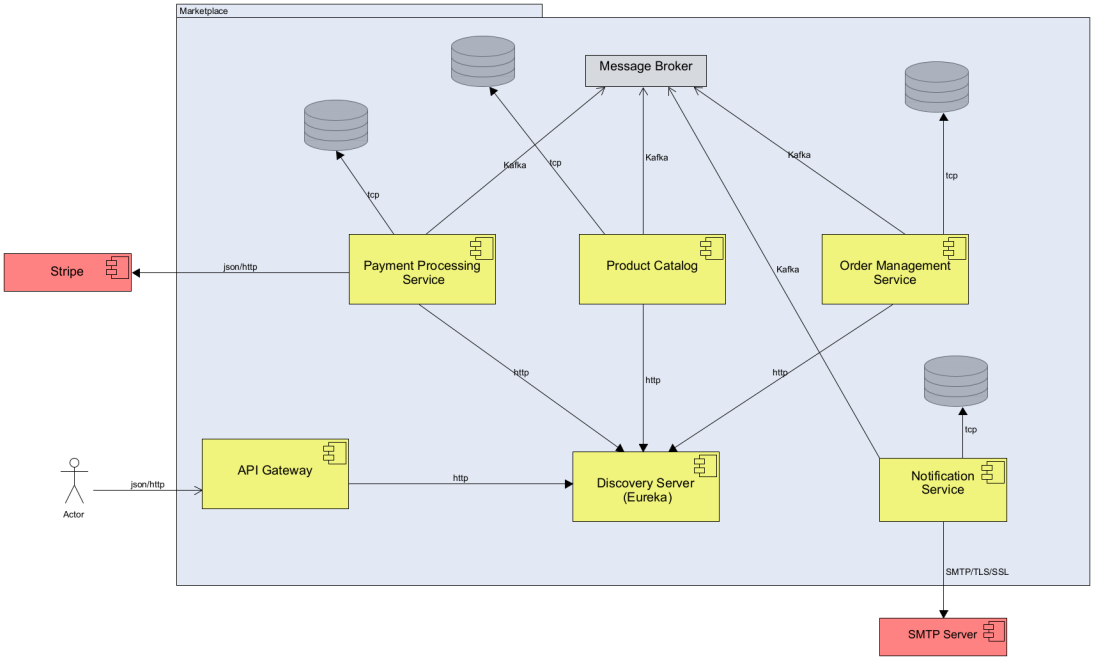


Abbildung 5.2: Komponentendiagramm des Systems

Element	**Verantwortlichkeit / Bedeutung**
API Gateway	Zentraler Einstiegspunkt für externe Anfragen; leitet Requests an die verantwortlichen Services im System weiter.
Discovery Server (Eureka)	Ermöglicht Service-Discovery Mechanismus innerhalb des Systems, pflegt Übersicht/Gesundheit aller verfügbaren Instanzen von registrierten Services.
Payment Processing Service	Kümmert sich um Zahlungsabwicklung, Zahlungsstatus-Updates und Kommunikation mit Stripe.
Product Catalog	Gibt Informationen über Produkte, verwaltet Reservierungen von Produkten, verfügt über Schnittstellen für das Hinzufügen neuer Produkte und für das Entfernen von Produkten aus dem Sortiment.
Order Management Service	Erstellt Bestellungen und pflegt den Bestellstatus.
Notification Service	Versendet Benachrichtigungen über relevante Statusänderungen und Ereignisse.
Message Broker	Ermöglicht asynchrone Kommunikation und entkoppelt die Services.
Datenbanken per Service	Persistieren Daten der jeweiligen Services; dabei hat jeder Service sein eigenes fachliches Datenmodell.

Tabelle 5.2: Elemente des Systems und ihre Verantwortlichkeiten

5.4 Teststrategie

Im Rahmen unserer Teststrategie haben wir einzelne Komponenten des Systems selbstverständlich mit Unit-Tests getestet. Der Fokus lag dabei auf der Absicherung der Kernkomponenten und deren Funktionalitäten in isolierter Umgebung.

Integrationstests sind zu einem wichtigen Bestandteil unserer Systemtests geworden. Die größte Herausforderung bestand darin, die Abdeckungstiefe zu bestimmen: Einerseits musste das Zusammenspiel von Schlüsselkomponenten wie Kafka und der Datenbank getestet werden, andererseits erwies sich die technische Implementierung solcher Tests als recht umständlich und verlangsamte die Pipeline. Deswegen haben wir dann den Umfang der Integrationstests absichtlich eingeschränkt. Wir haben beispielsweise die Integration mit dem Abrechnungsdienst Stripe ausgeschlossen. Grund dafür war, dass deren Implementierung eine spezielle Infrastruktureinrichtung erforderte und über den Umfang der in dieser Arbeit behandelten Aufgaben hinausging. Also haben wir uns darauf konzentriert, die Nachrichtenübermittlung zwischen den Kernkomponenten und die Datenspeicherung zu testen sowie schnelle Feedbackschleifen aufrechtzuerhalten.

Da unsere Services intern nur asynchron kommunizieren, haben wir während Integrationstests den Ansatz gewählt, bei dem wir die Kommunikation von einzelnen Services (sei es auf Producer- oder Consumer-Seite) mit Kafka getestet haben. Also wie es von E. Wolff vorgeschlagen wurde, wollen wir nur sicherstellen, dass unsere Komponenten „sich wechselseitig erreichen können“ [36]. Dafür haben wir u. a. `EmbeddedKafkaBroker` [12] benutzt. Dabei verlassen wir uns stark auf Kafka und auf manuell von uns auf der Producer-Seite konfigurierte „exactly-once delivery“ - Semantik (siehe Anhang D.1) und auf vorgegebene für uns „at-least-once delivery“ - Semantik auf der Consumer-Seite. Die Tests für die Kommunikation mit der Postgres haben wir mithilfe von Testcontainers durchgeführt [13].

Zudem wurden eingeschränkte Lasttests mit Gatling [14] durchgeführt. So wäre es möglich, einzelne Eintrittspunkte unter realitätsnahen Bedingungen zu überprüfen. Es ist jedoch zu berücksichtigen, dass alle Experimente in einer lokalen Umgebung durchgeführt wurden.

[12] https://docs.spring.io/spring-kafka/reference/testing.html/
[13] https://docs.spring.io/spring-boot/reference/testing/testcontainers.html
[14] https://docs.gatling.io/tutorials/scripting-intro/

Wir haben das Gesamtsystem aufgrund des fehlenden UIs vor allem durch manuelle End-to-End-Tests validiert. Einerseits ist die Anzahl der zu überprüfenden Szenarien begrenzt (tatsächlich sind es nur zwei zentrale Saga-Szenarien), andererseits steigt der Aufwand für eine vollautomatisierte Testumgebung erheblich, wenn externe Systeme (wie Stripe oder der Gmail SMTP-Server in unserem Fall) eingebunden sind. Der Nutzen einer solchen Umgebung wäre dabei nicht gerechtfertigt. Stattdessen wurde die Priorität auf manuelle Tests gelegt, um das Gesamtsystem schnell und effektiv zu validieren. Sollten zukünftig die Anforderungen steigen oder sich die Komplexität des Systems erhöhen, könnte man den Einsatz von Mocks und Stubs für automatisierte End-to-End-Tests in Betracht ziehen.

Der Schwerpunkt der Unit- und Integrationstests lag auf der Ebene der einzelnen Microservices, um deren korrekte Funktionsweise zu gewährleisten. Eine manuelle Gesamtprüfung sollte anschließend die reibungslose Interaktion aller Services bestätigen.

Im weiteren Verlauf werden die wesentlichen Aspekte der Umsetzung erläutert und die entsprechenden Strategien und Vorgehensweisen präsentiert.

5.5 Resilienzstrategien

Für die Absicherung haben wir verschiedene Sicherheitsmuster eingebaut bzw. bereits vorhandene Lösungen aus den Spring-Bibliotheken konfiguriert.

Dafür wurden im API-Gateway entsprechende Filter von Spring Cloud Gateway konfiguriert. Die Entscheidung, diese Schutzmechanismen im API-Gateway zu implementieren, basiert darauf, dass das API-Gateway zentraler Einstiegspunkt für externe Anfragen ist. Da sämtliche externe Zugriffe über diesen Service laufen, ist es von besonderer Bedeutung, den Eingang zu System robust zu gestalten und gleichzeitig dieses vor Überlastungen und fehlerhaften Antworten einzelner Dienste zu schützen. Um das zu erreichen, haben wir u. A. Circuit Breaker, den Request-Rate-Limiter und Retry eingesetzt.

Circuit Breaker

Der Circuit Breaker verhindert Überlastungen durch externe Services, die fehlerhafte Antworten liefern oder unerreichbar sind. In Szenarien, in denen während der Bearbeitung der Anfragen entweder Timeouts auftreten oder der Service komplett ausfällt, verhindert der

Circuit Breaker, dass alle Anfragen weiterhin an diesen Service weitergeleitet werden, sobald eine bestimmte Grenze an fehlerhaften oder langsamen Anfragen überschritten wird. In unserer Konfiguration wird hierfür der Circuit-Breaker-Filter verwendet. Sobald bestimmte Schwellenwerte erreicht werden, wird der Schaltkreis automatisch geöffnet (siehe Ausschnitt in Listing 5.1). Ein Beispiel für einen solchen Schwellenwert wäre eine Fehlerquote von über 50 % bei mindestens 10 Anfragen oder ein hoher Anteil langsam reagierender Anfragen (länger als 6 Sekunden). In diesem Zustand werden Anfragen an einen vordefinierten Fallback-Endpunkt (fallbackUri: `forward:/fallback/marketplace`) weitergeleitet. Der Mechanismus greift bei Netzwerkproblemen oder Timeouts, wenn etwa der zuständige Service für die Bearbeitung der Anfrage langsam reagiert oder gar nicht antwortet. Dazu kommen auch mögliche interne serverseitige Fehler. Dadurch wird verhindert, dass das System durch wiederholte fehlerhafte Aufrufe zusätzlich belastet wird. Dafür haben wir Spring Cloud Circuit Breaker [15] verwendet.

```
{
  "status": "UP",
  "components": {
    "circuitBreakers": {
      "status": "UNKNOWN",
      "details": {
        "orderServiceCircuitBreaker": {
          "status": "CIRCUIT_OPEN",
          "details": {
            "failureRate": "100.0%",
            "failureRateThreshold": "50.0%",
            "slowCallRate": "0.0%",
            "slowCallRateThreshold": "60.0%",
            "bufferedCalls": 10,
            "slowCalls": 0,
            "slowFailedCalls": 0,
            "failedCalls": 10,
            "notPermittedCalls": 2,
            "state": "OPEN"
          }
        }
      }
    },...}}
```

Listing 5.1: Schaltkreis in einem geöffneten Zustand

[15] https://docs.spring.io/spring-cloud-circuitbreaker/docs/current/reference/html/

Rate Limiting

Um das API-Gateway vor einem externen Overload zu schützen – sei es durch viele gleichzeitige Anfragen oder durch missbräuchliche Angriffe – wird ein Request-Rate-Limiter konfiguriert. Dieser Mechanismus begrenzt die Anzahl der Anfragen, die innerhalb einer bestimmten Zeitspanne von einem Client mit einer bestimmten IP verarbeitet werden. In der vorliegenden Konfiguration wird ein Key-Resolver verwendet, basierend auf der IP-Adresse des Clients. Mit einer Replenish-Rate von 5 Anfragen pro Sekunde und einer Burst-Capacity von bis zu 100 Anfragen wird sichergestellt, dass kurzzeitige Lastspitzen abgefangen werden können, ohne das System dauerhaft zu überlasten. Diese Strategie schützt das Backend effektiv vor einer Überflutung und trägt zur allgemeinen Stabilität bei. Rate Limiting trägt auch dazu bei, die Stabilität bei plötzlichen Verkehrsspitzen zu erhalten. Die Last wird dann gleichmäßig verteilt und übermäßige oder potenziell böswillige Anfragen werden verworfen. Die Funktionalität haben wir mithilfe von Spring Cloud Getaway [16] umgesetzt.

