Der Praxisguide für lose gekoppelte, resiliente Systeme

Event-driven Architecture in Action

von Lutz Hühnken

Events sind mehr als Nachrichten: Sie sind der Taktgeber moderner, verteilter Systeme. Dieses Whitepaper zeigt, wie Event-driven Architecture in der Praxis gelingt – vom sicheren Publizieren über verlässliche Verarbeitung bis zur Evolution im laufenden Betrieb. Für Architekturen, die robust bleiben, wenn Systeme wachsen.

- Events erzeugen: Grundlagen und sicheres Publizieren
- Events verarbeiten: verlässlicher Konsum von Events
- Events kombinieren: Zusammenarbeit mehrerer Services
- Events meistern: Betrieb und Evolution Event-getriebener Systeme

Event-driven Architecture: eine Einführung – Teil 1

Events erzeugen: Grundlagen und sicheres Publizieren

Das Interesse an Event-driven Architecture (EDA) hat in den letzten Jahren stetig zugenommen. Moderne Systeme müssen skalieren, ausfallsicher sein und trotzdem flexibel auf neue Geschäftsanforderungen reagieren können. Inwieweit kann uns dieser Ansatz dabei helfen? In diesem Whitepaper beleuchten wir das Thema aus architektonischer und praktischer Perspektive.

Wer sich mit Softwarearchitektur beschäftigt, wird früh gelernt haben, dass es nicht DIE gute und DIE schlechte Architektur gibt. Es gibt Rahmenbedingungen und Ziele, und es gibt Architekturen, die für den jeweiligen Zweck geeignet sind.

Ein heutzutage gängiges Architekturmuster sind Microservices. Große Systeme werden aufgeteilt in kleine Services, jeweils für einen Teilbereich der Fachlichkeit. Die Idee ist, dass jeder einzelne Service unabhängig von den anderen von einem eigenen Team entwickelt und dass jeder Service unabhängig von den anderen weiterentwickelt, deployt und skaliert werden kann.

Oft stellt sich heraus, dass die Implementierung von Microservices im großen Maßstab nicht so einfach ist wie gedacht. Obwohl das Konzept die Unabhängigkeit der Dienste vorsieht, findet sich in der Realität großer Systeme oft ein Phänomen, das als verteilter Monolith bekannt ist. Dieser entsteht, wenn ein mit Microservices aufgebautes System genauso eng gekoppelt und schwer zu verwalten ist wie die monolithischen Anwendungen, die es ersetzen sollte. Dieses Problem tritt typischerweise dann auf, wenn Microservices zu stark voneinander abhängig sind.

Diese Abhängigkeiten können verschiedene Ausprägungen haben, wie geteilte Datenmodelle oder globale Workflows mit eng gekoppelten Schritten – und insbesondere synchrone Kommunikation: Viele Microservices verwenden synchrone HTTP-Aufrufe. Dadurch wirken sich Ausfälle oder erhöhte Latenz in einem Service kaskadenartig auf das gesamte System aus und beeinträchtigen dessen Verfügbarkeit. Aufgrund dieser Probleme erweist sich die Verwaltung großer, verteilter Systeme in der Praxis oft als genauso komplex, wenn nicht sogar komplexer als die Verwaltung monolithischer Architekturen.

Wenn man mit dem Antipattern „verteilter Monolith“ konfrontiert ist, in welche Richtung sollte man gehen? Manche befürworten sogenannte modulare Monolithen, bei denen die Anwendung in Module innerhalb eines einzigen Deployment-Artefakts organisiert ist. Das kann zwar die interne Struktur und Wartbarkeit verbessern, modulare Monolithen bleiben jedoch Monolithen: Sie müssen als Ganzes bereitgestellt und skaliert werden, und die Module teilen sich Laufzeitumgebung und Abhängigkeiten. Diese inhärenten Einschränkungen machen sie weniger geeignet, wenn Unabhängigkeit, Agilität und Skalierbarkeit die Hauptziele sind.

Andere Organisationen entscheiden sich dafür, Microservices mit Hilfe von Event-driven Architecture (EDA) weiterzuentwickeln. Dieser Ansatz reduziert die Laufzeitkopplung der Services und ermöglicht es ihnen, unabhängig zu arbeiten. Die Entkopplung reduziert die Notwendigkeit enger Koordination zwischen Diensten und vermeidet viele Herausforderungen, die durch synchrone Kommunikation entstehen. Event-getriebene Architekturen können helfen, das Potenzial von Microservices auszuschöpfen.

Für welchen Zweck ist also Event-driven Architecture geeignet? Dann, wenn mehrere Teams möglichst unabhängig voneinander entwickeln können sollen und wenn die Anforderungen an Skalierbarkeit und Verfügbarkeit lauten, dass die einzelnen Services zur Laufzeit nicht voneinander abhängen sollen (d. h., dass mangelnde Verfügbarkeit oder Performance eines Services die anderen nicht beeinträchtigen soll).

EDA heute: Event-driven Microservices

Der Begriff Event-driven Architecture ist älter als Microservices und hat spätestens in den frühen 2000ern im Kontext von Service-oriented Architecture (SOA) einen festen Platz in der Softwarearchitektur gefunden. Wie auch der Begriff „Event“ selbst, kann EDA je nach Epoche und Kontext unterschiedliche Dinge bedeuten. Wenn wir heute von EDA sprechen, sind in den allermeisten Fällen Event-driven Microservices gemeint. Einige wichtige Eigenschaften, die moderne EDA ausmachen, sind die folgenden:

- EDA ist Teil der Makroarchitektur. Sie betrifft die Kommunikation von Services untereinander und befasst sich nicht mit den internen Implementierungsdetails einzelner Dienste. Insbesondere auch nicht damit, ob diese intern Events nutzen (z. B. Event Sourcing als Persistenzmodell) oder nicht.
- Starke Konsistenz intern, verzögerte extern. EDA unterscheidet zwischen, um den Titel eines wichtigen Aufsatzes von Pat Helland zu bemühen, „Data on the Inside vs. Data on the Outside“ [1]. Innerhalb eines Services kann es strikte Konsistenz mit ACID-Garantien geben. Nach außen, also zwischen Services, gibt es immer nur verzögerte Konsistenz, oft mit dem englischen Begriff „Eventual Consistency“ bezeichnet.
- Keine verteilten Transaktionen. Im Einklang mit modernen Best Practices für verteilte Systeme werden verteilte Transaktionen (2PC/XA/JTA) bewusst vermieden. Verteilte Transaktionen versuchen, strikte Konsistenzgarantien auf mehrere Dienste oder Datenspeicher auszudehnen, was zu erhöhter Komplexität und damit erheblichen Nachteilen in Bezug auf Skalierbarkeit und Verfügbarkeit führen würde.

Grundlagen: Was sind Events?

Im Kern der Event-driven Architecture stehen Events, also Ereignisse. Davon gibt es in der IT viele Varianten. In diesem Kontext nicht interessant sind zum Beispiel UI-Ereignisse (*MouseClicked*, *ButtonPressed*), Log-Events oder IoT-Daten. Eine angepasste Definition, die den Begriff sinnvoll einschränkt, könnte sein: *Ein Event beschreibt ein unbestrittenes Ereignis, das in der Vergangenheit aufgetreten ist und Auswirkungen auf den Zustand eines Geschäftsobjekts hatte.*

Hiermit sind einerseits alle nicht businessrelevanten Events (siehe oben) ausgeschlossen, andererseits wird das Faktische von Events betont. Man kann es gar nicht oft genug wiederholen: Events enthalten Tatsachen und sind unveränderlich.

In einem größeren Kontext, wenn man von verschiedenen Arten von Events spricht, würde man diese übrigens „Integration Events“ nennen, da sie der Kommunikation zwischen verschiedenen Domänen dienen (also die verschiedenen Subsysteme integrieren). Da sie hier die einzig relevante Art von Events sind, sparen wir uns die weitere Qualifizierung.

Noch klarer wird der Begriff vielleicht, wenn man ihn anderen Nachrichtenarten gegenüberstellt. Nicht jede Nachricht ist ein Event. In der Regel unterscheidet man zwischen den drei Typen Event (Ereignis), Command (Befehl) und Query (Abfrage). Befehle und Abfragen erwarten jeweils eine Antwort. Ein Befehl kann grundsätzlich fehlschlagen, d. h., das empfangende System ist vielleicht gerade nicht in der Lage, ihn auszuführen. Die Antwort auf den Befehl ist daher die Erfolgs- oder Fehlermeldung.

Bei einer Abfrage besteht die Antwort entweder aus den angeforderten Daten – in der Regel dem Zustand eines oder mehrerer Geschäftsobjekte – oder aus der Information, dass zu den angegebenen Kriterien keine Daten vorliegen (Tabelle 1).

	Event (Ereignis)	Command (Befehl)	Query (Abfrage)
Beschreibt …	eine Tatsache, ein Ereignis, das in der Vergangenheit bereits stattgefunden hat.	eine Intention, die Absicht, eine Operation auszuführen oder einen Zustand zu ändern.	eine Anfrage nach dem aktuellen Zustand eines oder vieler Objekte.
Erwartete Antwort	keine	Bestätigung, dass das Kommando ausgeführt werden konnte, oder Fehlermeldung	der angefragte aktuelle Zustand

TABELLE 1 Unterschiede zwischen Nachrichtenarten (Quelle: [2])

Events unterscheiden sich also von den anderen Nachrichtenarten grundsätzlich dadurch, dass sie keine Antwort erwarten. Es wird ja nur eine Tatsache mitgeteilt – warum sollte man darauf eine Antwort geben? Es wird daher oft gesagt, ein Event könne nicht fehlschlagen. Nun ist es leider nicht so, dass es in Event-getriebenen Systemen überhaupt keine Fehler mehr gibt. Aber im Gegensatz zum Befehl, bei dem der Befehlsabsender gegebenenfalls auf eine Fehlermeldung reagieren muss, liegt die Behandlung beim Empfänger. Wenn das System mit einer Tatsache nicht umgehen kann, ist es das Problem des (empfangenden) Systems, nicht das der Tatsache.

Im Kontext von Kommunikationsmustern können wir davon sprechen, dass Befehle und Abfragen einem „Request/Response"-Muster folgen, während für Events „Fire and Forget" gilt.

Wie sehen Events aus?

Events erfüllen in einer Event-driven Architecture eine duale Funktion. Einerseits dienen sie der Übertragung von Informationen. In eventgetriebenen Systemen werden viele Daten repliziert und liegen in verschiedenen Services lokal vor; Events sind das Mittel, um diese Informationen zu verbreiten. Andererseits können Events Aktionen auslösen. So kann beispielsweise ein *InventoryReserved*-Event den *PaymentService* dazu veranlassen, den Zahlungsvorgang zu starten.

Für viele Services werden beide Aspekte relevant sein. Sie werden, abhängig von der Art des Events, eine Aktion ausführen und lokale Daten aktualisieren. Andere Services interessieren sich vielleicht nur für die Art des Events, wieder andere nur für die aktualisierten Daten. Beides ist beim Event-Design zu berücksichtigen, und wir werden in einem der folgenden Serienteile näher darauf eingehen.

Schon an dieser Stelle sei aber darauf hingewiesen, wie wertvoll eine gute Namensgebung für Events ist. Oberflächlich ist das die Verwendung des Perfekts als Vergangenheitsform. Ein Event steht immer am Ende eines abgeschlossenen (Teil-)Prozesses und das schlägt sich auch in der Verbform nieder, z. B. *OrderPlaced*, *InventoryReserved*.

Dazu gehört, dass die Bezeichnung aussagekräftig ist. Die obige Abfolge erzählt eine kleine Geschichte, und so sollte es sein. Aus der Abfolge der Events soll sich erkennen lassen, wie der Geschäftsprozess aussieht. Daher ist eine Sprache zu verwenden, die sowohl Techniker als auch Nichtinformatiker verstehen. Aus dem Domain-Driven Design stammt der Begriff der „gemeinsamen Sprache" (Ubiquitous Language), ein Prinzip, dem hier unbedingt zu folgen ist.

Leider ist gelegentlich ein reiner Fokus auf die Daten zu beobachten, ohne dem Grund des Events Aufmerksamkeit zu schenken. Das führt zu generischen „updated“ Events. Der Nachteil sollte offensichtlich sein: Welche Sequenz würde man als jemand, der ein System und die Vorgänge darin verstehen möchte, lieber sehen: *OrderPlaced* → *InventoryReserved* → *PaymentAuthorized* → *OrderShipped* oder *OrderUpdated* → *OrderUpdated* → *OrderUpdated* → *OrderUpdated*?

Die Technik

Viele denken bei Event-driven sofort an Apache Kafka. Das ist grundsätzlich auch nicht falsch. Apache Kafka ist ein logbasierter Message Broker, der in Event-getriebenen Systemen sehr beliebt ist. Logbasierte Message Broker eignen sich architektonisch gut für eventgetriebene Architekturen. Die Gleichsetzung von Kafka mit EDA geht aber teilweise so weit, dass manche Menschen glauben, der Einsatz von Kafka allein würde das System schon Event-driven machen. Das ist natürlich nicht so!

Für Kafka ist das, was gesendet wird, lediglich ein „Record“ – im Grunde ein Bytearray. Ob damit Events oder andere Nachrichten versendet werden – Kafka ist es egal. Insofern ist die Bezeichnung „Kafka ist eine Open-Source-Plattform für verteiltes Event-Streaming“ auf der Homepage des Projektes etwas irreführend. Das ist zwar die beabsichtigte Verwendung, aber nichts hindert einen daran, es anders zu benutzen.

Genauso wie man Kafka auch ohne EDA nutzen kann, kann man auch Event-getriebene Systeme ohne einen logbasierten Message-Broker aufbauen. Beispielsweise mit „Store and Forward“-Queues wie ActiveMQ oder RabbitMQ. Es ginge sogar ganz ohne Messaging-Infrastruktur, zum Beispiel durch die Implementierung von HTTP-Feeds.

Dass es möglich ist, heißt aber nicht unbedingt, dass man es auch tun sollte! Logbasierte Message Broker haben sich für EDA bewährt. Dementsprechend geht diese Artikelserie auch von der Verwendung von Apache Kafka aus.

Die Architektur von Kafka

Die Architektur von Apache Kafka im Detail zu erörtern, würde den Rahmen der Serie sprengen, aber es gibt einige wichtige Grundbegriffe aus Entwicklersicht, die kurz eingeführt werden sollen.

Apache Kafka lässt sich am besten als zentrales, persistentes Log verstehen. Services schreiben Nachrichten in dieses Log, andere Services lesen sie. Kafka ist dabei kein System, das Nachrichten aktiv an Empfänger zustellt, sondern ein gemeinsames Gedächtnis für Events in einem verteilten System.

Services, die Nachrichten schreiben, heißen Producer. Sie publizieren Records auf einem dedizierten Kanal, dem Topic, ohne zu wissen, wer sie später konsumiert (**Abb. 1**). Ein Topic ist dabei kein einzelner Datenstrom, sondern in mehrere Partitionen aufgeteilt. Jede Partition ist ein eigenes, strikt geordnetes Log. Dass der Empfänger die Nachrichten in der Reihenfolge bekommt, in der sie gesendet wurden, ist immer nur innerhalb einer Partition garantiert.

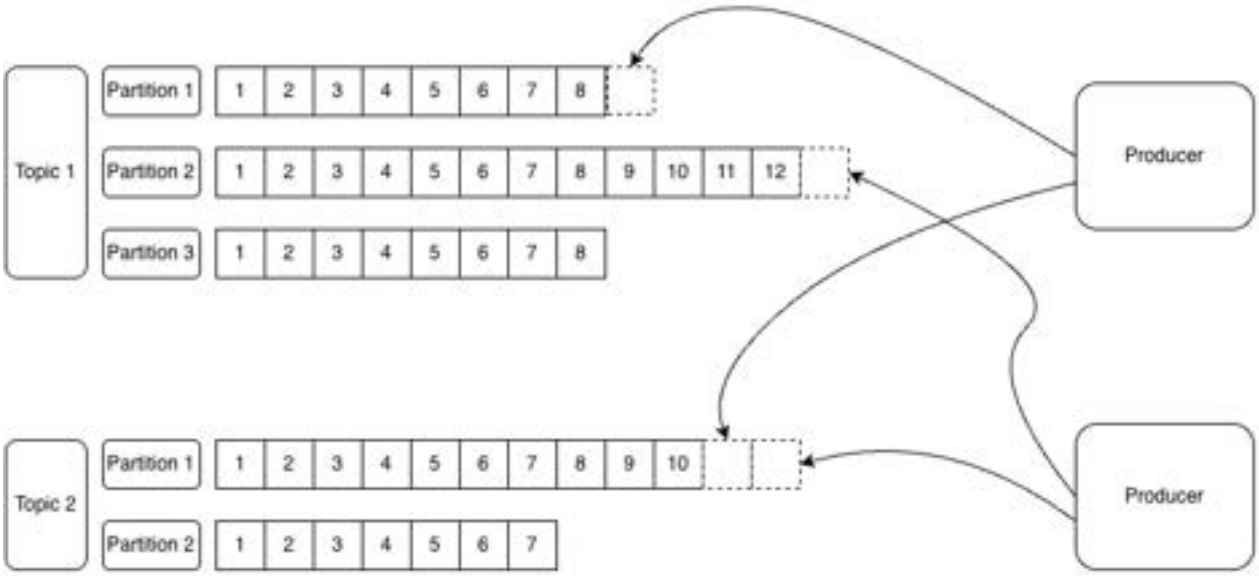


ABB. 1 Producer hängen ans Ende der Partition an

Anwendungen, die Nachrichten lesen, heißen Consumer. Consumer lesen Nachrichten selbstständig, Kafka schiebt keine Nachrichten aktiv zu den Consumern (Konsumieren ist „Pull“, nicht „Push“). Consumer entscheiden selbst, wann und wie viele Nachrichten sie lesen.

Mehrere Instanzen eines Consumer Service bilden eine Consumer Group. Innerhalb einer Gruppe wird jede Partition genau einem Consumer zugewiesen, sodass ein Topic parallel verarbeitet werden kann, ohne dass Nachrichten innerhalb der Gruppe doppelt verarbeitet werden. Verschiedene Consumer Groups können dasselbe Topic unabhängig voneinander lesen.

Kafka speichert Nachrichten für eine konfigurierbare Retention Time, unabhängig davon, ob sie bereits konsumiert wurden. Consumer haben für jede Partition eine aktuelle Leseposition, den Offset. Wird dieser gespeichert, spricht man von einem Commit. Das ist aber nur eine Vereinfachung für die Consumer, damit sie sich den Offset nicht selbst merken müssen, und hat für die Verfügbarkeit der Nachrichten keine Bedeutung.

Consumer können die vorhandenen Nachrichten jederzeit von einem beliebigen Offset an erneut lesen (**Abb. 2**). Kafka speichert also Nachrichten und stellt sie reproduzierbar bereit. Verantwortung für Verarbeitung, Fehlerbehandlung und Duplikate liegt bewusst bei den Consumern.

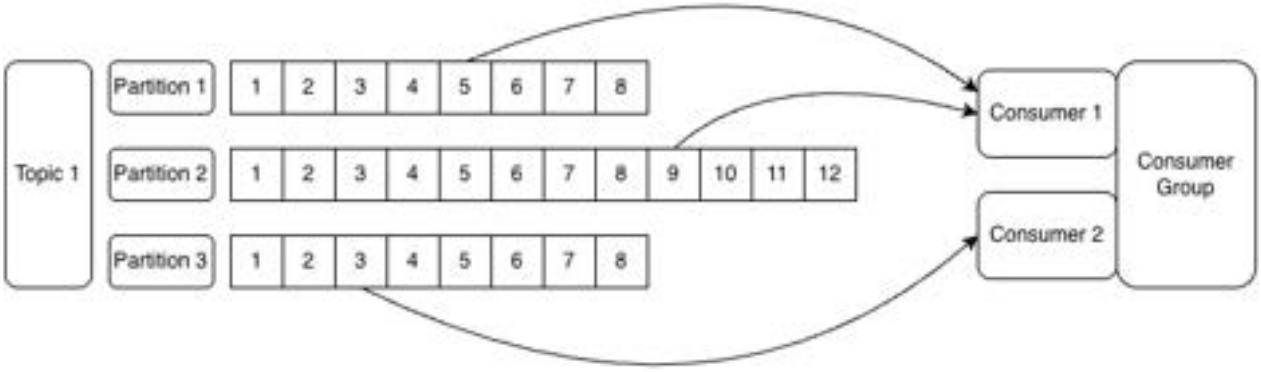


ABB. 2 Consumer lesen ab dem gespeicherten Offset

Die Beispielapplikation: eine Bestellung aufgeben

Um die besprochenen Muster auch mit Code zu belegen, gibt es zur Artikelserie eine kleine Beispielapplikation. Es handelt sich dabei um einen minimalen Bestellservice. Die Anwendung ist fiktiv und unvollständig. Alles, was nicht direkt der Illustration der Konzepte aus der Artikelserie dient, wurde vernachlässigt. Die Anwendung ist also auf keinen Fall als Blaupause für reale Systeme zu verstehen.

Das System besteht aus vier Services: *OrderService*, *InventoryService*, *PaymentService* und *ShippingService*. Sie werden im Lauf der Serie nach und nach eingeführt. Alle Code-Snippets im Artikel entstammen der Beispielapplikation. Das System verwendet Java mit Spring Boot, Apache Kafka als Message Broker und PostgreSQL als Datenbank. Der Source Code ist auf [3] verfügbar.

In der Praxis: die Producer-Seite

Ein Schlüssel, um eventgetriebene Architektur richtig zu verstehen und anzuwenden, ist ein klares Bild von den Verantwortlichkeiten von Producer und Consumer.

In der täglichen Programmierarbeit nehmen wir es als gegeben hin, dass ein Methodenaufruf oder ein HTTP Request in einem Fehler resultieren kann. Wenn wir darüber nachdenken, steckt dahinter eine bestimmte Vorstellung von geteilter Verantwortung. Nehmen wir einen Request über das Netzwerk. Für einen Befehl sähe die Verteilung der Verantwortung aus wie in Tabelle 2.

Verantwortung Befehlsabsender	Verantwortung Befehlsempfänger
auf Fehler reagieren	Befehl ausführen, wenn möglich
mit Nichtverfügbarkeit umgehen	
mit Nichtantwort umgehen (Zeitüberschreitung)	

TABELLE 2 Verantwortungsverteilung bei Befehlen

Viel Verantwortung liegt hier also beim Befehlsgeber. Er muss nicht nur Fehler verarbeiten, sondern auch im Fall von Nichtverfügbarkeit oder Timeouts dafür Sorge tragen, dass der Aufruf zu einem späteren Zeitpunkt wiederholt wird (Retry), und zwar so, dass das aufgerufene System nicht mit einer Lawine von Wiederholungen überschüttet wird (z. B. mit Circuit Breaker).

In einem eventgetriebenen System dreht sich die Verantwortung um. Ist der Event publiziert, ist die Arbeit für den Eventproduzenten erledigt. Die Aufgabe, den Event erfolgreich zu verarbeiten, liegt beim Empfänger. Dazu gehört zum Beispiel auch, mit Duplikaten umzugehen. Die Empfängerseite wird im zweiten Teil der Artikelserie im Mittelpunkt stehen.

Auch wenn der Produzent nicht sehr viel machen muss, ein bisschen ist doch erforderlich: Er muss dafür sorgen, dass alle Events auch tatsächlich mindestens einmal publiziert werden. Es muss eine „Zustellungsgarantie“ geben.

Zustellungsgarantien und das Dual-Write-Problem

Zustellungsgarantien beziehen sich darauf, wie oft eine Nachricht zugestellt wird. Im Fall des Event-Publizierens ist der Begriff etwas irreführend, da unsere Events keinen spezifischen Empfänger haben. Trotzdem hat sich der Begriff „Delivery Guarantee“, der besser zur Eins-zu-eins-Zustellung von Nachrichten an einen bestimmten Adressaten passt, erhalten. In unserem Kontext interpretieren wir es als „Wie oft erscheint die Nachricht im Log unseres Message Brokers?“. Das Problem besteht darin, dass zwei separate Operationen erforderlich sind:

1. Speichern einer Änderung in der lokalen Datenbank und
2. Veröffentlichen des Events auf dem Message Broker.

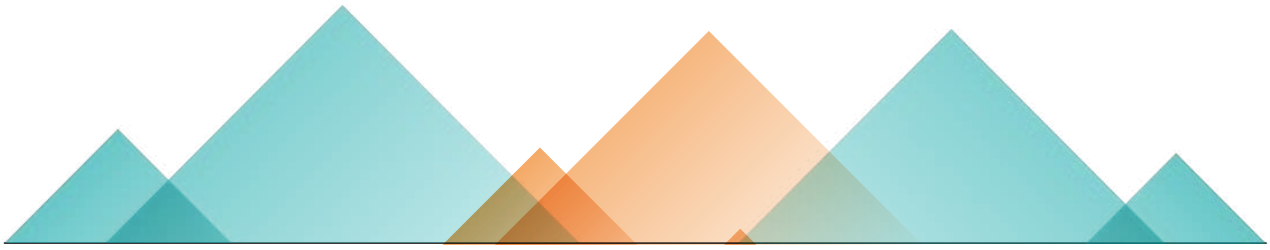
Das wird als Dual-Write-Problem bezeichnet. Auf verteilte Transaktionen verzichten wir dabei. Logbasierte Message Broker sind ohnehin keine transaktionalen Systeme. Kafka verfügt zwar intern auch über sogenannte Transaktionen, die es etwa erlauben, Nachrichten atomar an verschiedene Topics zu senden (oder nicht zu senden). Es gibt aber keinen „Two Phase Commit“, und Kafka kann nicht an verteilten XA/JTA-Transaktionen teilnehmen. Das Senden einer Nachricht an den Broker ähnelt eher einem HTTP-Request als einer Datenbanktransaktion. Es gibt drei typische Zustellungsgarantien.

Die Garantie „höchstens einmal“ (at-most-once) ist sehr leicht zu implementieren. Wir führen Schritt 1 (Update der Datenbank) durch, und dann versuchen wir Schritt 2. Wenn das funktioniert, ist die Nachricht einmal geschrieben, wenn nicht, ignorieren wir den Fehler und vergessen die Nachricht. Leider kommt diese Option aufgrund des offensichtlichen Nachteils, dass Nachrichten einfach verloren gehen können, nicht infrage.

Die Garantie „genau einmal“ (exactly-once) ist die komplexeste. Sie stellt sicher, dass eine Nachricht genau einmal, ohne Duplikate und ohne Verlust, zugestellt wird. Um das zu erreichen, sind zusätzliche Mechanismen wie z. B. Transaktionssysteme erforderlich, um sicherzustellen, dass der Event trotz potenzieller Fehler nur einmal publiziert wird. Bei einem nicht transaktionalen System ist das nur mit Einschränkungen möglich (z. B., wenn Duplikate nur in einem endlichen Zeitfenster auftreten können).

„Genau einmal“ klingt verlockend, ist aber nicht so nützlich, wie es klingt. Angenommen, unser Message Broker würde garantieren, dass die Nachricht genau einmal im Log erscheint. Der Consumer verarbeitet das Ereignis, aktualisiert seine eigene Datenbank, aber stürzt dann ab, bevor er die Verarbeitung bestätigt (d. h. den Offset speichert). Beim Neustart wird der Event erneut aus dem Log abgerufen – selbst, wenn er also nur einmal existiert, kann er mehrfach zugestellt werden. Zustellungs- und Verarbeitungsgarantien sind unterschiedliche Aspekte. Die exakt einmalige Publizierung schützt nicht vor Fehlern auf der Consumer-Seite. Uns interessiert letztendlich nicht die Zustellung, sondern die exakt einmalige Verarbeitung.

Mit der Garantie „mindestens einmal“ (at-least-once) stellt das System sicher, dass eine Nachricht mindestens einmal zugestellt wird. Das bedeutet, dass sie auch mehrmals zugestellt werden kann. Und genau das ist für unsere Zwecke angemessen: Es stellt sicher, dass keine Nachrichten verloren gehen und lässt sich mit überschaubarem Aufwand implementieren.



Wenn unsere erste Operation, das Speichern einer Änderung in der lokalen Datenbank, erfolgreich ist, muss sichergestellt werden, dass ein Event veröffentlicht wird, der andere Dienste über die Änderung informiert. Das gilt auch bei Fehlern beim Senden oder bei einem Systemabsturz, nachdem die Änderung gespeichert, aber bevor das Ereignis veröffentlicht wurde. Um das zu gewährleisten, verwenden wir ein Muster namens Transactional Outbox.

Transactional Outbox

Die Transactional Outbox funktioniert, indem unsere Nachricht zunächst in eine Outbox-Datenbanktabelle geschrieben wird, und zwar in derselben Transaktion, in der wir auch unser Businessobjekt aktualisieren. Ein separater Prozess oder Thread liest diese Nachrichten aus der Outbox-Tabelle und leitet sie an den Broker weiter.

Der Ablauf ist also folgender: Eine Benutzeraktion löst eine Änderung aus (z. B. das Aufgeben einer Bestellung), dann geschieht Folgendes:

- Unser Dienst aktualisiert die Tabelle *Orders*.
- Er fügt einen *OrderPlaced*-Event in die Outbox-Tabelle ein.
- Beide Änderungen erfolgen in derselben Datenbanktransaktion.
- Ein separater Thread (oder Prozess) fragt die Outbox ab, liest ausstehende Ereignisse, veröffentlicht sie an den Broker und markiert sie als gesendet.

Dieser Ansatz stellt sicher, dass die Nachricht bei erfolgreichem Datenbank-Commit garantiert gespeichert und schließlich veröffentlicht wird.

Die Stärke dieses Musters liegt in seiner Einfachheit und Robustheit. Indem wir uns ausschließlich auf die Transaktionsgarantien der Datenbank verlassen, vermeiden wir die Notwendigkeit verteilter Transaktionen. Und da die Nachricht zuverlässig gespeichert wird, kann sie jederzeit später oder erneut gesendet werden. Es trennt die Zuständigkeiten außerdem sauber: Unsere Domänenlogik schreibt Daten und Nachrichten, der Versand erfolgt.