Retry

Der Retry-Filter wurde konfiguriert, um temporäre Fehler wie Netzwerkprobleme oder kurzzeitige Nichtverfügbarkeit von Services abzufangen. Wenn eine Anfrage nicht sofort erfolgreich ist, wird sie automatisch wiederholt. In unserer Konfiguration wird festgelegt, dass POST-Anfragen bei bestimmten HTTP-Statuscodes (INTERNAL_SERVER_ERROR, BAD_GATEWAY, SERVICE_UNAVAILABLE, GATEWAY_TIMEOUT) bis zu dreimal wiederholt werden. Dabei wird nicht bei jedem Fehler ein neuer Versuch gestartet, sondern es wird abgewartet. Der erste Backoff beträgt 100 Millisekunden, der maximale 2000 Millisekunden und der Verzögerungsfaktor 2 sorgt dafür, dass die Wartezeiten bei aufeinanderfolgenden Versuchen immer länger werden. Dadurch können temporäre Probleme in der Zwischenzeit behoben werden, ohne dass der Endnutzer direkt einen Fehler bemerkt.

Eine detaillierte Konfiguration dieser Mechanismen kann dem Anhang F.1 entnommen werden.

In verteilten Systemen treten häufig temporäre Fehler wie kurze Netzwerkausfälle oder langsame Datenbankantworten auf. Die Nutzung von Retry-Mechanismen erstreckt sich

[16] https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/gatewayfilter-factories/requestratelimiter-factory.html

dabei weit über den Kontext des `api-getaway` hinaus. Die automatische Wiederholungslogik trägt dazu bei, diese Probleme zu mildern, jedoch nicht zu 100% zu fixen. Sei es RateLimitException, ApiConnectionException oder OptimisticLockingFailureException: Transiente Fehler sollen unser System nicht zum Absturz bringen und auch nicht unsere Saga dazu zwingen, kompensierende Transaktionen auszuführen. Zu diesem Zweck wurde Spring Retry [17] verwendet.

Asynchrone Kommunikation und Idempotenz können ebenfalls als Teil unserer Resilienz-Strategie betrachtet werden. Zum einen sorgt asynchrone Kommunikation dafür, dass Services voneinander entkoppelt sind und bei einem Ausfall eines Dienstes andere Services nicht in Mitleidenschaft gezogen werden. Kommunikation via Kafka fördert zudem Fehlertoleranz und verhindert sogenannte „Cascading Failures“, da Nachrichten verzögert verarbeitet werden, ohne dass das Gesamtsystem oder einzelne Teile dabei blockiert sind. Die Idempotenz von Kafka-Producer und Kafka-Consumer in unserem System sorgt dafür, dass dieses mit Duplikaten umgehen kann.

5.6 SAGA und Outbox

Saga

Wie bereits erwähnt, haben wir die Choreografie-Saga ausgewählt, um eine Fachtransaktion zu implementieren, die die Interaktion mehrerer Services erfordert, um eine gemeinsame Aufgabe zu erfüllen (in unserem Fall waren eine Saga für die Bestellerstellung und eine andere für die Stornierung der Bestellung zuständig).

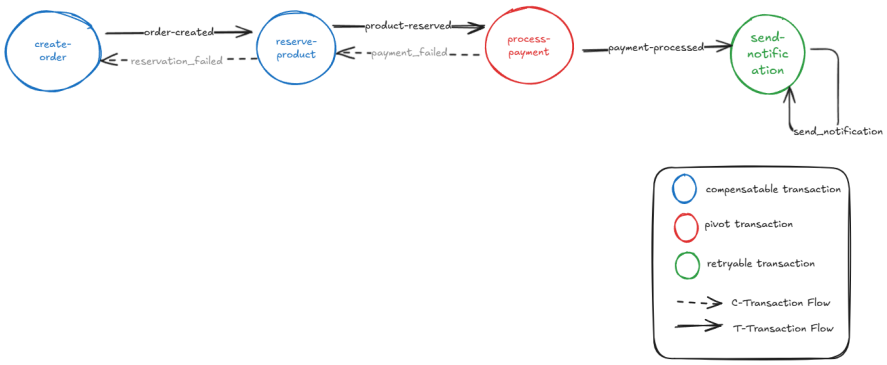


Abbildung 5.3: Workflow für Bestellerstellung

[17] https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Retryable.html

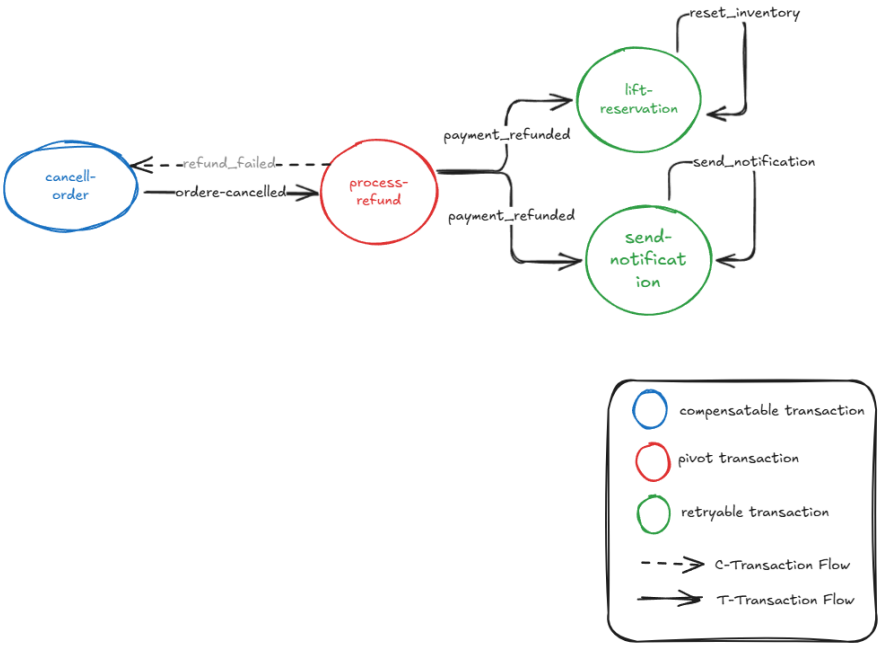


Abbildung 5.4: Workflow für Stornierung der Bestellung

Es gibt keinen zentralen Koordinator, und die Services kommunizieren miteinander asynchron via Kafka. Die Bestellerstellung bzw. Stornierung seitens des Kunden markiert den Beginn der Abwicklung der Saga. Der Gesamtablauf kann dem Sequenzdiagramm H.1 bzw. H.2 entnommen werden.

Wir verlassen uns dabei stark auf die Garantien von Kafka[18] und setzen voraus, dass Kafka fehlertolerant und nahezu immer verfügbar ist. So veröffentlicht jeder Microservice, abhängig vom Ergebnis seiner lokalen Transaktion, ein Domain-Event im jeweiligen „Erfolgs“ - oder „Fehler“ - Topic, und andere Microservices lauschen auf diese Events (siehe Anhang B.1 bzw. C.1 für so einen Kafka-Consumer). Wurde die lokale Transaktion erfolgreich ausgeführt, wird das entsprechende Event publiziert, sodass nachfolgende Services dieses abrufen und basierend darauf ihre Aufgaben erledigen können. Wenn jedoch eine lokale Transaktion nicht erfolgreich abgeschlossen werden kann, publiziert der Service ein Event, damit die in den vorherigen Schritten der Saga vorgenommenen Änderungen rückgängig gemacht werden. Es ist zu beachten, dass ein Topic nur einen Produzenten hat, wobei mehrere Services denselben konsumieren können.

[18] https://kafka.apache.org/documentation/

Die Präsentationsschicht im Order-Management-Service stellt den POST-Endpoint `http://localhost:8083/api/v1/orders` zum Erstellen neuer Bestellungen bereit. Dies stellt den ersten Schritt unserer SAGA dar. Sobald eine Bestellung aufgegeben wird, erhält der Kunde eine eindeutige ID, die verwendet werden kann, um den Bestellstatus zu überprüfen, die Zahlungs-URL abzurufen und den Zahlungsstatus zu überprüfen.

In den nächsten Schritten erfolgen die Produktreservierung (`product-catalog` Service) und die Zahlungsabwicklung (`payment-processing` Service). Sobald eine Zahlungssitzung im `payment-processing` Service erstellt wurde, hat man 30 Minuten, um die Zahlung zu tätigen.

Dabei kann man merken, dass nicht alle Schritte in Saga kompensierbar sind. Auf einer Seite haben wir Zahlungsvorgänge als Pivot-Transaktionen modelliert, d. h., der Ausgang dieser Transaktion bestimmt „Erfolg“ bzw. „Fehler“ für die gesamte Saga. Die darauf folgenden Schritte wurden als wiederholbare Transaktionen modelliert, d. h., wir rechnen damit, dass diese nach einer bestimmten Anzahl von Wiederholungen erfolgreich ausgeführt werden.

Wenn die Zahlung innerhalb von 30 Minuten erfolgreich verarbeitet wurde, können wir nicht zurückgehen und kompensieren. Stattdessen müssen wir unser Bestes tun, um nachfolgende wiederholbare Transaktionen (Benachrichtigungserstellung und Verschicken im `notification` Service) abzuschließen.