Listing 1

```
@Transactional
public Order createOrder(CreateOrderRequest request,
                         String incomingTraceparent) {
  …
  Order order = repository.insert(
    request.customerId(),
    request.items().stream()
           .map(this::mapItem)
           .toList()
  );

  OrderPlaced event = eventFactory.buildOrderPlaced(order, traceparent);
  outboxService.persistEvent(
    order.id().toString(),
    event.getEnvelope().getEventType(),
  …);
…
}
```

Mit der Transactional-Annotation in Listing 1 stellen wir sicher, dass Datenbankupdate und Eintrag in die Outbox in derselben Transaktion geschehen. In der Beispielapplikation werden die Events mit einem *SELECT*-Statement aus der Outbox-Tabelle gelesen. Das funktioniert, aber für größeren Durchsatz wird in Produktivsystemen häufig auf andere Techniken zurückgegriffen, zum Beispiel auf eine Kombination aus PostgreSQL und einem Tool für „Change Data Capture“ (CDC).

CDC-Tools beobachten das Transaktionslog der Datenbank (bei PostgreSQL: das „Write-ahead Log“, WAL) und erzeugen für jede Änderung einen Event. Das lässt sich auch auf einzelne Tabellen (in unserem Fall also die Outbox) beschränken. Dies ist eine nützliche Optimierung; an der grundlegenden Logik – atomare Datenänderung und Schreiben in die Outbox innerhalb einer Transaktion sowie asynchrones Versenden der Events – ändert sich dadurch jedoch nichts.

To be continued …

In diesem ersten Teil der Serie haben wir gezeigt, warum Event-driven Architecture ein wirkungsvoller Ansatz ist, um die Laufzeitkopplung von Microservices zu reduzieren, und was einen Event im architektonischen Sinn ausmacht. Wir haben die Rolle des Producers, grundlegende Kafka-Konzepte sowie mit der Transactional Outbox ein zentrales Muster für zuverlässiges Publizieren von Events eingeführt.

Im nächsten Teil richten wir den Blick auf die andere Seite: den Consumer. Dabei geht es anhand konkreter Java- und Kafka-Beispiele um zuverlässiges Konsumieren von Events, den Umgang mit „At-least-once Delivery“ und die Frage, wie sich doppelte Verarbeitung durch Idempotenz und Deduplizierung vermeiden lässt.

Links & Literatur

[1] https://www.cidrdb.org/cidr2005/papers/P12.pdf

[2] https://entwickler.de/microservices/events-uberall-events-001

[3] https://codeberg.org/lutzh/javamagazin-eda-series-2026

Event-driven Architecture: eine Einführung – Teil 2

Events verarbeiten: verlässlicher Konsum von Events

Das Interesse an Event-driven Architecture (EDA) hat in den letzten Jahren stetig zugenommen. Moderne Systeme müssen skalieren, ausfallsicher sein und zudem flexibel auf neue Geschäftsanforderungen reagieren können. Inwieweit kann uns EDA dabei helfen? In dieser vierteiligen Serie beleuchten wir das Thema aus architektonischer und praktischer Perspektive.

Wie wir im ersten Teil gesehen haben, enthalten Events Fakten. Diese nehmen wir als gegeben hin, Events können nicht fehlschlagen. Das bedeutet aber leider nicht, dass es in eventgetriebenen Systemen keine Fehler gibt. Das wäre auch zu schön, um wahr zu sein. Vielmehr geht es um Rollen und Verantwortlichkeiten. Der Produzent ist dafür verantwortlich, die Fakten zu veröffentlichen, alle Fakten und nichts als die Fakten. Sobald ein entsprechender Event veröffentlicht ist, ist seine Aufgabe als Produzent für diesen Prozess erledigt. Nun liegt es am Konsumenten, sich mit diesen Fakten auseinanderzusetzen.

Wir können uns das am Beispiel von Sensordaten veranschaulichen. Stellen wir uns einen Messpunkt vor, der die aktuelle Temperatur meldet. Bis zum Zeitpunkt der Entwicklung der Software, die die Daten verarbeitet, wurde nie eine Temperatur über 40 °C gemessen. Daher haben die Entwickler diesen Wert als Obergrenze des gültigen Temperaturbereichs festgelegt. Doch irgendwann steigt die Temperatur zum ersten Mal darüber, und wir erhalten einen Messwert von 42 °C, was zu einem Fehler führt.

Was tun wir also? Was wir nicht machen können, ist, den Event abzulehnen und dem Sensor mitzuteilen, dass der Wert ungültig sei (in einem Request-Response-Szenario wäre das möglich). Es ist nicht das Problem des Sensors: Er hat diese Temperatur gemessen, das ist eine Tatsache. Es ist nun die Aufgabe des Konsumenten, damit umzugehen.

Das ist ein grundlegendes Prinzip der eventgetriebenen Architektur. Verursacht ein Event Probleme, ist der Konsument dafür verantwortlich, sie zu lösen. Wir können nicht den Produzenten bitten, den Sachverhalt nachträglich zu ändern.

Fehlerbehandlung

Der wichtigste Aspekt bei der Fehlerbehandlung ist der Zeitpunkt der Bestätigung von Events. Bei einem logbasierten Message Broker (wie z. B. Kafka) protokolliert jeder Konsument, welche Events er bereits verarbeitet hat. Die Bestätigung eines Events (manchmal mit „Acknowledgement“ bezeichnet, bei Kafka mit „Offset Commit“) ist die technische Mitteilung, dass das Event verarbeitet wurde. „Verarbeitet“ muss dabei nicht unbedingt „erfolgreich verarbeitet“ heißen. Es bedeutet, der Konsument hat das Event gelesen und ist angemessen damit umgegangen – er muss ihm nicht erneut zugestellt werden.

Ganz wichtig: Events, die nicht verarbeitet wurden, dürfen niemals bestätigt werden! Eine verfrühte Bestätigung birgt das Risiko von Datenverlust und würde die „At least once“-Zustellungsgarantie verletzen.

Achtung vor Auto-Commit in Kafka

In Apache Kafka sind einige Client-Bibliotheken standardmäßig so konfiguriert, dass sie Offsets automatisch committen. Wenn der Auto-Commit aktiviert ist, speichert der Consumer die Offsets nach jedem Datenabruf nach der in *auto.commit.interval.ms* konfigurierten Zeit. Das geschieht unabhängig davon, ob die zuletzt abgefragten Events bereits verarbeitet wurden oder nicht! Daher besteht das Risiko, dass Events übersprungen werden, falls der Client abstürzt oder die Events anderweitig nicht angemessen verarbeitet werden können. Um die „At least once"-Zustellung zu gewährleisten, sollte immer *enable.auto.commit* auf *false* gesetzt werden. Werden die Offsets unverarbeiteter Events nicht committet, wird sichergestellt, dass sie später erneut gelesen werden, selbst wenn Konsumenten zwischenzeitlich abstürzen und neu gestartet werden.

Vor diesem Hintergrund ist es eine einfach zu implementierende und naheliegende Strategie zur Fehlerbehandlung, das Event nicht zu bestätigen (oder: den Offset nicht zu committen) und die Verarbeitung erneut zu versuchen. Das wird in einer Schleife so lange wiederholt (am besten mit einer Back-off-Strategie, d. h., die Zeitspanne zwischen den Wiederholungen wird erhöht), bis die Verarbeitung erfolgreich ist. Dieser Ansatz der blockierenden Wiederholung („Blocking Retry") scheint vielleicht zu simpel. Um zu beurteilen, ob er vertretbar ist, sehen wir uns an, welche Arten von Fehlern auftreten können.

Eine Möglichkeit der Kategorisierung besteht darin, zu betrachten, wie hartnäckig der Fehler ist:

- *Vorübergehende Fehler:* vorübergehende Probleme wie Netzwerkverzögerungen, kurzzeitige Ausfälle oder vorübergehende Nichtverfügbarkeit der Datenbank
- *Permanente Fehler:* Probleme, die sich auch durch Wiederholungsversuche nicht lösen lassen, wie z. B. Datenbeschädigung, Parsing-Fehler oder logische Inkonsistenzen

Die zweite Dimension ist das Ausmaß der Auswirkungen des Fehlers:

- *Einzelfehler:* Probleme, die nur den aktuellen Event betreffen, sodass andere Events normal verarbeitet werden können.
- *Systemische Fehler:* Probleme, die die Verarbeitung aller Events beeinflussen.

Für welche Fehler ist der „Blocking Retry" angemessen? Tabelle 1 gibt einen Überblick.

Beharrlichkeit	Auswirkungen	Ist „blockierende Wiederholung“ eine geeignete Strategie?
vorübergehend	systemisch	Da es sich um einen vorübergehenden Fehler handelt, ist ein erneuter Versuch angebracht – sobald die Ressource, die zeitweise ausgefallen ist, wieder verfügbar ist, wird die Verarbeitung fortgesetzt. ✅
vorübergehend	einzelnes Event	siehe oben ✅
dauerhaft	systemisch	Ein dauerhafter Fehler erfordert ein Eingreifen von außen. Ein erneuter Versuch führt in diesem Fall nicht zum Erfolg, ebenso wenig wie das Übergehen zum nächsten Event, da alle Events betroffen sind. Es gibt also keine bessere Alternative. Wir können lediglich sicherstellen, dass die Fehlerbehandlung entsprechende Warnmeldungen ausgibt. ✅
dauerhaft	einzelnes Event	Dies ist die einzige Kombination, bei der die blockierende Wiederholung keinen Sinn ergibt. Wir würden in einer Endlosschleife feststecken, in der wir das fehlerhafte Event immer wieder lesen, obwohl nachfolgende Events verarbeitet werden könnten. ❌

TABELLE 1 Übersicht über Fehlerarten und Blocking Retry

Idealerweise könnten wir für verschiedene Fälle unterschiedliche Strategien wählen. Leider wissen wir im Moment des Auftretens eines Fehlers oft nicht, in welche Kategorie er fällt. Wenn die Datenbank nicht erreichbar ist, handelt es sich dann um eine vorübergehende Netzwerkstörung? Oder hat jemand die Konfiguration verändert und die Netzwerkadresse ist jetzt ungültig? Wenn die Verarbeitung eine Exception auslöst, schlägt das nächste Event dann mit der gleichen Exception fehl? Oder wird er erfolgreich verarbeitet?

Die meisten Fehler fallen in die Kategorie „vorübergehende, systemische Fehler“: Ein Zugriff über das Netzwerk auf ein anderes System schlägt fehl, oder es kommt zu einem Timeout. Häufig liegt das an einer überlasteten Netzwerkverbindung oder einer hohen Auslastung der externen Komponente.

Permanente Fehler hingegen sind zum Beispiel auf fehlerhafte Konfigurationen zurückzuführen. Die Adresse eines Dienstes kann sich geändert haben, ein Zertifikat ist abgelaufen oder das Format der Daten hat sich geändert. In diesen Fällen ist es sinnlos, ein Event zu überspringen und den nächsten zu verarbeiten, da jedes Event dieselben Probleme verursacht.

Fehler, die dauerhaft sind, aber nur ein einzelnes Event betreffen, sind sehr selten. Es bedeutet, dass der Fehler durch den Inhalt des Events verursacht wird. Der Wert eines Felds führt zum Absturz des Clients (also ein Fehler im Code). Oder die Verarbeitung verschiedener Events hängt von unterschiedlichen externen Systemen ab, und eins davon ist dauerhaft ausgefallen.

Angesichts der Seltenheit permanenter Fehler, die ein einzelnes Event betreffen, ist die blockierende Wiederholung eine legitime Strategie und wird in Produktivsystemen eingesetzt. In der Beispielapplikation wird es im *OrderService* so umgesetzt. Es ist eine bewusste Abwägung von Aufwand gegen Risiko.

Wenn tatsächlich ein einzelnes Event dauerhaft fehlschlägt, führt das bei blockierender Wiederholung dazu, dass dieses Event zu einer „Poison Pill" wird. Während der Verarbeitungsversuch der fehlerhaften Nachricht wiederholt wird, sind alle darauffolgenden Nachrichten blockiert. Das wird als „Head of Line Blocking" bezeichnet.

Auf dieser Basis muss das Risiko bewertet werden. Selbst wenn es sehr unwahrscheinlich ist, kann es natürlich grundsätzlich vorkommen. Kann die Behebung dann bis zum nächsten Tag warten? Oder muss ein Entwickler, der Bereitschaftsdienst hat, sich gegebenenfalls mitten in der Nacht darum kümmern?

Wenn wir diesen Fall ausschließen wollen, müssen wir auf eine komplexere Fehlerbehandlung zurückgreifen. Im Fall von Kafka reichen dann die „Bordmittel", also die Funktionen des Standardclients, nicht aus.

Verzögerte Wiederholung

Wenn wir verhindern möchten, dass ein Event zur Poison Pill werden kann, müssen wir Fehler verursachende Events zunächst überspringen und die Verarbeitung später erneut probieren. Kafka ist jedoch grundsätzlich darauf ausgelegt, die Events sequenziell abzuarbeiten. Sobald ein Offset committed wird, gelten alle Events mit einem niedrigeren Offset ebenfalls als verarbeitet.

Wir müssen die fehlerhaften Events also an anderer Stelle ablegen, um mit der Verarbeitung der folgenden weitermachen zu können. Möglichkeiten dafür zeigen z. B. das Spring-Kafka-Template und der Parallel-Kafka-Client.

Verzögerte Wiederholung mit Spring Kafka

Spring Kafka bietet mit den „Retry Topics" [1] eine Lösung für das Problem des „Head of Line Blocking". Das Konzept basiert auf der Verwendung von Topics. Wenn die Verarbeitung eines Events fehlschlägt, wird dieser nicht blockierend wiederholt, sondern in ein spezielles Retry Topic verschoben. Der Konsument bestätigt das ursprüngliche Event und kann mit der Verarbeitung folgender Events fortfahren. Ein separater Listener liest die Events aus dem Retry Topic und versucht die Verarbeitung nach einer konfigurierbaren Verzögerung erneut.

Spring Kafka unterstützt dabei mehrere Retry Topics mit unterschiedlichen Verzögerungen. So kann beispielsweise der erste Wiederholungsversuch nach einer Minute erfolgen, der zweite nach fünf Minuten und der dritte nach einer Stunde. Schlägt auch der letzte Wiederholungsversuch fehl, landet das Event in einem Dead Letter Topic (DLT). Events, die dort landen, müssen analysiert und nach Behebung des Problems erneut verarbeitet werden.

Die Konfiguration wird dabei über die Annotation *@RetryableTopic* am Listener vorgenommen. Spring Kafka erzeugt die benötigten Topics und übernimmt die Mechanik. Diese Strategie löst das Problem des Head of Line Blockings, hat aber einen Preis: Da fehlerhafte Events in separate Topics verschoben werden, ist die Reihenfolge der Verarbeitung innerhalb einer Partition nicht mehr garantiert! Fehlerhafte Events können von anderen Events mit dem gleichen Key überholt werden.

Queue API

Als andere Option kündigt sich eine native Lösung an: Mit KIP-932 „Queues for Kafka“ [2] soll Kafka um eine Art Queue-Semantik erweitert werden. Events werden dabei individuell bestätigt, nicht mehr nur über den Offset.

Für die Fehlerbehandlung bedeutet das: Schlägt die Verarbeitung eines Events fehl, kann es direkt an einen anderen Consumer der Gruppe weitergereicht oder für einen späteren Wiederholungsversuch zurückgestellt werden. Andere Events auf derselben Partition werden davon nicht blockiert. Separate Retry Topics sind nicht erforderlich, da die Queue-Semantik diese Funktionalität bereits mitbringt.

Wie bei Spring Kafkas Retry Topics gilt jedoch auch hier: Die Reihenfolge der Verarbeitung innerhalb einer Partition ist nicht mehr garantiert. Events können von verschiedenen Konsumenten in beliebiger Reihenfolge verarbeitet werden.

Achtung: Das Kafka Queue API befindet sich derzeit noch im Early-Access-Status. Es ist ab Kafka 4.0 als Preview verfügbar, sollte aber noch nicht für produktive Workloads eingesetzt werden.

Der Parallel Consumer von Confluent

Der Parallel Consumer [3], [4] verfolgt einen anderen Ansatz. Die Bibliothek wurde ursprünglich entwickelt, um möglichst viele Events parallel zu verarbeiten, ohne die Anzahl der Partitionen erhöhen zu müssen. Es ist ein Wrapper für den Standard-Kafka-Client und ermöglicht eine höhere Parallelität, als es die Partitionsanzahl eigentlich zulassen würde.

Der Schlüssel liegt in der konfigurierbaren Verarbeitungsreihenfolge. Der Parallel Consumer bietet drei Modi: Ordnung nach Partition (wie beim Standard-Consumer), Ordnung nach Schlüssel (Key) oder vollständig ungeordnet. Im Key-Modus werden Events mit unterschiedlichen Schlüsseln parallel verarbeitet, während die Reihenfolge innerhalb eines Schlüssels strikt eingehalten wird. So können beispielsweise 10 000 verschiedene Kunden gleichzeitig verarbeitet werden, obwohl das Topic nur wenige Partitionen hat, und trotzdem bleibt die Reihenfolge pro Kunde gewahrt.

Der „Non-Blocking Retry" ist dabei ein Nebeneffekt, den wir uns zunutze machen können. Da der Parallel Consumer jeden einzelnen Offset individuell verfolgt und Events mit unterschiedlichen Schlüsseln parallel verarbeiten kann, blockiert ein fehlerhafter Event nur Events mit demselben Schlüssel. Events mit anderen Keys werden unabhängig davon weiterverarbeitet. Das fehlerhafte Event wird mit konfigurierbarer Verzögerung erneut versucht, ohne dass separate Retry Topics benötigt werden. Die Bibliothek speichert den Verarbeitungsstatus komprimiert im Offset Commit, sodass nach einem Neustart nur tatsächlich unverarbeitete Events wiederholt werden. Ein Dead Letter Topic gibt es hier nicht. Die Anzahl der Wiederholungen wird begrenzt, indem das Event im Rahmen des letzten Wiederholungsversuchs unverarbeitet gespeichert und bestätigt wird. In der Beispielapplikation [5] wird der Parallel Consumer im *PaymentService* demonstriert.

Antipattern Inbox

Im ersten Teil haben wir über die Transactional Outbox gesprochen. Dort werden die Events in einer Transaktion mit der Zustandsänderung in der Datenbank gespeichert, bevor sie an den Message Broker gesendet werden.

Eine Art Gegenstück, die „Transactional Inbox", kommt gelegentlich in Architekturdiskussionen auf. Die Idee dabei ist, alle Events erst einmal in einer Datenbanktabelle zu speichern. Die Änderung der Anwendungsdaten kann dann atomar in einer Transaktion durch Löschen (oder als verarbeitet markieren) des Events erfolgen. Das ist ein sehr pessimistischer Ansatz – für Events, die keine Probleme bereiten, ist damit nichts gewonnen. Statt nur Events zu speichern, die Fehler verursachen, werden einfach alle Events erst einmal in der Datenbank gespeichert.

In der Folge sind alle Events doppelt persistiert, auf dem Broker Topic und in der Datenbank. Bei einer relationalen Datenbank trichtern wir die Daten zudem aus den verteilten Partitionen in einen zentralen Server, sodass wir einen Skalierungs-Bottleneck erschaffen. Und wir erhöhen die Komplexität, z. B. müssen wir uns nun auch um das Housekeeping der Inbox-Tabelle kümmern und alte Einträge löschen, wenn sie nicht mehr benötigt werden.

Die Idee, das Lesen von Events von der Verarbeitung zu trennen, klingt oberflächlich attraktiv. In der Realität ist aber mit dem reinen Lesen nichts gewonnen, außer dass die Daten einmal kopiert wurden (in der Regel in ein weniger leistungsfähiges System). Von den tatsächlichen Herausforderungen, die zuvor beschrieben wurden, also die Wiederholung fehlerhafter Nachrichten, das Zurückhalten von Nachrichten, wenn die Reihenfolge eingehalten werden muss usw., ist damit nichts gelöst.

Fehlerbehandlung – ein Fazit

Kafka und Kafka-kompatible Broker sind optimiert für das sequenzielle Lesen der Events. Einzelne Events (vorübergehend) zu überspringen, bringt etwas zusätzliche Komplexität mit sich. Es ist daher für jede Domain, für jeden Use Case zu bewerten, ob das wirklich nötig ist. Oft reicht die simple, blockierende Wiederholung aus. Falls nicht, kann man auf Spring Kafka oder den Parallel Consumer zurückgreifen oder sich für eine mögliche Eigenentwicklung von diesen inspirieren lassen.

Die Tatsache, dass der Standard-Kafka-Client die Nutzer in Bezug auf Wiederholungen im Fehlerfall allein lässt, hat leider auch einige Antipatterns, wie eine „Transactional Inbox“, hervorgebracht. Bevor sie zum Einsatz kommt, sollten unbedingt die oben genannten Client-Libraries in Erwägung gezogen werden.

Andere Broker, die keinen Wert auf Kafka-Kompatibilität legen, bieten mehr Möglichkeiten. Als Beispiel sei Google Pub/Sub genannt. Hier wird grundsätzlich pro einzelner Nachricht bestätigt, und es können automatische Wiederholungen erfolgen (wahlweise auch mit Beibehaltung der Reihenfolge).

Der Umgang mit Fehlern ist aber nicht das Einzige, was dem Konsumenten zufällt. Wir müssen zwei weitere Dinge im Kopf behalten: Umgang mit Duplikaten und die Verarbeitung in der richtigen Reihenfolge.

Reihenfolge der Verarbeitung

Wenn unser System darauf angewiesen ist, dass Events in der richtigen Reihenfolge abgearbeitet werden, müssen wir sicherstellen, dass Events mit identischen Schlüsseln in der Reihenfolge ihres Eintreffens prozessiert werden. Das mag trivial erscheinen, da sie im Broker-Log (bzw. in ihrer Partition) der Reihe nach auftauchen. Aber Vorsicht! Normalerweise werden Events nicht einzeln gelesen. Stattdessen erhalten wir beim regelmäßigen Abfragen des Logs einen Stapel von Events. Und nichts hindert uns daran, mehrere (virtuelle) Threads zu starten und sie parallel zu verarbeiten.

Soll die Reihenfolge bewahrt werden, muss der Konsument Events entweder streng sequenziell abarbeiten: Die Parallelität besteht dann lediglich darin, dass mehrere Konsumenten parallel gestartet werden können, die dann jeweils Partitionen zugewiesen bekommen (*maximale Parallelität = Anzahl der Konsumenten <= Anzahl der Partitionen*). Oder, wenn auch innerhalb eines Konsumenten parallel verarbeitet werden soll, dann muss der Konsument die sequenzielle Verarbeitung pro Key sicherstellen – zum Beispiel mit dem oben erwähnten Parallel Consumer.

Idempotenz und Exactly-once-Verarbeitung

Idempotenz ist die Eigenschaft einer Operation, die bei mehrmaliger Anwendung denselben Effekt hat wie bei einmaliger Anwendung. Formal ausgedrückt: eine Funktion *f* ist idempotent, wenn *f(f(x)) = f(x)*. Klassische Beispiele hierfür sind die Betragsfunktion (*Math.abs*) oder die logischen Operationen *AND* und *OR*, wenn sie wiederholt auf dieselben Operanden angewendet werden.

Das Konzept basiert auf der Stabilität bei wiederholter Anwendung: Sobald eine Operation ausgeführt wurde, ändert ihre erneute Anwendung das Ergebnis nicht. In der Informatik wird der Begriff verwendet, um Operationen (insbesondere in APIs und verteilten Systemen) zu beschreiben, die sicher wiederholt werden können, ohne das Ergebnis über die ursprüngliche Anwendung hinaus zu verändern.

Wie im ersten Teil der Serie beschrieben, gehen wir von „At least once delivery“ aus, also davon, dass unser System eine Zustellung mindestens einmal garantiert. Das bedeutet, dass Events auch mehrfach zugestellt werden können. Das soll jedoch das Ergebnis nicht beeinflussen. Sofern die Verarbeitung nicht von Natur aus idempotent ist, müssen wir sicherstellen, dass die Events auch bei Duplikaten nur einmal verarbeitet werden.

Natürlich idempotente Events

Bestimmte Events sind aufgrund ihrer Natur idempotent in ihrer Verarbeitung. Beispielsweise können Events, die den vollständigen Zustand einer Entität anstatt inkrementeller Zustandsänderungen enthalten, oft sicher erneut verarbeitet werden. Das Erkennen und Ausnutzen natürlicher Idempotenz kann die Konsumentenlogik erheblich vereinfachen. In diesem Fall können wir das Event verarbeiten, ohne prüfen zu müssen, ob es ein Duplikat eines früheren Events ist.

Idempotente Konsumenten

Leider führen Events in Geschäftsobjekten üblicherweise zu Zustandsübergängen, und natürlich idempotente Events sind selten. Bei nicht idempotenten Events müssen unsere Konsumenten sicherstellen, dass jedes Event genau einmal verarbeitet wird und Duplikate ignoriert werden. Ein System mit dieser Eigenschaft wird als idempotenter Konsument bezeichnet (Idempotent Consumer). Erreicht wird das durch Deduplizierung, also durch das Erkennen und Überspringen doppelter Events.

Deduplizierung

Um Duplikate zu erkennen, muss jedes Event eine eindeutige ID besitzen. Ein gängiger und zuverlässiger Ansatz besteht darin, jedem Event zum Zeitpunkt seiner Erstellung eine global eindeutige Event-ID (GUID/UUID) zuzuweisen.

Der Client führt dann eine Tabelle mit bereits verarbeiteten Event-IDs. Bevor ein Event verarbeitet wird, prüft er, ob die ID bereits bekannt ist, in diesem Fall wird das Event übersprungen. Events mit bisher unbekannten IDs werden weiterverarbeitet.

Ein entscheidendes Detail ist der Zeitpunkt, zu dem wir die ID speichern. Wir müssen die Event-ID (unseren Deduplizierungs-Schlüssel) innerhalb derselben Transaktion speichern, in der wir die Daten unserer Applikation aktualisieren.

Wenn wir die ID vor der Transaktion speichern und der Dienst nach dem Speichern der ID, aber vor der Verarbeitung des Events abstürzt, überspringen wir Events (**Abb. 1**).

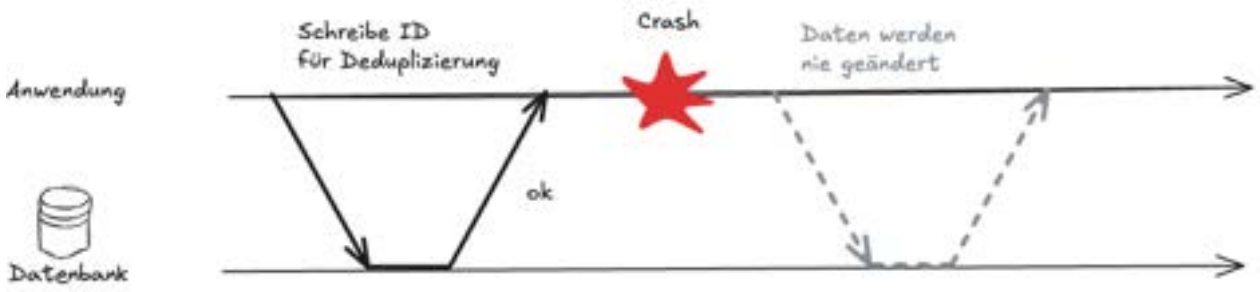


ABB. 1 Die ID vor Datenänderung zu speichern, führt zum Überspringen von Events

Wenn wir die ID nach der Transaktion speichern und der Dienst nach der Verarbeitung des Events, aber vor dem Speichern der ID, beendet wird, verarbeiten wir das Event mehrfach (**Abb. 2**).

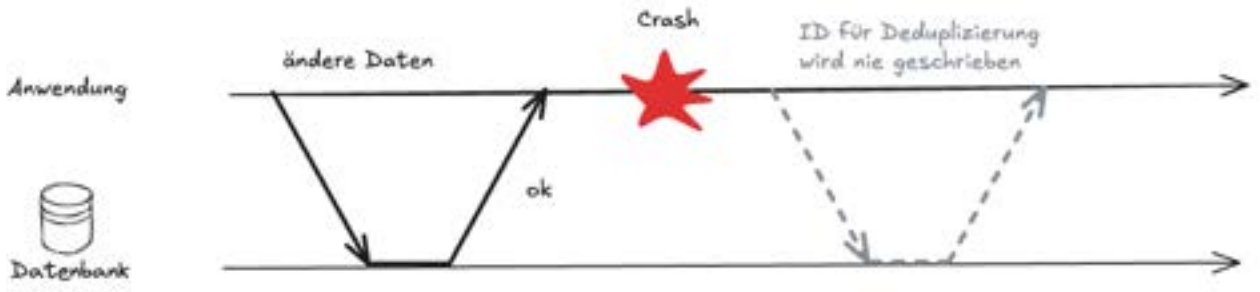


ABB. 2 Die ID nach Datenänderung zu speichern, führt zu doppelter Verarbeitung

Schließlich sollten wir erneut prüfen, ob die ID innerhalb derselben Transaktion bereits bekannt ist, da ein Event mit dieser ID möglicherweise noch in Bearbeitung war, als wir zuvor geprüft haben (bei paralleler Verarbeitung). Alternativ verhindert ein Unique Constraint der Spalte in einer relationalen Datenbank, dass die Transaktion erfolgreich ist, falls dasselbe Event zwischenzeitlich bereits verarbeitet wurde.

Deduplizierungsablauf

Aus der Designperspektive mag dieser Ansatz wie eine schlechte Trennung von Zuständigkeiten wirken. Man könnte die Deduplizierung als technisches Detail betrachten, das nicht mit der Geschäftslogik verknüpft werden sollte. Und tatsächlich könnten wir natürlich einen Filter einbauen, der alle doppelten Events verwirft, bevor unsere Event-Verarbeitung aufgerufen wird (**Abb. 3**). Doch das führt uns zurück zur Diskussion um „At least once delivery“ versus „At least once processing“ (siehe erster Serienteil).