Ansonsten, wenn eine Zahlungssitzung abläuft und die Zahlung nicht rechtzeitig getätigt wird (unser System hat einen Webhook, um auf relevante Stripe-Events zu lauschen), wird der `payment-processing` Service via Stripe-Event darüber informiert. Anschließend wird die Kompensierung asynchron durch `payment-processing` Service bzw. (`PaymentFailedEvent`) ausgelöst. `Product-catalog` Service (siehe Handler in 5.2) lauscht auf dieses Event und führt demnächst die Kompensationslogik aus (siehe dafür Methode `liftNotPaidReservation` in 5.3).

```
1 @Component
2 @KafkaListener(topics = {"payments.payment.failed.topic"}, containerFactory =
      "kafkaListenerContainerFactory")
3 @Slf4j
4 public class PaymentFailedEventHandler {
5
6     @Autowired
7     private ReservationService reservationService;
```

```
8     @Autowired
9     private ProcessedEventRepository processedEventRepository;
10    @KafkaHandler
11    @Transactional
12    public void handle(@Payload PaymentFailedEvent event, @Header("messageId"
          ) String messageId,
13                       @Header(KafkaHeaders.RECEIVED_KEY) String messageKey)
                             throws ReservationNotFoundException {
14
15        log.info("Handler␣received␣PaymentFailedEvent=␣{}␣␣", event);
16        ProcessedEventEntity processedEvent = processedEventRepository.
              findByMessageId(messageId);
17        if (processedEvent != null) {
18            log.info("Duplicate␣massageId␣detected=␣{}", messageId);
19            return;
20        }
21        reservationService.liftNotPaidReservation(event);
22        try {
23            processedEventRepository.save(ProcessedEventEntity.builder()
24                    .orderId(event.orderId())
25                    .messageId(messageId)
26                    .build());
27        } catch (DataIntegrityViolationException ex) {
28            log.error(ex.getMessage());
29            throw new NonRetryableException(ex.getMessage());
30        }
31    }
```

Listing 5.2: Consumer-Handler für PaymentFailedEvent

```
1
2     @Override
3     @Transactional
4     @Retryable(
5             value = OptimisticLockingFailureException.class,
6             maxAttempts = 3,
7             backoff = @Backoff(delay = 300)
8     )
9
10    public void liftNotPaidReservation(PaymentFailedEvent event) {
11        Reservation reservation = reservationRepository.findByOrderId(event.
              orderId())
12                .orElseThrow(() -> new ReservationNotFoundException("
                      Reservation␣not␣found␣for␣order␣ID:␣" + event.orderId()))
                      ;
```

```
13
14          //if reservation has already been canceled, just ignore
15          if (reservation.getReservationStatus().equals(ReservationStatus.
                RESERVATION_CANCELED)
16                  || reservation.getReservationStatus().equals(
                        ReservationStatus.RESERVATION_NOT_PAID)) {
17              return;
18          }
19          Product product = productRepository.findById(reservation.getProductId
                ())
20                  .orElseThrow(() -> new ProductNotFoundException("Product␣with
                        ␣ID␣" + reservation.getProductId() + "␣not␣found"));
21
22          PaymentFailedEventproduct.setQuantityAvailable(product.
                getQuantityAvailable() + reservation.getQuantityReserved());
23          productRepository.save(product);
24
25          reservation.setReservationStatus(ReservationStatus.
                RESERVATION_CANCELED);
26          reservationRepository.save(reservation);
27
28          handleCanceledReservation(event, ReservationStatus.
                RESERVATION_NOT_PAID.toString(),
29                  reservation.getProductId(), ReservationStatus.
                        RESERVATION_CANCELED);
30
31      }
```

Listing 5.3: Kompensierende Methode für PaymentFailedEvent

```
1
2       private void handleCanceledReservation(BaseEvent event, String reason,
            Long productId) {
3
4           ProductReservationCanceledEvent productReservationCanceledEvent =
                ProductReservationCanceledEvent
5                   .builder()
6                   .eventId(UUID.randomUUID())
7                   .timestamp(LocalDateTime.now())
8                   .orderId(event.getOrderId())
9                   .productId(productId)
10                  .reason(reason)
11                  .build();
12
13          OutboxTask outboxTask = outboxTaskService
```

```
14              .createOutboxTask(productReservationCanceledEvent,
15                      "SEND_PRODUCT_RESERVATION_CANCELED_EVENT");
16      }
```

Listing 5.4: Helper-Methode für ProductReservationCanceledEvent

Nach der Ausführung der kompensierenden Transaktion wird `ProductReservationCanceledEvent` veröffentlicht. Dieses Event wird seinerseits die Kompensierungslogik im `order-management` Service auslösen.

So werden in Saga Schritt für Schritt (Aktualisierung des Zahlungsstatus in `payment-processing`, Reservierungsaufhebung in `product-catalog` und Stornieren der Bestellung `order-management`) alle C-Transaktionen ausgeführt. Dies dient dazu, vorgenommene Änderungen rückgängig zu machen und das System in einen konsistenten Zustand wiederzubringen. Es ist wichtig, an dieser Stelle zu betonen, dass sowohl C-Transaktionen als auch T-Transaktionen Teil der Fachlogik der jeweiligen Services sind.

Jeder Service, der an Saga beteiligt ist, ist dabei zuständig für seine eigene lokale T-Transaktion und C-Transaktionen. Da diese lokalen Transaktionen Grenzen von Services nicht überschreiten, profitieren wir dort sowohl von ACID-Eigenschaften als auch von transaktionaler Unterstützung durch das Spring-Framework.

Einzelne Services interagieren nicht direkt miteinander, sondern sie abonnieren Topics, lauschen auf relevante Events oder publizieren Events selbst.

Den Status einer Saga kann man jederzeit aus der Datenbank entnehmen. In einer Choreografie-Saga gibt es jedoch keinen zentralen Koordinator, wodurch die Nachverfolgung des Saga-Zustands etwa schwierig ist. Theoretisch wär es denkbar, den Zustand anhand der veröffentlichten Events zu rekonstruieren, also kann ein separater Service z. B. aus den abgehörten Events den Saga-Zustand on-the-fly rekonstruieren, das geht ohne Orchestrator, erfordert aber einen extra Service. Fest bleibt aber, dass jeder Service selbst entscheidet, wann, wie und auf welche Events er reagiert.

Temporäre Inkonsistenzen im System sind möglich, diese gleichen sich aber im Laufe der Eventual Consistency von selbst aus. In so einer beispielhaften Situation behalten wir eine gültige Reservierung für ein Produkt im System für eine Weile, obwohl die Zahlung bisher nicht getätigt ist – oder `payment-processing` Service hat bereits von Stripe ein `checkout.session.expired` Event bekommen, da die Frist abgelaufen ist. Zu dem Zeitpunkt hat `product-catalog` Service noch keine Ahnung davon und hält die

Reservierung weiter für gültig. Erst wenn das `PaymentFailedEvent` beim `product-catalog` Service eintrifft und verarbeitet wird, wird die Reservierung freigegeben. So kann man das System in einem bestimmten Moment in einem inkonsistenten Zustand beobachten, passt sich aber im Sinne von Eventual Consistency automatisch an, sobald alle relevanten Events verteilt und verarbeitet sind.

```
1    @Override
2    @Retryable(
3            value = OptimisticLockingFailureException.class,
4            maxAttempts = 3,
5            backoff = @Backoff(delay = 300)
6    )
7    @Transactional
8    public void reserveProduct(OrderCreatedEvent event) {
9        //...
10       // fetch product
11       if (product.getQuantityAvailable() < 1) {
12           handleFailedReservation(event, ReservationStatus.
                 PRODUCT_OUT_OF_STOCK.toString(), event.productId()
13           );
14           return;
15       }
16       // handling success scenario
17       //...
18   }
19   private void handleFailedReservation(BaseEvent event, String reason, Long
          productId) {
20
21       ProductReservationFailedEvent productReservationFailedEvent =
             ProductReservationFailedEvent
22                 .builder()
23                 .eventId(UUID.randomUUID())
24                 .timestamp(LocalDateTime.now())
25                 .orderId(event.getOrderId())
26                 .productId(productId)
27                 .reason(reason)
28                 .build();
29       outboxTask outboxTask = outboxService
30               .createOutboxTask(productReservationFailedEvent,
31                       "SEND_PRODUCT_RESERVATION_FAILED_EVENT");
32   }
```

Listing 5.5: Beispiel mit Fehlerszenario (PRODUCT_OUT_OF_STOCK) bei Reservierung vom Product

Der Codeausschnitt 5.5 oben aus dem `product-catalog` Service demonstriert ein weiteres Beispiel für die Behandlung einer Fehlersituation. In so einem Fall, wenn der Vorrat nicht mehr reicht, tritt ein sog. fachlicher Fehler in Saga auf. An dieser Stelle ist ein Wiederholungsversuch nicht sinnvoll, und es gibt keine bis kaum Garantie, dass „Forward Recovery“ bzw. Retry uns an dieser Stelle weiterhelfen wird. Daher brechen wir die Saga ab, publizieren `ProductReservationFailedEvent` und wollen damit indirekt eine kompensierende Transaktion in einem anderen Service auslösen (siehe Consumer dafür in Anhang C.1). So könnten Services, die auf dieses Event lauschen, nach dessen Erhalt entsprechende kompensierende Logik ihrerseits ausführen, um die Auswirkungen der zuvor comitteten lokalen Transaktion zu korrigieren. In unserem Fall mit `ProductReservationFailedEvent` wäre das relativ einfach: `order-management` Service soll nur den Bestellstatus entsprechend intern anpassen und Kunden von der Situation benachrichtigen.

Es ist jedoch wichtig zu erwähnen, dass unsere Domain-Events nicht unmittelbar an Kafka gesendet werden, sondern zunächst in der Outbox-Tabelle gespeichert sind. Der Versand erfolgt zu einem späteren Zeitpunkt durch einen Scheduler. Wir werden im Anschluss auf diesen Aspekt genauer eingehen, da er für die Implementierung des Outbox-Musters und die Lösung des Dual-Write-Problems von entscheidender Bedeutung ist.

Outbox

Das Dual-Write-Problem wurde im theoretischen Teil nicht behandelt. Wir sind der Meinung, dass dieses Problem erst im Kontext der Implementierung behandelt werden sollte. Das Dual-Write-Problem tritt grundsätzlich auf, wenn wir versuchen, externe Systeme innerhalb einer lokalen ACID-Transaktion zu aktualisieren. Um dieses Problem zu lösen, haben wir Outbox-Muster implementiert. Wenn ein Service seinen Teil eines Workflows abgeschlossen hat, speichern wir die Ergebnisse der Verarbeitung innerhalb einer lokalen Transaktion und speichern OutboxTask (siehe Listing 5.6) in einer separaten Outbox-Tabelle. Diese Tasks werden zum einen zu einem späteren Zeitpunkt von einem Scheduler (siehe Listing 5.7) abgerufen und an Kafka gesendet.

```java
@Entity
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@EntityListeners(AuditingEntityListener.class)
public class OutboxTask {
    @Id
    private UUID id;
    @CreatedDate
    private Instant createdAt;
    @LastModifiedDate
    private Instant updatedAt;
    @Version
    private Integer version;
    @Column(columnDefinition = "jsonb")
    @ColumnTransformer(write = "?::jsonb")
    private String payload;
    private String type;
    private String status;
    private Instant retryTime;

}
```

Listing 5.6: Beispiel für OutboxTask

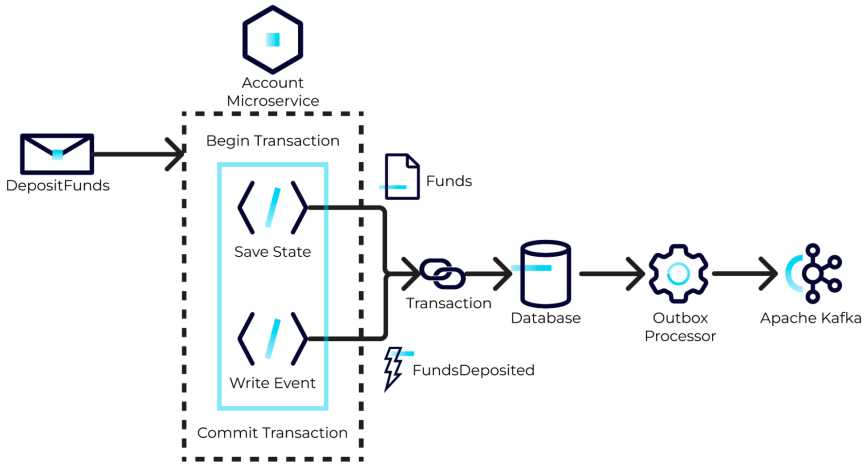


Abbildung 5.5: Outbox Pattern. Quelle [34].

```java
@Slf4j
@RequiredArgsConstructor
@Component
public class OutboxTaskScheduler {


    private final OutboxTaskService outboxService;
    private final Map<String, OutboxTaskProcessor> outboxTaskProcessorMap;

    @Value("${outbox.fixedRate}")
    private String fixedRate;

    @Value("${outbox.initialDelay}")
    private String initialDelay;

    @PostConstruct
    public void validateProcessors() {
        if (outboxTaskProcessorMap.isEmpty()) {
            throw new IllegalStateException("No OutboxTaskProcessor beans found. " +
                "Please define at least one OutboxTaskProcessor in the application.");
        }
        log.info("OutboxTaskScheduler initialized with processors: {}",
            outboxTaskProcessorMap.keySet());
    }

    @Scheduled(fixedRateString = "#{T(java.lang.Integer).parseInt('${outbox.fixedRate}')}",
            initialDelayString = "#{T(java.lang.Integer).parseInt('${outbox.initialDelay}')}")

    public void executeOutboxTask(){

        for(Map.Entry<String, OutboxTaskProcessor> entry:
            outboxTaskProcessorMap.entrySet()){

            String type = entry.getValue().getTaskType();
            OutboxTaskProcessor taskProcessor = entry.getValue();

            List<OutboxTask> tasksForProcessing = outboxService.
                getOutboxTasksForProcessing(type);
            taskProcessor.processOutboxTasks(outboxTasksForProcessing);
        }
    }
}
```

Listing 5.7: Scheduler für OutboxTasks

In unserem Prototyp haben wir das Outbox-Pattern selbst implementiert. Das Outbox selbst ist kein Bestandteil des Sagas, sondern wirkt als rein technische Zwischenschicht zwischen Datenbank und unserem Broker. Während Saga sich um die fachliche Orchestrierung bzw. Kompensation kümmert, sorgt die Outbox dafür, dass keine lokalen Änderungen verloren gehen. Das bedeutet, dass die Domänenänderung und das darauf folgende Event in derselben lokalen Transaktion persistiert werden. Ein separater Publisher liest die Outbox-Tabelle regelmäßig aus und schickt Events demnächst an Kafka. Damit bleibt fachliche Konsistenz gewahrt und jede Änderung verlässt garantiert den Service später als Nachricht.

Um Code-Redundanz zu vermeiden, haben wir das Outbox-Pattern als eigenes Spring-Boot-Starter Modul verpackt. Ein Service, der Outbox-Funktionalität braucht, fügt einfach die Abhängigkeit in seiner pom.xml hinzu – alle dazugehörigen Beans werden dann automatisch eingebunden:

```
<!-- pom.xml -->
<dependency>
    <groupId>com.acme</groupId>
    <artifactId>outbox-spring-boot-starter</artifactId>
    <version>${project.version}</version>
</dependency>
```

Listing 5.8: Outbox Dependency

Die einzige manuelle Aufgabe, die erforderlich ist, besteht darin, einen OutboxTaskProcessor für jeden Event-Typ zu registrieren, damit klar ist, welche Events zu welchen Topics zuzuordnen sind:

```
@Bean
public OutboxTaskProcessor orderCreatedTaskProcessor() {
        return new OutboxTaskGenericProcessor<OrderCreatedEvent>(
                        "SEND_ORDER_CREATED_EVENT", // Task type
                        "orders.order.created.topic",
                        OrderCreatedEvent.class
        );
}
```

Listing 5.9: Beispiel für OutboxTaskProcessor Bean aus OrderServiceConfig

Somit ist eine Konfiguration von Bean nur an einer Stelle erforderlich, während der Rest durch das Outbox-Modul zentral gesteuert wird. So bleibt die Fachlogik schlank, und

wir garantieren trotzdem, dass relevante Domänenänderungen letztendlich als Events in den gewünschten Kafka-Topics landen. Es besteht durchaus die Möglichkeit, auf Outbox zu verzichten und Nachrichten direkt an Kafka zu senden, ohne eine Zwischenschicht einzubeziehen.

Alternativ könnte man es auch mithilfe von Debezium, Kafka Connect und Kafka Streams lösen, dennoch wollten wir aber in unserem Prototyp volle Kontrolle über spezifische Anforderungen (sei es idempotenter Producer oder Zustellgarantien) und über den Code für uns selbst behalten. Ferner lassen sich dadurch Debugging und Testen einfacher handhaben.

Distributed Scheduler

Die Implementierung des Schedulers (siehe Listing 5.7) in einer verteilten Umgebung erfordert die Akzeptanz der Tatsache, dass der Scheduled Job auf jeder Instanz eines entsprechenden Services ausgeführt wird.

Die Implementierung des Schedulers (siehe Listing 5.7) in einer verteilten Umgebung erfordert die Akzeptanz der Tatsache, dass der Scheduled Job auf jeder Instanz eines entsprechenden Services laufen wird, d. h., falls wir 3 Instanzen vom `Order-Management-Service` starten, laufen im Hintergrund 3 parallele Scheduled Jobs, die OutboxTasks für `Order-Management-Service` verarbeiten sollen. Dabei sollen wir auch darauf vorbereitet werden, dass Instanzen ausfallen können, und an deren Stelle werden neue gestartet.

Einige Ideen für die Implementierung von Scheduler in so einem verteilten System wurden dem Talk auf Spring I/O 2024 [19] entnommen. Dazu zählen solche wichtigen Überlegungen wie:

1. Die Konsistenz unserer Daten wird durch das Transaktionsmanagement in Spring Boot sichergestellt. Unsere Geschäftslogik für die Verarbeitung von Aufgaben wird in Transaktionen verpackt.
2. Wir geben lediglich eine begrenzte Anzahl von Tasks zur Verarbeitung bei jedem Durchlauf von Scheduler zurück.

[19] https://www.youtube.com/watch?v=ghpljMg8Ecc

3. Wir haben einen Sperrmechanismus eingeführt (pessimistic locking), der eine Sperre auf Zeilenebene bei Abfrage setzt, und ein `@QueryHint` [20] wird benutzt, um gesperrte Einträge zu überspringen (`select for update skip locked` [21]).

[20] https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html
[21] https://www.postgresql.org/docs/current/sql-select.html

6 Evaluation und Diskussion

Im Rahmen dieser Arbeit haben wir einen funktionsfähigen Backend-Prototyp für einen C2C – Marktplatz entworfen und implementiert. Sämtliche Schritte fanden in der lokalen Entwicklungsumgebung statt. Als Koordinierungsmechanismus haben wir Choreografie – Saga implementiert. Dazu kommen noch geeignete Strategien zur Fehlerbehandlung.

Um zu prüfen, ob das System funktioniert, wurden alle Services in Docker-Containern lokal gestartet. Die Zahlungsabwicklung wird mit der Stripe-API im Testmodus simuliert.

Dabei sollten folgende Punkte beachtet werden:

- Kaum Latenzen, da alle Services auf demselben Host im Docker-Netzwerk laufen.
- Kaum bis keine Netzwerkstörungen, da keine Ausfälle oder Paketverluste

6.1 Ergebnisse und Beobachtungen

6.1.1 Systemstabilität und Leistung

Um Systemstabilität, Reaktionsfähigkeit und Zuverlässigkeit zu bewerten, haben wir unterschiedliche Lasttestsimulationen mit Gatling [1] durchgeführt. Dabei ging es primär um 2 Setups: eines mit einem konstanten Load (mit 10, 15 und 50 Requests Per Second (RPS)) und eines, bei dem die Last stufenweise von 5 auf 50 RPS zugenommen hat. Die Simulationsdauer war auf 60 Sekunden für alle Setups begrenzt.

Es lässt sich an dieser Seite erklären, dass unsere Rate-Limiter-Konfigurationen die Ergebnisse in gewisser Weise beeinflussen, indem man von einer IP-Adresse nur ca. 400 Anfragen senden kann (100 Burst Capacity plus jede Sekunde werden 5 Tokens zum

[1] https://gatling.io

Bucket hinzugefügt). Da unsere Simulationen lokal durchgeführt werden und alle Anfragen die gleiche IP-Adresse haben, betrachtet unser System dies restriktiv und lässt nur eine begrenzte Anzahl an Anfragen durch. Daher müssen wir diese Einschränkung bei Betrachtung der Ergebnisse im Hinterkopf behalten.

Die Ergebnisse haben gezeigt, dass sowohl bei konstanten Lastszenarien (bzw. 10, 15 oder 50 RPS) als auch beim Ramp-Setup die Verarbeitung von Anfragen stabil blieb. Das System verarbeitet immer eine fest definierte Anzahl von „erfolgreichen“ Anfragen (etwa 400 pro Testdurchlauf). Das zeigt, dass die Teile, aus denen das System besteht, zuverlässig arbeiten. Die konsistente und stabile Leistung wird trotz der Anzahl der parallel aktiven Benutzer gewährleistet. Dadurch können wir die Zuverlässigkeit des Systems beobachten.

Reaktionsfähigkeit

Die Antwortzeiten der KO-Anfragen sind sehr positiv zu bewerten (definitiv unter 1000 ms im Durchschnitt für alle Setups), Fehlerantworten mit Statuscode 429 kommen sehr schnell, sonst treten keine weiteren serverseitigen Fehler auf. Genaue Zahlen kann man dem Anhang G entnehmen. Dennoch soll man diese zeitmessungsbezogenen Ergebnisse etwas kritisch betrachten, da unser System während des Lasttestings in einer lokalen Umgebung auf einem einzelnen Knoten lief. Daher war die Latenz quasi 0, was aber in einer realen Umgebung, in der Knoten über das Netzwerk kommunizieren, nicht möglich wäre.

Wenn die Last (insb. im Ramp-Setup mit einem stufenweisen Anstieg von 5 auf 50 RPS innerhalb von 60 Sekunden) steigt, werden mehr Anfragen abgelehnt, aber die „erfolgreichen“ Anfragen werden dadurch nicht langsamer. Das zeigt, dass die zugrunde liegende Logik und die Ressourcen des Services für die erlaubten Anfragen ausreichend dimensioniert sind. Das System „überlebt“ also die hohe Last, indem es die zusätzlichen Anfragen konsequent ablehnt. Die Zahl der erfolgreichen Requests bleibt aber auch in diesem Setup stabil (etwa 400 OK). Das gilt unabhängig von der erhöhten Last. Man kann zwar feststellen, dass die Erfolgsquote in Szenarien mit dynamischer Belastung niedrig wird. Das liegt aber daran, dass der Rate Limiter sehr restriktiv arbeitet. Nur wenige Anfragen werden angenommen, die meisten (ca. 88%) werden mit einem 429 Statuscode abgelehnt. Deshalb liegt alles im bewusst restriktiven Rahmen.