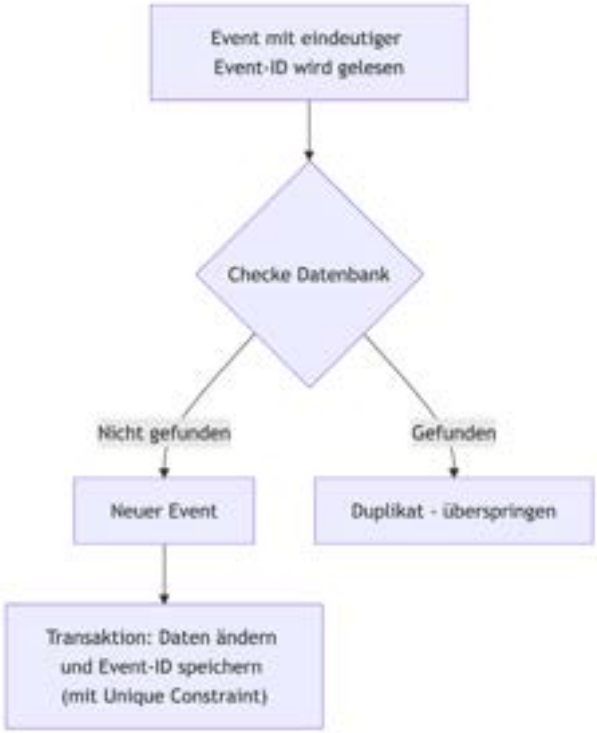


ABB. 3 Deduplizierung im Überblick

Ein in sich geschlossener Duplikatfilter liefert zwar das gewünschte Ergebnis, befreit uns aber dennoch nicht von der Notwendigkeit, bei der Verarbeitung erneut auf Duplikate zu prüfen. Da unsere Geschäftstransaktion und die Speicherung der ID atomar erfolgen müssen (entweder beides erfolgreich oder keines von beiden), lassen sie sich leider nicht vollständig trennen.

Es ist üblich, die Deduplizierung auf ein Zeitfenster (das „Deduplication Window“) zu beschränken, da Duplikate meist durch sporadische Fehler und nachfolgende Wiederholungsversuche entstehen. Ein Duplikat eines Events taucht nicht einfach einige Tage nach dem ersten auf. Dadurch können wir die Nachschlagetabelle regelmäßig bereinigen und alte IDs löschen.

Wenn wir jedoch aus irgendeinem Grund ein sehr großes Zeitfenster oder einfach eine große Anzahl von Events (oder beides) haben, kann die Überprüfung der Datenbank für jedes einzelne Event einen zu hohen Aufwand bedeuten. Dann müssen wir die Abfrage optimieren, z. B. durch eine Vorfilterung durch einen Bloom-Filter. Ein Bloom-Filter ist eine Datenstruktur, mit deren Hilfe sehr schnell festgestellt werden kann, welche Daten in einem Datenstrom schon einmal vorgekommen sind und welche erstmals auftreten [6].

To be continued …

Im ersten Teil haben wir uns mit der Eventdefinition beschäftigt und mit den Aufgaben des Produzenten von Events. Wie wir dort schon angekündigt haben, ist die komplexere Seite in der Event-driven Architecture die Konsumentenseite.

Diese Folge widmete sich daher komplett dem Konsumieren von Events: Fehlerbehandlung, Reihenfolge und Deduplizierung.

Bisher haben wir in unserem System immer nur von zwei Services gesprochen: Produzent und Konsument. Im nächsten Teil werden wir uns ansehen, welche Herausforderungen entstehen, wenn wir Prozesse betrachten, in die mehrere Services involviert sind.

Links & Literatur

[1] Spring-Kafka-Dokumentation: https://docs.spring.io/spring-kafka/reference/retrytopic.html

[2] KIP: https://cwiki.apache.org/confluence/display/KAFKA/KIP-932%3A+Queues+for+Kafka

[3] Github: https://github.com/confluentinc/parallel-consumer

[4] Blog-Post: https://www.confluent.io/blog/introducing-confluent-parallel-message-processing-client/

[5] Beispielapplikation zur Artikelserie: https://codeberg.org/lutzh/javamagazin-eda-series-2026

[6] Wikipedia-Seite zu Bloom-Filter https://de.wikipedia.org/wiki/Bloomfilter

Event-driven Architecture: eine Einführung – Teil 3

Events kombinieren: Zusammenarbeit mehrerer Services

In den letzten Jahren ist das Interesse an Event-driven Architecture (EDA) stetig gestiegen. Moderne Systeme müssen skalierbar und ausfallsicher sein und zudem flexibel auf neue Geschäftsanforderungen reagieren können. Inwieweit kann uns dieser Ansatz dabei helfen? Im dritten Teil geht es darum, wie mehrere Services in einem Event-getriebenen System zusammenarbeiten und wie sich Orchestrierung und Choreografie dabei unterscheiden.

In den ersten beiden Teilen haben wir uns mit dem Publizieren und Konsumieren von Events beschäftigt, jeweils aus der Perspektive zweier beteiligter Services: eines Produzenten und eines Konsumenten. In diesem Teil erweitern wir den Blick, denn in der Praxis bestehen Geschäftsprozesse selten aus nur zwei Beteiligten. In der Domäne unseres kleinen Beispiels erfordert eine Bestellung die Prüfung des Inventars, die Abwicklung der Zahlung und schließlich den Versand. Wie koordinieren wir das Zusammenspiel mehrerer Services?

Damit stehen zwei Architekturansätze im Raum: Orchestrierung und Choreografie. Beide Ansätze bieten Lösungen für die Koordination verteilter Prozesse. Doch nicht beide passen gleich gut zu einer Event-driven Architecture.

Das Beispiel: eine Bestellung als Prozess

Erweitern wir unser Bestellsystem [1]. Bisher hat der *OrderService* ein *OrderPlaced*-Event publiziert und ein einzelner Konsument hat darauf reagiert. Nun betrachten wir den kompletten Ablauf, der in **Abbildung 1** dargestellt ist.

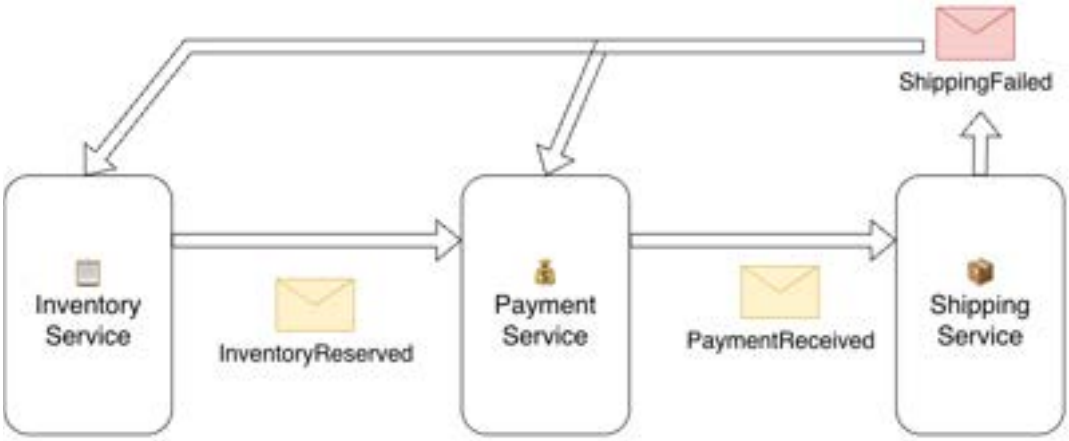


ABB. 1 Der komplette Happy Path der Bestellung

Demnach müssen zwei Bedingungen erfüllt sein, bevor eine Bestellung bestätigt werden kann: *InventoryReserved* und *PaymentReceived*. Das wirft die Frage auf: Wer stellt sicher, dass dieser Prozess korrekt abläuft? Wer „weiß“, dass sowohl Inventar als auch Zahlung bestätigt sein müssen?

Orchestrierung: ein zentraler Koordinator

Ein naheliegender Ansatz ist, einen zentralen Koordinator einzuführen, der den Prozess steuert. Dieser Koordinator kennt die Schritte des Prozesses und deren Reihenfolge. Er weist die beteiligten Services an, ihre Aufgaben auszuführen, wartet auf die Ergebnisse und bestimmt den nächsten Prozessschritt.

In der Praxis könnte das so aussehen: Der *OrderService* nimmt die Bestellung entgegen und agiert selbst als Koordinator. Er fordert den *InventoryService* auf, das Inventar zu reservieren, und den *PaymentService*, die Zahlung zu autorisieren. Sobald beide Bestätigungen vorliegen, weist er den *ShippingService* an, den Versand vorzubereiten.

Variante 1: bestehender Service

Die einfachste Form der Orchestrierung ist es, einen der bestehenden Services zum Koordinator zu machen. Wir könnten diese Rolle dem *OrderService* zuweisen, da er ohnehin am Anfang des Prozesses steht. Er würde dann nicht nur Bestellungen entgegennehmen, sondern auch den gesamten Ablauf steuern, d. h., Events publizieren, auf Antworten warten und den nächsten Schritt anstoßen.

Wie **Abbildung 2** zeigt, bekommt der *OrderService* dadurch eine Sonderrolle. Er ist nicht mehr ein Service unter Gleichen, sondern trägt die Verantwortung für den gesamten Prozess. Prozessänderungen betreffen immer den *OrderService*, selbst wenn sie eigentlich die Zahlung oder den Versand betreffen. Fachfremde technische Aufgaben wie Timeout-Behandlung, Fehlerszenarien und Zustandsverwaltung landen in diesem Service.

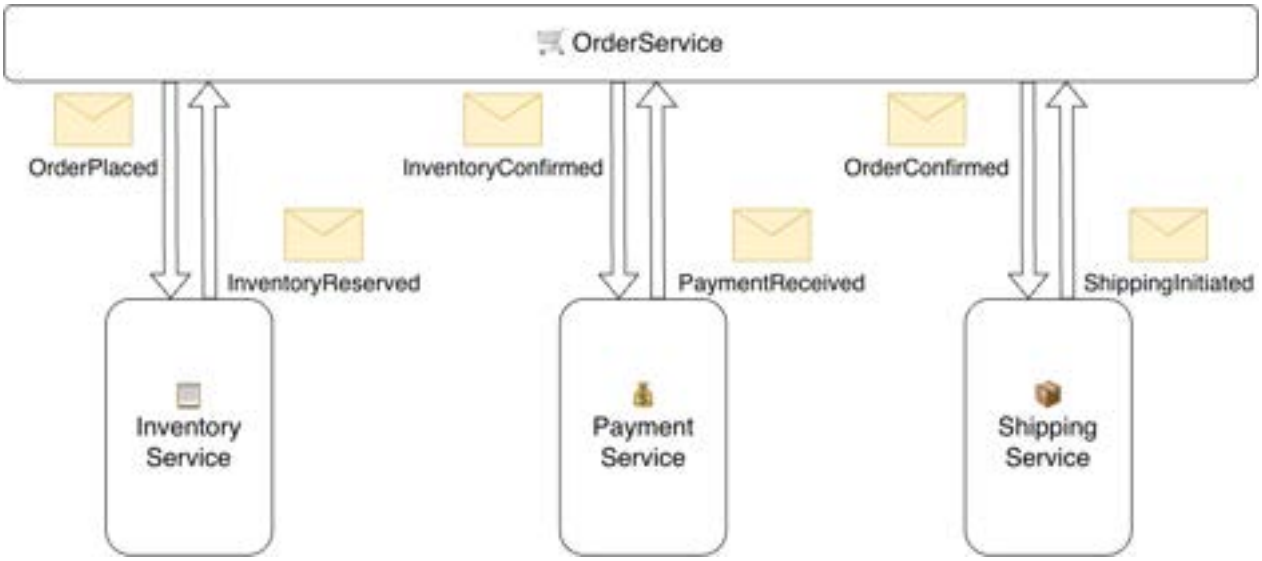


ABB. 2 Der Order-Service als Orchestrator

Variante 2: eine Workflow-Engine

Um diese Komplexität aus dem fachlichen Service herauszuhalten, kann man stattdessen auch auf dedizierte Workflow-Engines zurückgreifen. Diese Tools sind darauf spezialisiert, Geschäftsprozesse über mehrere Services hinweg zu koordinieren. Einige bekannte Vertreter:

- *Temporal.io* verwendet eine Programmiersprache (Java, Go, TypeScript u. a.) zur Definition von Workflows. Der Workflow wird als Code geschrieben, Temporal übernimmt Zustandsverwaltung und Fehlerbehandlung [2].
- *Camunda* setzt auf BPMN (Business Process Model and Notation), einen XML-basierten Standard zur grafischen Modellierung von Geschäftsprozessen [3].
- *Google Cloud Workflows* beschreibt Abläufe in YAML und integriert sich in das Google-Cloud-Ökosystem [4].
- Die *Cloud Native Computing Foundation (CNCF)* definiert mit der Serverless Workflow Specification ein herstellerunabhängiges Format, ebenfalls YAML-basiert [5].

Diese Werkzeuge bieten eine saubere Trennung: Die fachlichen Services kümmern sich um ihre Domäne, der Workflow beschreibt den übergreifenden Prozess. Das klingt zunächst nach einer eleganten Lösung. Bevor wir aber die Nachteile betrachten, lohnt sich ein Blick auf ein Muster, das in diesem Zusammenhang fast immer auftaucht.

Das Saga-Pattern

Wer sich mit der Koordination verteilter Prozesse beschäftigt, wird mit ziemlicher Sicherheit auf das Saga-Pattern stoßen. Es ist eines der meistzitierten Muster im Microservices-Kontext, das beschreibt, wie sich verteilte Systeme ohne verteilte Transaktionen einem ‚Alles-oder-nichts'-Verhalten annähern können.

Das Grundprinzip

Die Idee ist einfach: Für jeden Schritt im Prozess gibt es eine kompensierende Aktion. Wenn ein Schritt fehlschlägt, werden alle bereits ausgeführten Schritte durch ihre jeweilige Kompensation rückgängig gemacht (eine Anwendung auf unser Bestellbeispiel zeigt Tabelle 1). Das Saga-Pattern wird auch als „Compensating Transaction Pattern" bezeichnet, was den Kern besser trifft.

Schritt	Aktion	Kompensierende Aktion
1	Inventar reservieren	Reservierung aufheben
2	Zahlung einziehen	Zahlung zurückbuchen

TABELLE 1 Aktionen und kompensierende Aktionen in unserem Bestellbeispiel

Schlägt die Zahlung fehl, nachdem das Inventar bereits reserviert wurde, muss die Reservierung rückgängig gemacht werden. Die Saga stellt sicher, dass das System am Ende wieder in einem konsistenten Zustand ist, entweder vollständig abgeschlossen oder vollständig zurückgerollt.

Saga und Eventual Consistency

Die Saga akzeptiert dabei bewusst temporäre Inkonsistenzen. Zwischen der Reservierung des Inventars und der (möglicherweise fehlschlagenden) Zahlung befindet sich das System in einem Zwischenzustand: Das Inventar ist reserviert, aber die Bestellung ist noch nicht bestätigt. Erst wenn die Saga entweder erfolgreich abgeschlossen oder vollständig kompensiert wurde, ist ein konsistenter Zustand erreicht. Man könnte die Saga als eine Art geistigen Nachfolger verteilter Transaktionen betrachten. Sie bietet „alles oder nichts“, allerdings nicht mit strikter, sondern mit verzögerter Konsistenz.

Saga-Koordinator

Eine Saga wird von einem zentralen Saga-Koordinator gesteuert (**Abb. 3**). Dieser kennt die Abfolge der Schritte sowie die zugehörigen Kompensationen. Er steuert den Prozess aktiv: Er sendet Befehle an die Services, wartet auf Antworten und entscheidet basierend auf dem Ergebnis, ob der nächste Schritt oder eine Kompensation ausgeführt wird.

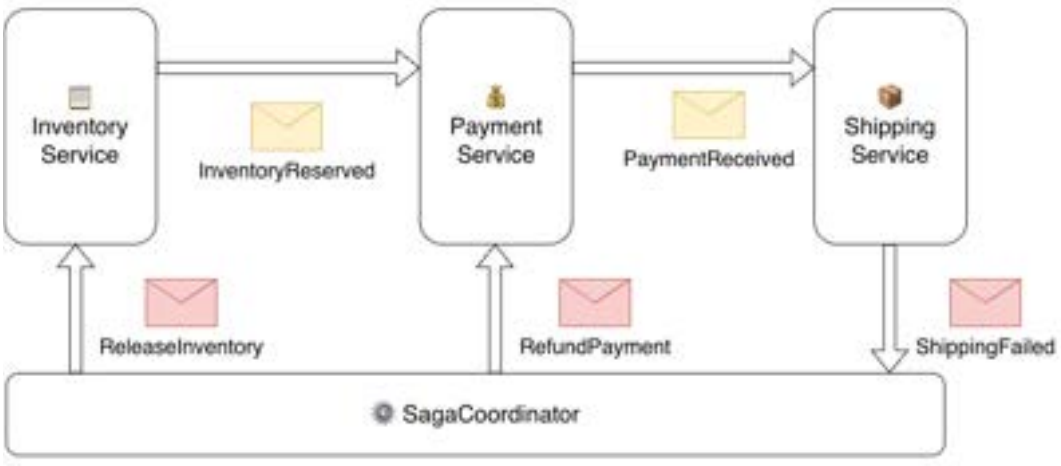


ABB 3 Das Saga-Pattern

Ein Saga-Koordinator kann als eigener Service implementiert oder durch eine Workflow-Engine bereitgestellt werden. Alle zuvor genannten Workflow-Engines unterstützen das Saga-Pattern.

Hinweise zum Saga-Pattern

Die Begrifflichkeit rund um das Saga-Pattern ist nicht einheitlich. In diesem Artikel verwenden wir „Saga“ als Synonym für das Pattern, das Microsoft als „Kompensierende Transaktion“ (Compensating Transaction Pattern) beschreibt: ein orchestriertes Muster mit einem zentralen Koordinator, der die Schritte und deren Kompensationen verwaltet [6]. Der Begriff wird andernorts mitunter breiter gefasst und schließt auch andere Varianten ein.

Sagas sind kein Allheilmittel. Zum Beispiel ist zu beachten, dass auch die kompensierende Transaktion fehlschlagen kann. Zudem muss das Rückgängigmachen einer Aktion nicht immer die beste Lösung sein. Um in unserer Beispieldomäne zu bleiben: Ist die Zahlung bereits abgewickelt, aber der Artikel vorübergehend nicht verfügbar, gibt es verschiedene Lösungen. Vielleicht möchte der Kunde warten oder akzeptiert als Ersatz einen anderen, ähnlichen Artikel oder einen Gutschein. Vorsicht ist geboten, damit das Saga-Pattern nicht zu früh als Lösung für das Problem in Betracht gezogen wird.

Die Probleme der Orchestrierung

Sowohl die Orchestrierung durch einen eigenen Service als auch die Verwendung einer Workflow-Engine sowie das Saga-Pattern teilen eine grundlegende Eigenschaft: Sie alle setzen auf einen zentralen Koordinator, der den Prozessablauf steuert. Dadurch entstehen ähnliche Probleme.

Komplexe Infrastruktur

Ein Prozesskoordinator muss den Zustand des Prozesses verwalten. Er muss wissen, welche Schritte bereits ausgeführt wurden, welche noch ausstehen und welche fehlgeschlagen sind. Dieser Zustand muss persistent sein, da der Koordinator abstürzen kann. Zudem muss er in einer verteilten Umgebung funktionieren, da nach einem Absturz möglicherweise eine andere Instanz den Prozess fortführen muss.

Das bedeutet: Es wird eine verteilte Zustandsverwaltung benötigt, Crash-Recovery muss möglich sein und ein Single Point of Failure muss vermieden werden. Hier kommen Koordinationsprotokolle wie Raft oder Paxos zum Einsatz. Das ist kein triviales Problem.

Wer eine Workflow-Engine einsetzt, muss diese Komplexität nicht selbst implementieren. Doch dafür handelt man sich eine neue, komplexe Infrastrukturkomponente ein, die betrieben, überwacht und aktualisiert werden muss. Ein Teil der Geschäftslogik lebt nun in der Sprache der Workflow-Engine, sei es BPMN/XML, YAML oder ein proprietäres API. Das erschwert das Testen und die Wartung.

Wer den Koordinator selbst implementiert, sieht sich mit einer noch größeren Herausforderung konfrontiert. Der Koordinator muss Timeouts verwalten, auch solche, die während eines Absturzes ablaufen. Er muss den Timeout-Zustand speichern. Wer einen eigenen Saga-Koordinator entwickelt, landet am Ende möglicherweise bei einer selbst entwickelten Workflow-Engine – wobei man dann vielleicht doch lieber ein fertiges Produkt eingesetzt hätte.

Unklare Verantwortlichkeiten

Ein oft unterschätztes Problem ist die organisatorische Unschärfe, die die Orchestrierung mit sich bringt. Wenn ein Workflow die Koordination übernimmt, stellt sich bei jedem Fehler die Frage: Liegt das Problem im Workflow oder im Service?

In der Praxis führt das zu aufwendigen Abstimmungen. Der Workflow muss die Fehlermeldungen der Services verstehen. Die Services müssen wissen, welche Fehlertypen der Workflow erwartet. Geht etwas schief, müssen mehrere Teams gemeinsam analysieren, ob es sich um einen erwarteten Fehlerfall handelt, den der Workflow kompensieren sollte, oder um einen echten Bug im Service.

Diese geteilte Verantwortung widerspricht einem Grundprinzip autonomer Teams: Jedes Team sollte für seinen Service verantwortlich sein, von der Entwicklung über das Deployment bis zur Fehlerbehandlung.

Nicht Event-driven

Da unser Ziel eine Event-getriebene Architektur ist, kommt ein grundlegendes Problem hinzu: Orchestrierung ist nicht Event-driven. Eventuell werden die Nachrichten zwar über Kafka verschickt. Und vielleicht sind sie auch pro forma als „Events“ formuliert. Betrachten wir jedoch das zugrunde liegende Muster: Sowohl Orchestrierung als auch das Saga-Pattern setzen voraus, dass der Koordinator über den Erfolg oder Misserfolg jeder Aktion informiert wird. Der Koordinator sendet einen Auftrag an einen Service und erhält eine Rückmeldung: erledigt oder fehlgeschlagen. Auf Basis dieser Rückmeldung entscheidet der Koordinator, was als Nächstes passiert, ob der nächste Schritt eingeleitet wird oder ob eine Kompensation nötig ist. Das ist, unabhängig vom Transportweg, ein Request-Response-Muster.

In Teil 1 haben wir definiert: Ein Event beschreibt ein unbestreitbares Ereignis, das in der Vergangenheit stattgefunden hat. Events sind Fakten. Der Publisher kümmert sich zur Laufzeit nicht darum, wer das Event konsumiert oder was damit geschieht. Er wartet nicht auf eine Antwort, denn auf eine Tatsache gibt es nichts zu antworten.

In einem orchestrierten Prozess verhält es sich anders. Manchmal ist die Abweichung offensichtlich: Der Koordinator sendet einen expliziten Befehl per HTTP oder gRPC und wartet synchron auf die Antwort. Im obigen Beispiel ist sie etwas subtiler: Die Nachrichten tragen Event-Namen. Aber der Koordinator wartet trotzdem auf eine Bestätigung, bevor er fortfährt. Das zugrunde liegende Muster ist Request-Response, auch wenn der Transportweg asynchron ist.

Ob die Kommunikation über HTTP, gRPC oder einen Message Broker läuft, ist für diese Frage zweitrangig. Entscheidend ist: Erwartet der Absender eine Antwort? Muss er wissen, ob die Aktion erfolgreich war? Wenn ja, ist es Request-Response, und damit Command-driven, nicht Event-driven.

Choreografie: der Event-driven Ansatz

Wenn Orchestrierung nicht Event-driven ist, wie sieht dann der Event-driven Ansatz aus? Die Antwort heißt „Choreografie“. Bei der Choreografie gibt es keinen zentralen Koordinator. Stattdessen reagiert jeder Service auf Events, führt seine Aufgabe aus und publiziert neue Events. Der Prozess ergibt sich somit aus dem Zusammenspiel der Services, das wir bei der Entwicklung festlegen, und nicht aus einer zentralen Steuerung zur Laufzeit. **Abbildung 4** zeigt einen Ausschnitt eines möglichen Ablaufs als Choreografie. Jeder Service konsumiert Events, führt seine Aufgabe aus und publiziert ein neues Event. Kein Service wartet auf eine Antwort.

ABB. 4 Ein möglicher Teilablauf ohne Orchestrierung

- Der *OrderService* nimmt die Bestellung entgegen und publiziert ein *OrderPlaced*-Event. Die Bestellung befindet sich zu diesem Zeitpunkt im Zustand *SUBMITTED*. Damit ist der *OrderService* fertig. Es erfolgt keine Rückmeldung an den *OrderService*, wie es mit der Bestellung weitergeht.
- Der *InventoryService* konsumiert das *OrderPlaced*-Event. Er ist die Source of Truth für den Bestand und prüft, ob die gewünschten Artikel verfügbar sind. Wenn ja, reserviert er sie und publiziert ein *InventoryReserved*-Event. Wenn nicht, publiziert er ein *OrderRejected*-Event. Auch der *InventoryService* ist dann fertig. Er wartet nicht auf eine Rückmeldung.
- Der *PaymentService* konsumiert das *InventoryReserved*-Event und leitet die Zahlung ein. Bei Erfolg publiziert er ein *PaymentAuthorized*-Event, bei Misserfolg ein *PaymentFailed*-Event.
- Der *ShippingService* konsumiert das *PaymentAuthorized*-Event und bereitet den Versand vor.

Kein Service weiß (oder muss wissen), wer seine Events konsumiert. Jeder Service hat eine klar definierte Verantwortung und einen klar definierten Abschluss. Jedes Event ist ein Fakt, keine Aufforderung.

Daten dorthin bringen, wo die Entscheidung fällt

Im orchestrierten Modell gibt es oft einen Schritt, in dem der Koordinator Daten von einem Service abfragt, bevor er eine Entscheidung trifft. Zum Beispiel: Der Koordinator fragt den *InventoryService*, ob genügend Bestand vorhanden ist. In dieser Query steckt eine Laufzeitabhängigkeit. Der anfragende Service muss warten, bis die Antwort kommt. Fällt der angefragte Service aus, steht der anfragende Service still. Genau diese Laufzeitkopplung wollten wir mit EDA vermeiden.

In einer Choreografie lösen wir das anders. Statt Daten zur Laufzeit abzufragen, bringen wir die Daten dorthin, wo Entscheidungen fallen. Der *InventoryService* publiziert Events über Bestandsänderungen (*InventoryUpdated*). Andere Services, die diese Informationen benötigen, konsumieren diese Events und halten lokal eine Kopie der für sie relevanten Daten vor.

Im Bestellprozess könnte der *OrderService* zum Beispiel eine lokale Bestandsübersicht pflegen und offensichtlich aussichtslose Bestellungen direkt ablehnen, wenn ein Produkt als nicht verfügbar gekennzeichnet ist. Diese lokale Sicht ist allerdings „eventually consistent“, also nicht in jedem Moment aktuell. So könnten zwei Kunden gleichzeitig die letzte Einheit eines Produkts bestellen, und in der lokalen Sicht würde bei beiden noch „verfügbar“ angezeigt werden. Die verbindliche Entscheidung über die Reservierung trifft daher der *InventoryService*, der die autoritative Datenhoheit über den Bestand hat. Die lokale Sicht im *OrderService* dient also als Vorabfilter, nicht als Garantie.

Daten können und sollen über Kontextgrenzen hinweg repliziert werden, um Laufzeitabhängigkeiten zu vermeiden. Für das Verhalten gilt das nicht: Jeder Service ist für sein eigenes Verhalten verantwortlich. Entscheidungen fallen innerhalb einer Domäne, im zuständigen Service.

Micro-Workflows

Ein häufiger Einwand lautet: „Das funktioniert vielleicht für ein triviales Beispiel, aber nicht für unsere komplexen Geschäftsprozesse.“ Hier stoßen zwei ganz grundsätzlich verschiedene Ansätze aufeinander: Ein komplexes Problem mit komplexer Technologie zu lösen oder das Problem in mehrere, einfachere Probleme zu zerlegen, für die es dann entsprechend einfachere Lösungen gibt.

Der Schlüssel liegt darin, nicht zu versuchen, einen großen Workflow eins zu eins in eine Choreografie zu übersetzen, sondern ihn aufzubrechen. So wie Monolithen in einigen Projekten in Microservices zerlegt werden, müssen wir auch unsere Workflows in Micro-Workflows zerlegen.

Jeder Service hat seinen eigenen kleinen Workflow: Der *Payment-Service* ist beispielsweise dafür verantwortlich, eine Zahlung abzuwickeln. Dieser Prozess kann intern durchaus mehrere Schritte umfassen (Autorisierung, Capture, Bestätigung). Das ist jedoch eine interne Angelegenheit des *PaymentService*. Nach außen kommuniziert er über Events: *PaymentAuthorized* oder *PaymentFailed*.

Die Zerlegung großer Workflows in Micro-Workflows ist nicht trivial und erfordert ein Umdenken. Es handelt sich dabei nicht primär um eine technische, sondern vielmehr um eine fachliche Aufgabe. Die Frage lautet nicht: „Wie implementiere ich diesen Workflow verteilt?“, sondern: „Wie verteile ich die Aufgaben so, dass jeder Service seinen Teil eigenständig erledigen kann?“ [7]

Kontrolle und Beobachtung trennen

Ein weiterer, gängiger Einwand gegen Choreografie betrifft die Sichtbarkeit: Wenn es keinen zentralen Koordinator gibt, woher weiß man, in welchem Zustand sich eine Bestellung befindet? Wie erhält man eine End-to-End-Sicht auf den Prozess?