Das Gute daran ist, dass das System bei jedem Setup den erlaubten Schwellenwert von 400 OK-Anfragen einhält und die durchschnittlichen Antwortzeiten niedrig sind. Es gibt aber einzelne Fälle, in denen es viel länger dauert (3.000 bis 4.500 ms), aber es geht eher um Ausreißer.

Einen genauen Überblick über alle Ergebnisse von allen Setups kann man dem Anhang G entnehmen. Die hohe Anzahl an KO-Zahlen ist dabei nicht überraschend, sondern beruht auf der spezifischen Konfiguration des Rate Limiters.

6.1.2 Erweiterbarkeit & Wartbarkeit

Die Microservice-Architektur sorgt in unserem Prototyp für eine klare Trennung der Verantwortlichkeiten zwischen den einzelnen Services. Jeder Service ist mit einer eindeutig definierten Domain und Verantwortung gekennzeichnet.

Die Implementierung von unterschiedlichen Entwurfsmustern und die Nutzung von Spring-Bibliotheken haben uns dabei geholfen, Boilerplate-Code zu reduzieren. Die Nutzung von Spring trägt zudem zur Wartbarkeit bei, da das Framework eine umfassende Unterstützung durch die Community bietet. Zusätzlich erleichtern Spring Boot und Dependency Injection das Testen und Anpassen von Komponenten.

Die Geschäftsabläufe, die mit Saga modelliert und implementiert wurden, sind exzellent anpassungsfähig. Es ist möglich, neue Schritte zu bereits existierenden SAGAs hinzuzufügen. Da die Kommunikation intern asynchron abläuft, können neue Services vorhandene Topics abonnieren und die erhaltenen Nachrichten verarbeiten. Es wäre durchaus möglich, die Funktionalität des bereits vorhandenen Servers für die Interaktion mit den neuen Services zu erweitern. Zudem sorgt die asynchrone Kommunikation zwischen den unabhängigen Services für eine lose Kopplung.

Die entworfene und umgesetzte Architektur erlaubt das Hinzufügen neuer Services, ohne bestehende zu modifizieren. Vielmehr kann ein bereits bestehender Service entfernt und durch einen neuen ersetzt werden. Dabei ist es immer wichtig, die klare Trennung der Verantwortlichkeiten zwischen verschiedenen Services und die Idee von einer hohen Kohäsion im Hinterkopf zu behalten und weiter zu pflegen.

Um die Wartbarkeit zu verbessern, könnte es im nächsten Schritt sinnvoll sein, sämtliche Konfigurationen auszulagern. Es wäre eigentlich ziemlich unkompliziert, dies in Spring

Boot zu verwenden, indem man Spring Cloud Config [2] nutzt. Es wäre auch ratsam, die zentrale Protokollierung und Überwachung zu integrieren, um das Systemverhalten besser zu überwachen und Fehler rechtzeitig zu erkennen.

6.1.3 Zuverlässigkeit

Um die Resilienz des Systems zu bewerten, haben wir uns an solchen ISO/IEC 25010 [3] Qualitätsmerkmalen orientiert, wie Verfügbarkeit, Fehlertoleranz und Wiederherstellbarkeit.

Verfügbarkeit wird in unserem System dadurch erreicht, dass sämtliche Services völlig zustandslos konzipiert sind: Mithilfe eines Tools kann man Instanzen bei Bedarf ohne Probleme skalieren. Die Kommunikation zwischen den Services läuft asynchron über ein mehrfach replizierbares Kafka-Cluster: Fällt ein Broker aus, übernimmt ein Replika die Partition, ohne dass Nachrichten verlorengehen.

Fehlertoleranz entsteht durch Verwendung der unterschiedlichen Stabilitätsmuster. Circuit Breaker kappen externe Abhängigkeiten, bevor lange oder komplett abgestürzte Services das Gesamtsystem blockieren, während Retry und Rate Limiter kurzfristige Netzwerkprobleme oder Lastspitzen behandeln. Ferner werden fachliche Fehler durch Saga behandelt: Jeder Schritt in Saga kennt seine kompensierende Transaktion, sodass globale Transaktionen konsistent bleiben, selbst wenn mittendrin etwas scheitert. Zusätzlich sorgt das Outbox-Pattern dafür, dass Commit zu Datenbank und Publikation in Kafka atomar ablaufen; idempotente Producer und Consumer sichern die „exactly-once"-Semantik ab, damit weder Duplikate noch uncommittete Nachrichten vorkommen. Tritt bei der Verarbeitung auf Consumer-Seite dennoch ein gravierender Fehler auf, so dass die Nachricht gar nicht auf Consumer-Seite verarbeitet werden kann, wird die betroffene Nachricht automatisch in Dead-Letter-Topic (DLT) verschoben. Diese DLT ist selbst repliziert und ermöglicht ihre spätere, kontrollierte Neuverarbeitung – ohne den Traffic zu blockieren.

Wiederherstellbarkeit ist schließlich fast ein Nebeneffekt der Microservice-Architektur: Stürzt ein Service ab, startet er neu, liest dank Kafka-Offset-Management exakt dort weiter, wo er aufgehört hat, und holt verpasste Events automatisch nach (wenn die Instanz Teil einer Consumer-Gruppe war, dann erfolgt das alles natürlich mit Rebalancing

[2] https://docs.spring.io/spring-cloud-config/docs/current/reference/html/
[3] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

von Consumer in Consumer-Group durch Kafka). Sollte eine Saga dennoch in einen widersprüchlichen Zustand geraten, löst die hinterlegte Kompensation den Konflikt auf, ohne dass ein manueller Eingriff nötig wäre. In dem Sinne arbeitet Saga eng mit Kafka zusammen und verlässt sich auf Mechanismen, die uns Kafka bietet (i. e. Offset-Management, Retry und DLT). Da alle Ereignisse unveränderlich in Kafka vorliegen, lassen sich vergangene Systemzustände bei Bedarf durch simples Replay rekonstruieren.

Die Verfügbarkeit, Fehlertoleranz und Wiederherstellbarkeit werden durch zustandslose Services, Kafka und gezielte Stabilitätsmuster nicht nur dokumentiert, sondern auch während des Betriebs deutlich gemacht.

Ferner wäre es auch ratsam, die zentrale Protokollierung und Überwachung zu integrieren, um das Systemverhalten besser zu überwachen und Fehler rechtzeitig zu erkennen.

Zusammenfassung

Die Evaluation hat gezeigt, dass der Prototyp lokal stabil und leistungsstark läuft. Der asynchrone Ansatz ermöglicht es, unmittelbar nach Ende der Saga ein Feedback zu erhalten, ohne auf das vollständige Ende der Transaktion warten zu müssen. Dies erhöht die Verfügbarkeit des Systems.

Saga erhöht zwar die Verfügbarkeit, führt jedoch zu potenziellen Inkonsistenzen, indem die Ergebnisse von committeten lokalen Subtransaktionen für andere parallel laufende Sagas sichtbar sind. Solange die Saga-Transaktion nicht abgeschlossen ist, kann das System inkonsistente Daten aufweisen. In einem solchen System erfordert dies die Inkaufnahme von „eventual consistency“. Die Fehlerbehandlung und kompensierende Transaktionen im Hintergrund erfordern dabei eine hohe Aufmerksamkeit der Entwickler.

Die Erweiterung des Service-Spektrums bedeutet in der Regel auch eine Steigerung der Komplexität. Dies kann im schlimmsten Fall dazu führen, dass die Übersicht über das Gesamtsystem verloren geht. Um dies zu verhindern, wird im weiteren Verlauf eine vernünftige Einführung von Logging, Tracking und Monitoring empfohlen.

Ferner zeigen die Ergebnisse der Lasttests, dass das System stabil und vorhersagbar unter dem Druck arbeitet. Wenn mehr Durchsatz benötigt wird, muss man in den Konfigurationseinstellungen mal genauer überlegen.

Die Evaluation zeigt auch, dass das gewählte Saga-Muster und die Resilienz-Strategien ihre Aufgaben erfüllen. Um jedoch aussagekräftige Ergebnisse bezüglich der Leistung, der Skalierbarkeit und der Ausfallsicherheit zu erhalten, sind weitere Tests in einer produktionsnahen Umgebung erforderlich. Als nächster Schritt könnte das Deployment auf Kubernetes mit simulierten Netzwerkausfällen erfolgen.

7 Fazit und Ausblick

Im theoretischen Teil haben wir uns zunächst mit den Unterschieden zwischen Microservices-Architekturen und dem monolithischen Ansatz auseinandergesetzt. Im weiteren Verlauf wurde auf die besonderen Anforderungen an Transaktionen in Microservices eingegangen. Microservices bieten zwar zahlreiche Vorteile, wie verbesserte Skalierbarkeit, Flexibilität und einfaches Deployment, bringen aber auch Herausforderungen mit sich, insbesondere im Hinblick auf das Transaktionsmanagement und die Datenkonsistenz. Da die Daten in unserem System über mehrere Knoten verteilt sind, erfordert die Sicherstellung der Konsistenz besondere Sorgfalt. Es wurden zwei passende Ansätze für die Umsetzung der Transaktionen in unserem System behandelt: 2PC und Saga. 2PC stellt hohe Anforderungen an die Koordination und Verfügbarkeit der Teilnehmer im System, bietet dafür aber stärkere Konsistenzgarantien. Das Saga hingegen setzt auf asynchrone Kommunikation und „Eventual Consistency“, was in der Regel keine strenge Koordination und keine permanente Verfügbarkeit aller an der Transaktion beteiligten Services erfordert. Auch auf einige Aspekte der Resilienz, die in verteilten Systemen von Bedeutung sind, um Ausfällen oder Störungen effektiv entgegenzuwirken, wurde eingegangen.

Zudem wurde ein funktionierender Backend-Prototyp mit Saga-Muster entworfen und implementiert. Mit Saga könnten wir serviceübergreifende Transaktionen umsetzen, indem wir eine solche globale Transaktion in kleinere, überschaubare und „ACID-konforme“ lokale Transaktionen aufteilen, die jeweils von einem einzelnen Microservice ausgeführt werden. Es ist jedoch zu beachten, dass bei fehlender Isolation zwischen den Sagas Anomalien im Gesamtsystem auftreten können. Dies bedeutet aber auch, dass das Monitoring und die Fehleranalyse mit zunehmender Komplexität der Sagas deutlich anspruchsvoller werden.

Saga gewährleistet eine schnelle Rückmeldung an die Benutzer, wobei alle Schritte der globalen Transaktionen asynchron verarbeitet werden, was die Nutzer jedoch nicht belastet, da die Arbeit intern vom System erledigt wird.

Unsere Tests in einer lokalen Umgebung validieren zwar, dass das System grundsätzlich funktioniert, erlauben aber keine umfassende Aussage über dessen Verhalten in einer produktionsnahen Umgebung. Da die Forderung nach starker Konsistenz nur in Ausnahmefällen gerechtfertigt ist, können wir generell die Idee von „eventual consistency“ akzeptieren und Saga zur Modellierung des Transaktionsverhaltens in so einem System von Microservices verwenden.

Die Erkenntnisse zeigen auch, dass die Umsetzung solcher verteilter Transaktionen nicht nur ein fundiertes Verständnis der Fachlogik erfordert. Sie erfordert auch einen erheblichen technischen und betrieblichen Aufwand, insbesondere im Hinblick auf Monitoring, Logging und Integration.