Dazu ist festzuhalten, dass Kontrolle und Beobachtbarkeit nicht zwangsläufig direkt miteinander zusammenhängen. Es sind zwei verschiedene Anliegen, die nicht vermischt werden müssen.

- *Kontrolle bedeutet:* Wer entscheidet, was als Nächstes passiert? In einer Choreografie wird das bei der Entwicklung definiert und verteilt. Zur Laufzeit entscheidet jeder Service für sich.
- *Beobachtbarkeit bedeutet:* Wer kann den Gesamtzustand des Prozesses einsehen? Das ist eine separate Frage, die unabhängig von der Frage nach der Kontrolle gelöst werden kann. Eine Möglichkeit ist ein separater Service, wie in **Abbildung 5** dargestellt.

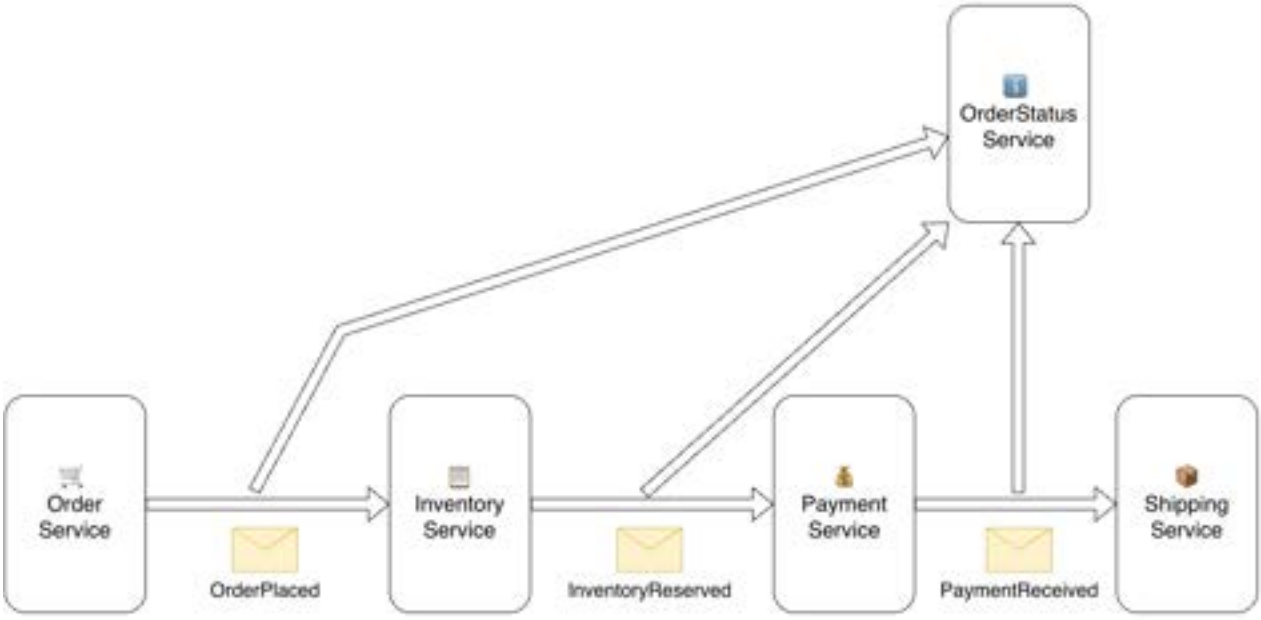


ABB 5 Sichtbarkeit durch einen separaten OrderStatusService

Der *OrderStatusService* hat keine Kontrolle über den Prozess. Er abonniert die relevanten Events und aggregiert den Status. Kunden, Supportmitarbeiter oder Dashboards können den Status abfragen. Fällt der *StatusService* aus, hat das keinerlei Auswirkungen auf den eigentlichen Bestellprozess.

Selbst Temporal.io, eine der populärsten Workflow-Engines, trennt intern die Ausführung der Workflows von der Abfrage des Status. Letzteres läuft über ein separates Subsystem (das „Visibility"-System, das auf Elasticsearch basiert). Die Trennung von Kontrolle und Beobachtbarkeit ist also kein radikaler Gedanke, sondern wird technisch auch in orchestrierten Systemen praktiziert.

Fehlerbehandlung ohne Orchestrator

In Teil 2 dieser Artikelserie haben wir ausführlich über die Fehlerbehandlung gesprochen, darunter blockierende und verzögerte Wiederholungen sowie Dead Letter Topics. All das bezog sich auf einen einzelnen Consumer. In einer Choreografie ändert sich daran nichts Grundsätzliches.

Wenn der *PaymentService* ein *InventoryReserved*-Event nicht verarbeiten kann, ist das sein Problem. Er muss es erneut versuchen, den Fehler loggen, bei Bedarf einen Alarm auslösen. Der *InventoryService* bekommt davon nichts mit und muss davon auch nichts mitbekommen.

Das ist keine Schwäche, sondern eine Stärke. Die Verantwortung ist eindeutig. Wenn eine Zahlung nicht funktioniert, ist das *PaymentService*-Team zuständig. Es ist keine Abstimmung darüber notwendig, ob der übergeordnete Workflow oder der Service den Fehler behandeln soll.

Timeouts

Ein weiterer Unterschied zwischen Orchestrierung und Choreografie ist die Herangehensweise bei Zeitüberschreitungen. Mit Workflow-Engines besteht die Möglichkeit, für die asynchrone Kommunikation erwartete Zeiten zu hinterlegen, sodass es eine Art Timeout-Überwachung gibt. Das klingt zunächst gut, trägt jedoch nur wenig zur Verhinderung von Zeitüberschreitungen bei. Oft dient es lediglich als Monitoring: Wenn ein Timeout auftritt, wissen wir, dass etwas nicht stimmt.

In einer Event-driven Architektur liegt die Zuständigkeit für rechtzeitige Verarbeitung beim jeweiligen Service. Der Service gibt ein Versprechen, was seine Leistungsfähigkeit angeht. Wenn sich Events stauen, muss der Service skalieren. Jeder Service muss daher überwachen, wie lange Events auf Verarbeitung durch ihn warten. Je nach eingesetzter Technologie erfolgt die Überwachung mit Hilfe der Anzahl nicht verarbeiteter Events (sogenannter Consumer Lag) oder der Wartezeiten (oldest message age). Steigt der Consumer Lag, muss der Service reagieren (in der Regel durch das Starten weiterer Instanzen).

Grundsätzlich sind alle Fälle, in denen ein bestimmter Zeitrahmen geschäftlich relevant ist (z. B. ein Container muss innerhalb von 48 Stunden abgeholt werden), durch Monitoringtools oder wiederum dedizierte Services wie unseren *OrderStatusService* abbildbar. Die dezentrale Herangehensweise erfordert allerdings ein Umdenken.

Und wenn Kompensation nötig ist?

Auch in einer Choreografie können kompensierende Aktionen erforderlich sein. Wenn die Zahlung fehlschlägt, muss das reservierte Inventar freigegeben werden. In der Choreografie publiziert der *PaymentService* ein *PaymentFailed*-Event. Der InventoryService konsumiert dieses Event und hebt die Reservierung auf.

Die kompensierende Logik liegt dort, wo sie hingehört: beim Service, der die ursprüngliche Aktion ausgeführt hat. Es wird kein zentraler Koordinator benötigt, der die Kompensation steuert. Der *InventoryService* weiß selbst am besten, wie eine Reservierung rückgängig gemacht wird.

Zusammenfassung: Orchestrierung oder Choreografie?

Orchestrierung und Choreografie sind keine gleichwertigen Alternativen, zwischen denen man je nach Geschmack wählen kann (Tabelle 2). Sie basieren auf grundlegend verschiedenen Kommunikationsmustern.

	Orchestrierung	Choreografie
Kommunikationsmuster	Request-Response (Command)	Fire-and-Forget (Event)
Prozesssteuerung	zentraler Koordinator	verteilt auf die Services
Verantwortung bei Fehlern	geteilt (Workflow + Service)	beim jeweiligen Service
Laufzeitkopplung	hoch (Koordinator wartet auf Antworten)	niedrig (Services agieren unabhängig)
Sichtbarkeit des Prozesses	im Workflow sichtbar	separat, durch Monitoring oder dedizierten Service
Passt zu EDA?	dedingt (Command-driven)	ja (Event-driven)

TABELLE 2 Orchestrierung und Choreografie im Vergleich

Orchestrierung ist im Kern Command-driven. Ein Koordinator gibt Befehle und wartet auf Antworten. Choreografie ist Event-driven. Services publizieren Fakten und reagieren auf Fakten. Damit soll nicht gesagt sein, dass Orchestrierung generell schlecht sei. Es kommt vielmehr darauf an, auf welcher Ebene sie eingesetzt wird.

Choreografie zwischen Subdomains und Teams

Auf der Ebene der Zusammenarbeit zwischen verschiedenen Subdomains und insbesondere zwischen verschiedenen Teams sollte Orchestrierung vermieden werden. Die oben beschriebenen Probleme – unklare Verantwortlichkeiten, organisatorische Reibung, Laufzeitkopplung – treffen hier mit voller Wucht zu.

Wenn Team A einen Workflow betreibt, der Team B und Team C koordiniert, dann sind die Teams nicht wirklich unabhängig voneinander. Prozessänderungen erfordern Abstimmung. Fehler müssen gemeinsam analysiert und Deployments koordiniert werden. All das widerspricht dem Ziel autonomer Teams mit eigenen Services.

Auf dieser Ebene ist Choreografie der geeignetere Ansatz: Jeder Service, jedes Team publiziert Events und reagiert auf Events. Die Verantwortung ist klar verteilt. Die Services sind zur Laufzeit unabhängig voneinander.

Orchestrierung innerhalb einer Subdomain

Auf einer tieferen Ebene, innerhalb einer Subdomain, kann Orchestrierung ein sinnvolles Werkzeug sein. Wenn ein Service intern einen Drittanbieter einbinden muss, etwa ein Zahlungs-Gateway, ein ERP-System oder ein Legacy-System, dann ist ein kleiner, lokaler Workflow dafür durchaus angemessen.

Der entscheidende Unterschied: Dieser Workflow lebt innerhalb eines Micro-Workflows, innerhalb einer Subdomain, unter der Verantwortung eines einzelnen Teams. Er koordiniert keine teamübergreifenden Prozesse, sondern löst ein technisches Integrationsproblem. Nach außen kommuniziert der Service weiterhin über Events. Die Tatsache, dass intern ein Workflow die Einbindung einer Drittanwendung koordiniert, ist ein Implementierungsdetail.

Die Empfehlung

Die Empfehlung lautet daher, nicht vorschnell zu Workflow-Engines zu greifen, wenn der Prozess mehrere Teams und Subdomains umspannt. Stattdessen sollte der Prozess hinterfragt werden. Lässt er sich in Micro-Workflows zerlegen? Können die Services ihre Aufgaben eigenständig erledigen? Kann die Verantwortung verteilt werden?

Innerhalb eines Micro-Workflows, innerhalb einer Subdomain ist Orchestrierung als technisches Hilfsmittel in Ordnung. Hier ist der Scope begrenzt, die Verantwortung eindeutig, und die Kopplung betrifft keine teamübergreifenden Grenzen.

Es geht also nicht um ein absolutes Entweder-oder, sondern um die richtige Zuordnung. Choreografie passt auf höhere Ebenen, auf denen Entkopplung und Teamautonomie zählen; Orchestrierung passt dort, wo sie als begrenztes technisches Mittel einen konkreten Zweck erfüllt.

To be continued …

In diesem dritten Teil haben wir uns mit der Frage beschäftigt, wie mehrere Services in einer Event-driven Architecture zusammenarbeiten. Wir haben Orchestrierung und Choreografie gegenübergestellt und argumentiert, dass Choreografie auf der Ebene zwischen Subdomains und Teams der natürliche Ansatz für EDA ist, während Orchestrierung innerhalb einer Subdomain als technisches Hilfsmittel ihre Berechtigung hat.

Im nächsten und letzten Teil widmen wir uns dem Betrieb und der Evolution Event-getriebener Systeme: Wie gehen wir mit Schema-Versionierung um? Wie stellen wir Rückwärtskompatibilität sicher? Und wie behalten wir in einem wachsenden System den Überblick?

Links & Literatur

[1] Beispielapplikation zur Artikelserie: https://codeberg.org/lutzh/javamagazin-eda-series-2026

[2] https://temporal.io

[3] https://camunda.com

[4] https://cloud.google.com/workflows

[5] https://www.cncf.io/projects/serverless-workflow

[6] https://learn.microsoft.com/de-de/azure/architecture/patterns/compensating-transaction

[7] https://www.codecentric.de/wissens-hub/blog/wer-microservices-richtig-macht-braucht-keine-workflow-engine-und-kein-bpmn

Event-driven Architecture: eine Einführung – Teil 4

Events meistern: Betrieb und Evolution Event-getriebener Systeme

Sobald Event-getriebene Systeme produktiv laufen, beginnen die eigentlichen Betriebsfragen: Schemas ändern sich, neue Consumer brauchen historische Daten, Consumer Lag muss überwacht werden und Legacy-Systeme bleiben selten außen vor. Der vierte Teil des Whitepapers zeigt, wie sich EDA-Systeme über ihren Lebenszyklus betreiben, beobachten und weiterentwickeln lassen.

In den ersten drei Serienteilen haben wir unser System aufgebaut: Wir haben Events publiziert, sie konsumiert und betrachtet, wie wir Prozesse, die mehrere Services einschließen, abbilden. Bis hierhin haben wir die Entwicklung des Systems betrachtet. Doch wenn unsere Software einmal produktiv läuft, hat sie noch ein langes Leben vor sich. Schemas entwickeln sich weiter, neue Services kommen hinzu.

Der vierte und letzte Teil behandelt deshalb die Herausforderungen nach der initialen Entwicklung: Betrieb und Evolution Event-getriebener Systeme. Wir sehen uns an, wie sich Event-Schemas weiterentwickeln lassen, ohne die Kommunikation zu unterbrechen. Außerdem geht es darum, neue Services nachträglich anzubinden und das System beobachtbar und skalierbar zu machen. Zudem werfen wir einen kurzen Blick auf die Integration mit der Außenwelt.

Schema-Evolution

Wir möchten unsere Services unabhängig voneinander deployen können, ohne den laufenden Betrieb zu stören. Wenn wir das Event-Schema in einem Producer erweitern, wollen wir das so tun, dass bestehende Consumer die Events weiterverarbeiten können, ohne ein Update zu benötigen. Das ist mit REST APIs vergleichbar, bei denen eine Änderung des Servers bestehende Clients auch nicht unbrauchbar machen sollte.

In der Event-driven Architecture stehen wir aber noch vor einer weiteren Herausforderung. Anders als HTTP Payloads, die nach dem Request-Response-Zyklus verschwinden, können Events in einem logbasierten Broker lange leben. Sie werden möglicherweise erst Tage, Wochen oder Monate später erneut gelesen. Das heißt in einigen Fällen: Wenn wir einen Consumer updaten, können wir nicht davon ausgehen, dass er nur die aktuelle Version der Events verstehen muss.