Ausblick

Als nächster Schritt kann man überlegen, umfassende Logging- und Tracing-Lösungen zu integrieren. Diese bieten nicht nur eine detailliertere Nachvollziehbarkeit des Transaktionsverlaufs, sondern helfen auch bei der Überwachung des Gesamtsystems auf auffälliges Verhalten. Ein weiterer Vorschlag wäre auch, Chaos-Engineering im Projekt zu integrieren, um die Stabilität des Systems genauer zu untersuchen. Ferner kann man auch die Möglichkeit überlegen, das Gesamtsystem in einer produktionsnahen Infrastruktur zu deployen, beispielsweise in Kubernetes in der Cloud, um Skalierbarkeit und Fehlertoleranz unter realen Bedingungen zu testen.

In Bezug auf Saga und dessen Implementierung wäre es durchaus denkbar, mit verschiedenen Frameworks wie beispielsweise Camunda, Spring State Machine oder Axon Framework zu experimentieren. Es ist aber zu beachten, dass die meisten Werkzeuge jedoch für Orchestrierung – Saga geeignet sind, da diese spezifische Orchestrierungsmechanismen genau für diese Art bereitstellen. Dennoch bieten sie sich als eine gültige Alternative zur Eigenimplementierung an.

Literaturverzeichnis

[1] Alonso, G. ; Agrawal, D. ; El Abbadi, A. ; Kamath, M. ; Gunthor, R. ; Mohan, C.: Advanced transaction models in workflow contexts. In: *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, S. 574–581

[2] Berenson, Hal ; Bernstein, Phil ; Gray, Jim ; Melton, Jim ; O'Neil, Elizabeth ; O'Neil, Patrick: A critique of ANSI SQL isolation levels. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : Association for Computing Machinery, 1995 (SIGMOD '95), S. 1–10. – URL https://doi.org/10.1145/223784.223785. – ISBN 0897917316

[3] Brewer, Eric: CAP Twelve years later: How the "Rules"have Changed. In: *Computer* 45 (2012), 02, S. 23–29

[4] Brewer, Eric: *NoSQL: Past, Present, Future*. InfoQ. 2012. – URL https://www.infoq.com/presentations/NoSQL-History/?ref=highscalability.com. – Zugriffsdatum: 27.02.2025

[5] Colombo, Christian ; Pace, Gordon: Recovery within Long-Running Transactions. In: *ACM Computing Surveys (CSUR)* 45 (2013), 06

[6] Colyer, Adrian: *A Critique of ANSI SQL Isolation Levels*. 2016. – URL https://web.archive.org/web/20250102080423/https://blog.acolyer.org/2016/02/24/a-critique-of-ansi-sql-isolation-levels/. – Zugriffsdatum: 03.03.2025

[7] Frank, Lars ; Zahle, Torben U.: Semantic ACID properties in multidatabases using remote procedure calls and update propagations. In: *Softw. Pract. Exper.* 28 (1998), Januar, Nr. 1, S. 77–98. – ISSN 0038-0644

[8] Friedrichsen, Uwe: *The limits of the Saga pattern: Why it is not a replacement for distributed transactions.* – URL https://www.ufried.com/blog/limits_of_saga_pattern/. – Zugriffsdatum: 10.02.2025

[9] Friedrichsen, Uwe: *Resilience: A New Paradigm for the 21st Century.* – URL https://www.ufried.com/blog/resilience/. – Zugriffsdatum: 10.02.2025

[10] Friedrichsen, Uwe: *Resilience vs. Fault Tolerance.* 2022. – URL https://www.ufried.com/blog/resilience_vs_fault_tolerance/. – Zugriffsdatum: 03.03.2025

[11] Friedrichsen, Uwe: *The long way towards resilience - Part 4: Availability revisited.* Oktober 2024. – URL https://www.ufried.com/blog/road_to_resilience_4/. – Zugriffsdatum: 03.03.2025

[12] Garcia-Molina, Hector ; Salem, Kenneth: Sagas. In: *ACM Sigmod Record* 16 (1987), Nr. 3, S. 249–259. – URL https://dl.acm.org/doi/pdf/10.1145/38714.38742. – ISSN 0163-5808

[13] Hanmer, Robert S.: *Patterns for Fault Tolerant Systems.* Hoboken, NJ, USA : Wiley, 2007. – ISBN 978-0-470-31979-6

[14] Härder, Theo ; Reuter, Andreas: Principles of Transaction-Oriented Database Recovery. In: *ACM Computing Surveys* 15 (1983), Nr. 4, S. 287–317. – URL https://cs-people.bu.edu/mathan/reading-groups/papers-classics/recovery.pdf. – Zugriffsdatum: 15.12.2025

[15] Kleppmann, Martin: *Datenintensive Anwendungen designen: Konzepte für zuverlässige, skalierbare und wartbare Systeme.* Heidelberg : O'Reilly, 2018 (Animals). – URL http://nbn-resolving.org/urn:nbn:de:bsz:31-epflicht-1301073. – ISBN 3-96009-075-7

[16] Kumar S, Arun ; Gayathri ; S, Anusha M. ; Hegde, Uttam U. ; Kumar G R, Vinay ; Pasha, N N.: Resilience and Fault Tolerance in Cloud Computing. In: *International Journal of Innovative Research in Technology* 11 (2024), Nr. 7, S. 1478–1483. – URL https://ijirt.org/Article?manuscript=170735. – ISSN 2349-6002

[17] Laigner, Rodrigo ; Zhou, Yongluan ; Vaz Salles, Marcos A. ; Liu, Yijian ; Kalinowski, Marcos: *Data Management in Microservices: State of the Practice, Challenges, and Research Directions.* 02 2021

[18] Mailewa, Akalanka ; Akuthota, Arunkumar ; Dissanayake Mohottalalage, Thivanka: A Review of Resilience Testing in Microservices Architectures: Implementing Chaos Engineering for Fault Tolerance and System Reliability, 01 2025

[19] Messina, Antonio ; Rizzo, Riccardo ; Storniolo, Pietro ; Tripiciano, Mario ; Urso, Alfonso: The Database-is-the-Service Pattern for Microservice Architectures, 09 2016, S. 223–233. – ISBN 978-3-319-43948-8

[20] Microsoft: *Saga Pattern in Azure Architecture*. 2024. – URL https://learn.microsoft.com/de-de/azure/architecture/reference-architectures/saga/saga. – Zugriffsdatum: 13.12.2024

[21] Miranda, Fábio de S. ; Santos, Daniel S. dos ; Vilela, Ricardo F. ; Assunção, Wesley Klewerton G. ; Santos, Reginaldo C. dos ; Pinto, Victor Hugo Santiago C.: A proposed catalog of development patterns for fault-tolerant microservices. New York, NY, USA : Association for Computing Machinery, 2024 (SBQS '24). – URL https://doi.org/10.1145/3701625.3701678. – ISBN 9798400717772

[22] Newman, Sam: *Monolith to Microservices*. First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., 2019

[23] Newman, Sam: *Building microservices: Designing fine-grained systems*. Second edition. Beijing [i pozostałe] : O'Reilly, opyright 2021. – ISBN 1492034029

[24] Pritchett, Dan: BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. 6 (2008), Mai, Nr. 3, S. 48–55. – URL https://doi.org/10.1145/1394127.1394128. – ISSN 1542-7730

[25] Red Hat: *What is High Availability?* 2025. – URL https://www.redhat.com/de/topics/linux/what-is-high-availability. – Zugriffsdatum: 15.03.2025

[26] Richardson, Chris: *Microservices Patterns: With examples in Java*. 1st edition. Manning Publications, 2018. – ISBN 9781617294549

[27] Rotem-Gal-Oz, Arnon: Fallacies of Distributed Computing Explained. In: *Doctor Dobbs Journal* (2008), 01. – URL https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained. – Zugriffsdatum: 13.12.2025

[28] Silberschatz, Abraham ; Mehrotra, Sharad: A Transaction Model for Multidatabase Systems. In: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, IEEE, June 1992. – URL https://doi.org/10.1109/ICDCS.1992.235055

[29] Starke, Gernot: *Effektive Software-Architekturen: Ein praktischer Leitfaden*. München : Carl Hanser Verlag, 2020. – ISBN 978-3-446-46589-3

[30] Steen, Maarten van ; Tanenbaum, Andrew S.: *Distributed Systems*. 3rd. Self-published, 2020. – Version 3.03, 2020

[31] Štefanko, Martin ; Chaloupka, Ondřej ; Rossi, Bruno: The Saga Pattern in a Reactive Microservices Environment. In: *Proceedings of the 14th International Conference on Software Technologies*, SCITEPRESS - Science and Technology Publications, 2019, S. 483–490. – ISBN 978-989-758-379-7

[32] Taibi, Davide ; Lenarduzzi, Valentina ; Pahl, Claus: Microservices Anti-patterns: A Taxonomy. In: *Microservices*. Springer International Publishing, 2020, S. 111–128. – URL https://arxiv.org/pdf/1908.04101

[33] Uwe Friedrichsen: *The limits of the Saga pattern: Why it is not a replacement for distributed transactions*. 2021. – URL https://www.ufried.com/blog/limits_of_saga_pattern/. – Zugriffsdatum: 3.10.2024

[34] Waldron, Wade: *Solving the Dual-Write Problem: Effective Strategies for Atomic Updates Across Systems*. 2024. – URL https://www.confluent.io/blog/dual-write-problem/. – Zugriffsdatum: 27.02.2025

[35] Wikipedia contributors: *2-Phase Commit Protocol*. 2024. – URL https://en.wikipedia.org/wiki/Two-phase_commit_protocol. – Zugriffsdatum: 15.12.2024

[36] Wolff, Eberhard: *Microservices, 2nd Edition*. 2nd edition. [Erscheinungsort nicht ermittelbar] and Sebastopol, CA : dpunkt and O'Reilly Media Inc, 2018. – ISBN 3-86490-555-9

[37] Zhou, Xiang ; Peng, Xin ; Xie, Tao ; Sun, Jun ; Ji, Chao ; Li, Wenhai ; Ding, Dan: Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. In: *IEEE Transactions on Software Engineering* 47 (2021), Nr. 2, S. 243–260

A Anhang

A.1 Use Cases

UC1:	**Produktsuche**
Akteure	Nutzer (Käufer)
Auslöser	Aufruf der Produktsuche
Aktionsschritte	1. Der Nutzer gibt einen Suchbegriff in die Suchzeile ein. 2. Der Nutzer bestätigt die Suchanfrage. 3. Das System führt eine Suche in der Datenbank durch (nach Wort, Sortierung nach Preis). 4. Das System gibt die Suchergebnisse zurück.
Ergebnis	Der Nutzer sieht eine Liste von Produkten, die der Suchanfrage entsprechen.

UC2:	**Bestellung aufgeben**
Akteure	Käufer
Auslöser	Der Käufer wählt ein Produkt aus und startet den Bestellprozess.
Aktionsschritte	1. Der Käufer wählt ein Produkt aus. 2. Der Käufer bestätigt die Bestellung. 3. Das System gibt sofort die Bestellnummer zurück und erstellt eine Bestellung im Status „Created“. 4. Das System überprüft die Verfügbarkeit des Produktes. 4.1 Falls das Produkt verfügbar ist, wird es reserviert. 4.2 Falls das Produkt nicht verfügbar ist, wird die Bestellung storniert und der Käufer wird davon entsprechend benachrichtigt. 5. Das System stellt dem Käufer nach der erfolgreichen Überprüfung des Bestandes einen personalisierten Zahlungslink zur Verfügung, der 30 Minuten gültig ist.
Ergebnis	Die Bestellung wurde erstellt, und das System wartet 30 Minuten auf die Zahlung.

UC3:	**Zahlung einer Bestellung**
Akteure	Käufer
Auslöser	Der Käufer klickt den Zahlungslink für die Bestellung.
Aktionsschritte	1. Käufer fragt einen personalisierten Zahlungslink ab (Session URL von Stripe). 2. Der Käufer nutzt den Zahlungslink und bezahlt Waren mit Kreditkarte innerhalb von 30 Minuten. 3. Falls die Zahlung erfolgreich ist, wird die Bestellung bestätigt. 4. Falls die Zahlung nicht innerhalb von 30 Minuten erfolgt, wird die Bestellung automatisch storniert.
Ergebnis	Die Bestellung ist entweder bezahlt oder automatisch nach 30 Minuten storniert.

UC4:	**Stornierung einer Bestellung**
Akteure	Käufer
Auslöser	Der Käufer entscheidet sich, eine bereits bezahlte Bestellung zu stornieren.
Aktionsschritte	1. Der Käufer kann die Bestellung innerhalb von 12 Stunden nach Zahlungseingang stornieren. 2. Das System verarbeitet die Rückerstattung. 3. Das System sendet Benachrichtigungen an den Käufer und den Verkäufer über die Stornierung. 4. Falls die 12-Stunden-Frist abgelaufen ist, kann die Bestellung nicht mehr storniert werden.
Ergebnis	Die Bestellung wird entweder storniert und erstattet oder bleibt gültig.

B Anhang

```java
@Component
@KafkaListener(topics = {"products.product.reserved.topic"}, containerFactory
    = "kafkaListenerContainerFactory")
@Slf4j
public class ProductReservedEventHandler {

    @Autowired
    private ProcessedEventRepository processedEventRepository;
    @Autowired
    private OrderService orderService;

    @KafkaHandler
    @Transactional
    public void handle(@Payload ProductReservedEvent event, @Header("
        messageId") String messageId,
                       @Header(KafkaHeaders.RECEIVED_KEY) String messageKey)
                           {


        ProcessedEventEntity processedEvent = processedEventRepository.
            findByMessageId(messageId);
        if (processedEvent != null) {
            log.info("Duplicate massageId detected= {}", messageId);
            return;
        }

        log.info("Handler received ProductReservedEvent= {}  ", event);

        try {

            processedEventRepository.save(ProcessedEventEntity.builder()
                    .orderId(event.orderId())
                    .messageId(messageId)
                    .build());

```

```
32              orderService.updateOderStatus(UUID.fromString(event.orderId()),
                    OrderStatus.PENDING_PRODUCT_RESERVED);
33
34          } catch (DataIntegrityViolationException ex) {
35              log.error(ex.getMessage());
36              throw new NonRetryableException(ex.getMessage());
37          }
38      }
39  }
```

Listing B.1: Beispiel Kafka-Consumer für ProductReservedEvent

C Anhang

```java
@Component
@KafkaListener(topics = {"products.product.reservation.failed.topic"},
    containerFactory = "kafkaListenerContainerFactory")
@Slf4j
public class ProductReservationFailedEventHandler {

    @Autowired
    ProcessedEventRepository processedEventRepository;
    @Autowired
    private OrderService orderService;

    @KafkaHandler
    @Transactional
    public void handle(@Payload ProductReservationFailedEvent event, @Header(
        "messageId") String messageId,
                       @Header(KafkaHeaders.RECEIVED_KEY) String messageKey)
                            {


        ProcessedEventEntity processedEvent = processedEventRepository.
            findByMessageId(messageId);

        if (processedEvent != null) {
            log.info("Duplicate massageId detected= {}", messageId);
            return;
        }

        orderService.cancelOrder(UUID.fromString(event.orderId()), event.
            reason());

        try {
            processedEventRepository.save(ProcessedEventEntity.builder()
                    .orderId(event.orderId())
                    .messageId(messageId)
                    .build());
```

```
31          } catch (DataIntegrityViolationException ex) {
32              log.error(ex.getMessage());
33              throw new NonRetryableException(ex.getMessage());
34          }
35      }
36  }
```

Listing C.1: Beispiel Kafka-Consumer für ProductReservationFailedEvent

D Anhang

```java
@Configuration
public class KafkaProducerConfig {


    @Value("${spring.kafka.bootstrap-servers}")
    private String bootStrapServer;

    @Value("${spring.kafka.producer.transaction-id-prefix}")
    private String transactionalIdPrefix;

    public Map<String, Object> producerConfig(){

        HashMap<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootStrapServer);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            JsonSerializer.class);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120_000); //2
            min
        //props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG,20000);
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG,30_000);
        //if set to 0ms, we are not collecting messages in batches, we do not
             retain messages to be sent
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,
            transactionalIdPrefix);
        return props;
    }

    @Bean
    public ProducerFactory<String, Object> producerFactory(){
        return new DefaultKafkaProducerFactory<>(producerConfig());
```

```
31      }
32
33      @Bean
34      public KafkaTemplate<String, Object> kafkaTemplate(ProducerFactory<String
            , Object> producerFactory){
35          KafkaTemplate<String, Object> kafkaTemplate = new KafkaTemplate<>(
                producerFactory);
36          kafkaTemplate.setObservationEnabled(true);
37          return kafkaTemplate;
38      }
39
40      @Bean(name = "kafkaTransactionManager")
41      KafkaTransactionManager<String, Object> kafkaTransactionManager(
            ProducerFactory<String, Object> producerFactory){
42          KafkaTransactionManager<String, Object> kafkaTransactionManager = new
                 KafkaTransactionManager<>(producerFactory);
43          System.out.println("KafkaTransactionManager␣initialized:␣" +
                kafkaTransactionManager);
44          return kafkaTransactionManager;
45      }
46
47  }
```

Listing D.1: Kafka Producer Config

E Anhang

```java
@Configuration
public class KafkaConsumerConfig {

    @Value("${spring.kafka.bootstrap-servers}")
    private String bootStrapServer;
    @Autowired
    Environment environment;


    @Bean
    public ConsumerFactory<String, Object> consumerFactory(){
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootStrapServer);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class);
        //skip offset if error, consumer will throw error only unce -> no
            endless cycles
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            ErrorHandlingDeserializer.class);
        props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
            JsonDeserializer.class);
        props.put(ConsumerConfig.ENABLE_METRICS_PUSH_CONFIG, true);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, environment.getProperty("
            spring.kafka.consumer.group-id"));
        props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, environment.
            getProperty("spring.kafka.consumer.isolation-level"));

        return  new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public KafkaTelemetry kafkaTelemetry(OpenTelemetry openTelemetry) {
        return KafkaTelemetry.create(openTelemetry);
```

```
30      }
31
32
33      @Bean
34      public ConcurrentKafkaListenerContainerFactory<String, Object>
            kafkaListenerContainerFactory(
35              KafkaTemplate<String, Object> kafkaTemplate){
36
37          DefaultErrorHandler errorHandler = new DefaultErrorHandler(new
                DeadLetterPublishingRecoverer(kafkaTemplate),
38                  //kafka will try 3 times to deliver the message with fixed
                        backoff
39                  new FixedBackOff(3000, 3));
40
41          errorHandler.addNotRetryableExceptions(NonRetryableException.class);
42          errorHandler.addRetryableExceptions(RetryableExceptions.class);
43          ConcurrentKafkaListenerContainerFactory<String, Object> factory =
44                  new ConcurrentKafkaListenerContainerFactory<>();
45
46          factory.setConsumerFactory(consumerFactory());
47          factory.getContainerProperties().setObservationEnabled(true);
48          factory.setCommonErrorHandler(errorHandler);
49
50          return factory;
51      }
52  }
```

Listing E.1: Kafka Consumer Config

F Anhang

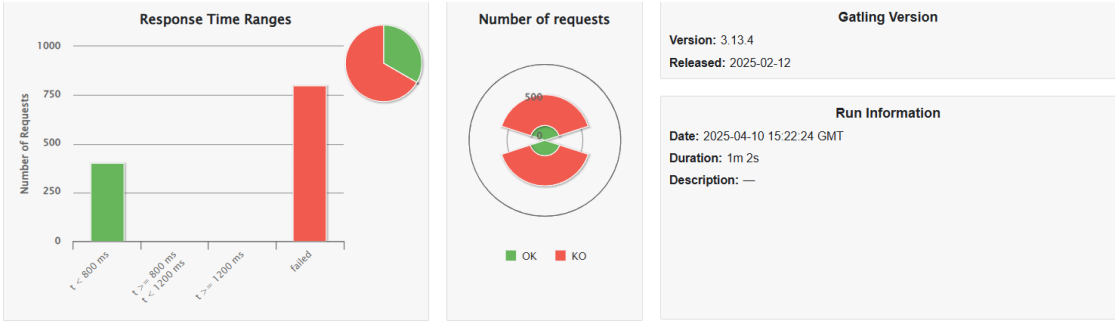
```yaml
spring:
  application:
    name: api-gateway
  cloud:
    loadbalancer:
      enabled: true
    gateway:
      routes:
        - id: order-management
          uri: lb://ORDER-MANAGEMENT
          predicates:
            - Path=/api/v1/orders/**
        - id: payment-processing
          uri: lb://PAYMENT-PROCESSING
          predicates:
            - Path=/api/v1/payments/**
        - id: product-catalog
          uri: lb://PRODUCT-CATALOG
          predicates:
            - Path=/api/v1/products/**
      httpclient:
        connect-timeout: 5000
        response-timeout: 10s
      default-filters:
        - name: CircuitBreaker
          args:
              name: orderServiceCircuitBreaker
              fallbackUri: forward:/fallback/marketplace
        - name: RequestRateLimiter
          args:
            key-resolver: "#{@ipKeyResolver}"
            redis-rate-limiter.replenishRate: 5  # Allow 5 requests per
                second
            redis-rate-limiter.burstCapacity: 100 # Allow bursts up to 100
                requests
```

```
34            redis-rate-limiter.requestedTokens: 1
35            redis-rate-limiter.ttl: 120
36        - name: Retry
37          args:
38            retries: 3                      # Retry request up to 3 times
39            statuses: INTERNAL_SERVER_ERROR, BAD_GATEWAY, SERVICE_UNAVAILABLE
                  , GATEWAY_TIMEOUT    # Retry on these HTTP errors
40            methods: POST                   # Only retry POST requests
41            backoff:
42              firstBackoff: 100ms           # First retry delay = 100ms
43              maxBackoff: 2000ms            # Max retry delay = 2s
44              factor: 2                     # Exponential backoff factor
45              basedOnPreviousValue: true    # Next delay = previous * factor
46  data:
47    redis:
48      host: localhost
49      port: 6379
50resilience4j:
51  timelimiter:
52    instances:
53      orderServiceCircuitBreaker:
54        timeoutDuration: 11s
55  circuitbreaker:
56    instances:
57      orderServiceCircuitBreaker:
58        register-health-indicator: true
59        failureRateThreshold: 50             # Open circuit if 50% of last
              requests failed
60        minimum-number-of-calls: 10          # Wait for at least 10 requests
              before deciding
61        slowCallRateThreshold: 60            # If 60% of requests are slow ->
               open
62        slowCallDurationThreshold: 6s       # Calls longer than 6s are slow
63        waitDurationInOpenState: 5s          # Keep circuit open for 5s
              before retrying
64        permittedNumberOfCallsInHalfOpenState: 5  # Allow 5 test requests
              before closing circuit
65        slidingWindowSize: 10                # Monitor last 10 requests for
              failures
```

Listing F.1: API Config

G Anhang

G.1 Ergebnisse vom Setup mit 10 RPS, die Simulationsdauer 60 Sekunden



Assertions

Assertion	Status
Global: count of successful events is between 350.0 and 400.0 inclusive	OK
Global: mean of response time is less than 1000.0	OK
Global: 95th percentile of response time is less than 2000.0	OK
Global: count of all events is greater than or equal to 600.0	OK

Expand all groups | Collapse all groups

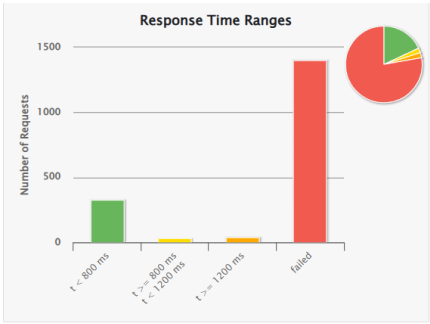
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	1,200	400	800	NaN	19.05	4	8	17	34	64	205	13	15
Create Order	600	196	404	67.33	9.52	5	8	21	43	69	205	15	16
Get Order	600	204	396	66.00	9.52	4	8	14	24	44	204	12	13

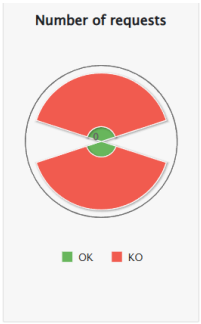
Errors

Error	Count	Percentage
status.find.is(201), but actually found 429	404	50.5%
status.find.in(200,404), but actually found 429	396	49.5%

Abbildung G.1: Setup mit 10 RPS, die Simulationsdauer 60 Sekunden

G.2 Ergebnisse vom Setup mit 15 RPS, die Simulationsdauer 60 Sekunden





Gatling Version

Version: 3.13.4
Released: 2025-02-12

Run Information

Date: 2025-04-10 17:55:18 GMT
Duration: 1m 2s
Description: —

Assertions

Assertion	Status
Global: count of successful events is between 350.0 and 400.0 inclusive	OK
Global: mean of response time is less than 1000.0	OK
Global: 95th percentile of response time is less than 2000.0	OK
Global: count of all events is greater than or equal to 600.0	OK

Expand all groups | Collapse all groups

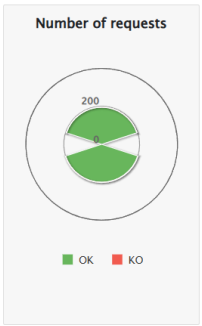
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	1,800	400	1,400	NaN	28.57	5	73	300	1,013	1,889	2,784	244	386
Get Order	900	195	705	78.33	14.29	5	70	278	935	1,994	2,784	230	374
Create Order	900	205	695	77.22	14.29	5	76	324	1,092	1,934	2,688	258	397

Errors

Error	Count	Percentage
status.find.in(200,404), but actually found 429	705	50.36%
status.find.is(201), but actually found 429	695	49.64%

Abbildung G.2: Setup mit 15 RPS, die Simulationsdauer 60 Sekunden

G.3 Ergebnisse vom Setup mit 50 RPS, die Simulationsdauer 60 Sekunden



Gatling Version

Version: 3.13.4
Released: 2025-02-12

Run Information

Date: 2025-04-10 19:10:50 GMT
Duration: 1m 2s
Description: —

Assertions

Assertion	Status
Global: count of successful events is between 350.0 and 400.0 inclusive	OK
Global: mean of response time is less than 1000.0	OK
Global: 95th percentile of response time is less than 2000.0	OK
Global: count of all events is greater than or equal to 600.0	OK

Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	6,000	395	5,605	NaN	95.24	4	7	13	440	2,093	2,768	90	335
Get Order	3,000	195	2,805	NaN	47.62	4	7	12	437	2,064	2,625	88	331
Create Order	3,000	200	2,800	NaN	47.62	4	7	13	454	2,116	2,768	91	339

Errors

Error	Count	Percentage
status.find.in(200,404), but actually found 429	2805	50.04%
status.find.is(201), but actually found 429	2800	49.96%

Abbildung G.3: Setup mit 50 RPS, die Simulationsdauer 60 Sekunden

G.4 Ergebnisse vom Setup mit stufenweiser Erhöhung der Last von 5 auf 50 RPS, die Simulationsdauer 60 Sekunden

Assertions

Assertion	Status
Global: max of response time is less than 3000.0	KO
Global: percentage of successful events is greater than 95.0	KO
Global: count of successful events is between 350.0 and 400.0 inclusive	OK
Global: 95th percentile of response time is less than 2000.0	OK
Global: mean of response time is less than 1000.0	OK

Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	3,300	400	2,900	NaN	53.23	5	16	52	412	2,601	4,560	116	404
Get Order	1,650	197	1,453	NaN	26.61	5	17	50	356	2,557	4,404	108	377
Create Order	1,650	203	1,447	NaN	26.61	5	16	55	480	2,687	4,560	125	429

Errors

Error	Count	Percentage
status.find.in(200,404), but actually found 429	1453	50.1%
status.find.is(201), but actually found 429	1447	49.9%

Abbildung G.4: Ramp von 5 auf 50 RPS, die Simulationsdauer 60 Sekunden

H Anhang

H.1 Sequenzdiagramm 1

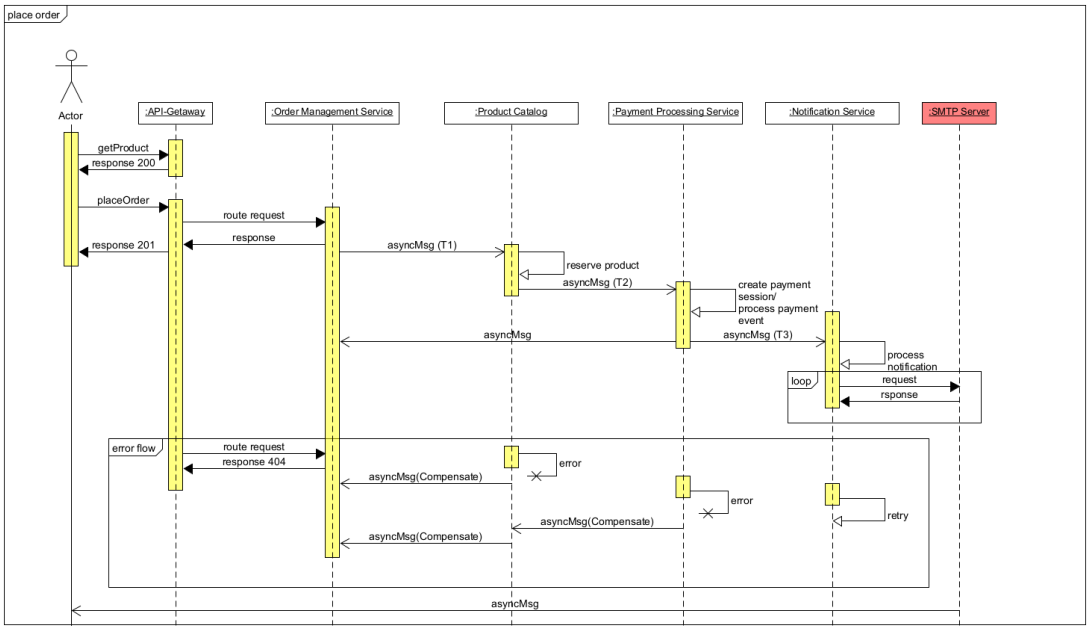


Abbildung H.1: Sequenzdiagramm zur Aufgabe der Bestellung

H.2 Sequenzdiagramm 2

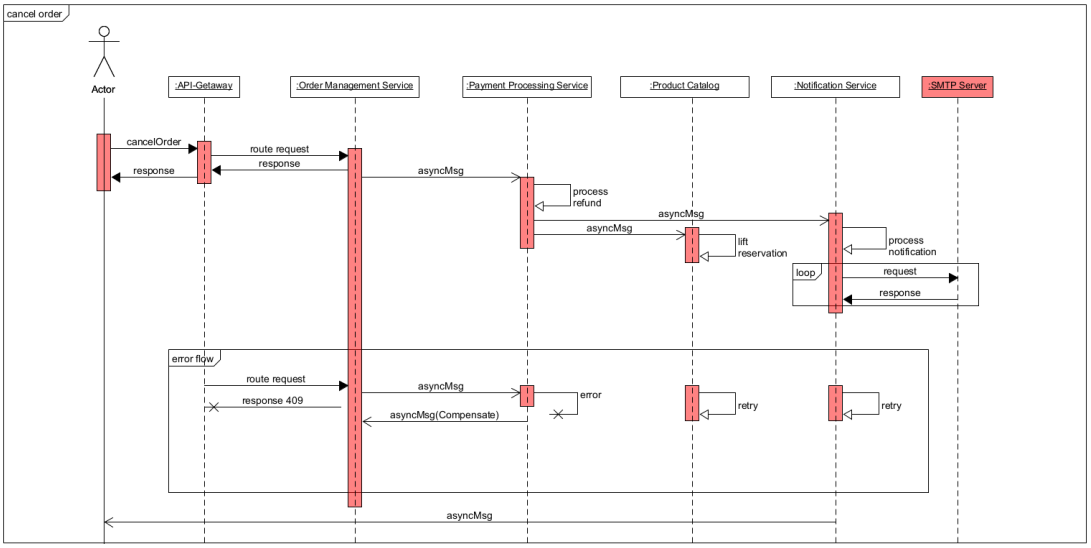


Abbildung H.2: Sequenzdiagramm zur Stornierung der Bestellung

Glossar

Debezium Ein Open-Source-Change-Data-Capture-Tool, das Datenbankänderungen in Echtzeit streamt.

Kafka Connect Eine Komponente von Apache Kafka, die zur einfachen Integration externer Systeme (z. B. Datenbanken) dient und Daten zuverlässig in und aus Kafka überträgt..

Kafka Streams Eine Stream-Verarbeitungsbibliothek, die auf Apache Kafka aufbaut und das Erstellen verteilter, fehlertoleranter Echtzeitanwendungen für Datenströme ermöglicht.

MTBF Mittlere Betriebsdauer zwischen Ausfällen (Mean Time Between Failures) – definiert als die Summe aus MTTF und MTTR.

MTTF Mittlere Zeit bis zum Ausfall (Mean Time to Failure) – bezeichnet die erwartete Betriebszeit eines Systems, bevor ein Ausfall auftritt.

MTTR Mittlere Reparaturzeit (Mean Time to Repair) – beschreibt die durchschnittliche Zeit, die benötigt wird, um ein System nach einem Ausfall wiederherzustellen.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original