In der EDA-Welt hat sich die folgende Terminologie etabliert:

- *Backward Compatibility* bedeutet, dass ein Consumer, der bereits auf ein neues Schema aktualisiert wurde, auch noch alte Events verarbeiten kann. Das ist relevant, wenn zuerst Consumer aktualisiert werden und der Producer später nachzieht. Es gilt auch, wenn wir alte Events aus dem Log erneut lesen.
- *Forward Compatibility* bedeutet das Gegenteil: Bestehende Consumer, die auf dem alten Schema laufen, können trotzdem neue Events lesen. Das ist die Eigenschaft, auf die wir bei einer Änderung am Producer typischerweise abzielen, denn sie erlaubt uns, diesen zu aktualisieren, ohne sofort alle Consumer mitzuziehen.
- *Full Compatibility* verbindet beides: Jede Kombination aus alten und neuen Producern und Consumern funktioniert.

Wann immer möglich, sollten wir Breaking Changes, also Änderungen, die die nicht aktualisierten Services funktionsunfähig machen, vermeiden. Welche Änderungen das genau sind, hängt auch vom Serialisierungsformat ab: Avro, Protobuf und JSON haben jeweils eigene Regeln. Als Faustregel gilt: Hinzufügen ist in Ordnung, Ändern oder Entfernen nicht.

Praxis: ein Feld hinzufügen

Nehmen wir unser *OrderPlaced*-Event aus den vorherigen Serienteilen. Bisher enthält es Bestellnummer, Kundennummer und eine Liste von Artikeln. Nun wollen wir ein optionales Feld *promoCode* ergänzen, weil das Marketingteam eine Aktion mit Rabattcodes plant. Die Schema-Erweiterung in Avro zeigt Listing 1.

Listing 1: Avro-Schema

```
{
  "type": "record",
  "name": "OrderPlaced",
  "namespace": "com.example.orders",
  "fields": [
    { "name": "orderId", "type": "string" },
    { "name": "customerId", "type": "string" },
    { "name": "items", "type": { "type": "array", "items": "OrderItem" } },
    { "name": "promoCode", "type": ["null", "string"], "default": null }
  ]
}
```

Entscheidend ist hier zweierlei: Das neue Feld ist als Union mit *null* deklariert, und es hat einen Defaultwert. Damit gilt die Änderung in Avro als „forward compatible“. Ein alter Consumer, der das Feld nicht kennt, kann das neue Event lesen. Er sieht das Feld schlicht nicht, weil sein Reader-Schema es nicht enthält.

In Protobuf, gezeigt in Listing 2, ist das Hinzufügen optionaler Felder noch einfacher, weil Protobuf von Haus aus alle Felder optional behandelt.

Listing 2: Protobuf-Schema

```
message OrderPlaced {
  EventEnvelope envelope = 1;
  string order_id = 2;
  string customer_id = 3;
  repeated OrderItemContract items = 4;
  string promo_code = 5;  // neu
}
```

Wichtig ist, dass die Tagnummer (hier *5*) für dieses Feld einmalig vergeben und nie wiederverwendet wird, auch nicht für ein anderes Feld in einer späteren Version. Alte Consumer ignorieren unbekannte Tags und lesen das Event weiter, neue Consumer können das Feld auswerten.

Stolperstein Enum-Werte

Ein Fall, der oft zu Verwirrung führt, ist das Hinzufügen neuer Enum-Werte. Streng genommen ist das eine inkompatible Änderung, denn ein Consumer, der ein Enum als abgeschlossen behandelt, kann mit einem unbekannten Wert nicht umgehen. Abhängig vom Format wird auch schon die Deserialisierung nicht funktionieren.

Avro behandelt Enums strikt: Sendet der Producer einen Wert, der im Reader-Schema nicht existiert, schlägt die Deserialisierung fehl.

Protobuf verfolgt einen anderen Ansatz: Wenn aus dem Schema eine Java-Klasse generiert wird, wird auch ein zusätzlicher Enum-Wert *UNRECOGNIZED* generiert, der für unbekannte Werte zurückgeliefert wird. Damit nichts kaputtgeht, muss der Code damit umgehen können. Das ist jedenfalls das Verhalten für die Kombination Protobuf 3 und Java – für Protobuf 2 und für andere Programmiersprachen gelten andere Regeln.

Glücklicherweise gibt es Werkzeuge, die uns vor solchen Fehlern bewahren können. Für Protobuf prüft das Kommandozeilentool buf [2] Schemas auf inkompatible Änderungen. Schema-Registries, auf die wir gleich noch zurückkommen, bieten ebenfalls Kompatibilitätsprüfungen.

Breaking Changes ausrollen

Trotz aller Disziplin werden wir früher oder später eine inkompatible Änderung vornehmen müssen. Vielleicht haben sich unsere Erkenntnisse über die Domäne so weiterentwickelt, dass eine andere Struktur deutlich besser passt. Vielleicht haben wir uns über Jahre eine Sammlung von deprecated Feldern eingehandelt, die wir endlich loswerden wollen.

Ein bewährter Ansatz: Wir behandeln Topics wie HTTP APIs ihre Endpunkte. So wie REST oft eine Versionsangabe im Pfad trägt (*/api/v1/orders versus /api/v2/orders*), versehen wir auch Topic-Namen mit einer Version. Die goldene Regel lautet: Ein Client funktioniert weiter, solange er auf demselben Topic lauscht. Für die inkompatible Änderung wird ein neues Topic eingeführt, beide laufen während einer Übergangsphase parallel. Der Rollout vollzieht sich in vier Schritten:

1. Alle Konsumenten lauschen zunächst auf *order-events-v1*.
2. Der Producer beginnt, parallel auch auf *order-events-v2* zu publizieren, mit dem neuen Format.
3. Die Konsumenten migrieren nach und nach von *v1* auf *v2*.
4. Wenn alle umgestellt sind, kann *v1* abgeschaltet werden.

In der Praxis ist das doppelte Publizieren am Producer manchmal aufwendig. Wenn die neuen Events strukturell ein Superset der alten sind, hilft ein Kniff: Der Producer publiziert nur noch das neue Format auf dem *v2*-Topic. Ein temporärer Service abonniert *v2*, transformiert jedes Event in das alte Format und publiziert es auf *v1*. Unser Hilfs-Service agiert als Single Message Transformer (jedes Event wird einzeln und unabhängig transformiert, ohne dass ein Event-übergreifender Zustand vorgehalten wird). Sobald alle Consumer migriert sind, wird der Transformer abgeschaltet.

So oder so sind solche Migrationen anstrengend und langwierig. Zudem verdoppelt sich das Datenvolumen während der Übergangsphase. Das sollte Motivation genug sein, inkompatible Änderungen wo immer möglich zu vermeiden.

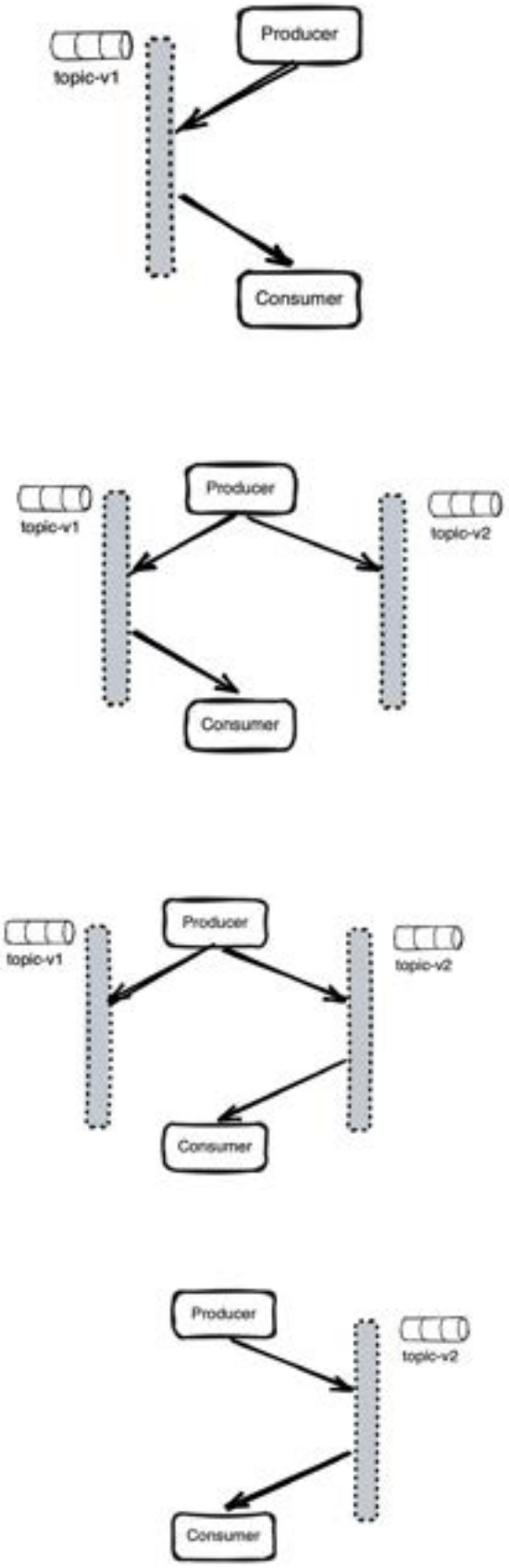


ABB. 1 Rollout einer inkompatiblen Schema-Änderung in vier Schritten

Schema-Registry

Wer mit vielen Schemas arbeitet, sollte sie zentral ablegen, damit andere sie leicht finden und verwenden können. Eine solche zentrale Stelle ist ein durchsuchbarer Katalog aller Eventtypen im System. In größeren Organisationen mit vielen Teams ist das ein nicht zu unterschätzender Vorteil. Das kann ein dediziertes Quellcode-Repository sein oder eine Schema-Registry.

Bei einem codebasierten Ansatz liegen die Schemas im Versionskontrollsystem, typischerweise als *.avsc*- oder *.proto*-Dateien. Producer und Consumer haben das Schema zur Compile-Zeit zur Verfügung. Kompatibilitätsprüfungen laufen in der CI/CD Pipeline und brechen den Build, wenn jemand eine inkompatible Änderung einreicht. Das ist operationell schlank, weil keine zusätzliche Infrastruktur betrieben werden muss.

Eine Schema-Registry ist ein zentraler Dienst, der Schemas verwaltet und ausliefert. Producer registrieren ihre Schemas zur Laufzeit, und jedes Schema bekommt eine eindeutige ID, die in jeder Nachricht mitgeschickt wird. Consumer lesen die ID, holen das passende Schema aus der Registry (mit lokalem Caching) und deserialisieren entsprechend. Die Registry kann zudem Kompatibilitätsregeln durchsetzen, indem sie die Registrierung eines inkompatiblen Schemas ablehnt.

Im Kafka-Ökosystem ist die Confluent Schema Registry der bekannteste Vertreter, daneben gibt es weitere, wie z. B. das offene Apicurio [3]. Bei Google Pub/Sub ist die Schemaverwaltung direkt in den Broker integriert, eine separate Registry entfällt [4].

Die beiden Ansätze schließen sich gegenseitig nicht aus, auch ein Hybrid ist möglich: Schemas leben im Quellcode, werden in CI/CD geprüft, und eine Registry stellt sie zur Laufzeit für Discovery und Validierung bereit. Damit hat man beides, die Versionierung über Git und die zentrale Sicht zur Laufzeit.

Neue Services im laufenden Betrieb

Eine häufige Herausforderung bei einem langlaufenden, Event-getriebenen System ist das Hinzufügen neuer Services. Ein neuer Service hat ein Latecomer-Problem: Von seinem Start an kann er Events konsumieren und verarbeiten. Aber um konsistent zu sein, müsste er auch wissen, was vor seinem Start passiert ist.

Stellen wir uns vor, unser Bestellsystem bekommt einen neuen *RecommendationService*. Er soll auf Basis vergangener Bestellungen Empfehlungen aussprechen. Damit das funktioniert, braucht er Zugriff auf die Bestellhistorie, nicht nur auf Bestellungen, die ab seinem Deployment-Zeitpunkt eingehen.

Wenn Kafka verwendet wird, ist der erste Gedanke oft: „Dann lassen wir ihn einfach das Topic von Anfang an replayen.“ Das funktioniert nur, wenn das Topic eine ausreichend lange oder gar unbegrenzte Retention hat. In den meisten Set-ups ist das nicht der Fall, denn der Broker ist ein Transporter, kein dauerhafter Speicher.

Wir brauchen also einen anderen Weg, den neuen Service mit historischen Daten zu versorgen. Dieser Vorgang heißt Bootstrapping oder Backfill. Ein gängiger Ansatz dafür ist ein temporäres Backfill Topic. Dazu wird die Historie als Folge von Events auf ein dediziertes Topic geschrieben. Der neue Service konsumiert es vollständig und schaltet danach auf den Livestream um. Sobald der Backfill abgeschlossen ist, kann das temporäre Topic gelöscht werden. Wichtig: Es handelt sich hier um Snapshot-Events, sie sollen den Service auf den Stand bringen, aber keine Geschäftsaktionen auslösen.

Andere Möglichkeiten sind Dateiexport und -import, oder, für kleine Datenmengen, ein HTTP Endpoint, von dem die Daten abgerufen werden können. Backfills liegen nicht jeden Tag an, aber häufiger, als man denkt. Schon bei der Entwicklung eines Services lohnt die Frage, wie er später hinzukommende Services lückenlos mit seinen Daten versorgen kann.

Observability für Event-getriebene Systeme

Verteilte Systeme im Allgemeinen und Event-getriebene Systeme im Besonderen sind ohne ordentliche Observability nicht zu betreiben. Die übliche Einteilung in Logs, Metrics und Traces gilt auch hier, allerdings mit eigenen Akzenten.

Logs bleiben wertvoll, um detailliert nachvollziehen zu können, was innerhalb eines Services passiert ist. Strukturierte Logs (etwa JSON mit konsistenten Feldnamen) lassen sich gut filtern und korrelieren. Eine Empfehlung: Volle Event-Payloads nicht loggen; das kann schnell teuer werden und ist aus Datenschutzsicht problematisch. Stattdessen loggen wir die Event-ID, den Typ und den Message Key. Wer die Payload braucht, kann sie anhand der ID aus dem Broker holen.

Metrics liefern eine aggregierte Sicht auf Durchsatz, Latenz, Fehlerraten und ein Maß, das in EDA besonders wichtig ist: den Consumer Lag. Mehr dazu im nächsten Abschnitt.

Traces sind das Werkzeug, um über Service-Grenzen hinweg zu verfolgen, was mit einem einzelnen Vorgang geschieht. In synchronen Systemen werden Trace-Informationen über HTTP-Header propagiert. In Event-getriebenen Systemen geschieht dasselbe über Message-Header.

Metadaten und Header

In der Nachricht selbst sollten wir einen Metadatenblock vorsehen. Dort hinein gehören die fachlichen Identifier, die uns helfen, Events in einen Geschäftskontext einzuordnen, zum Beispiel:

- *Correlation ID:* Verknüpft alle Events, die zu einem fachlichen Vorgang gehören. In unserem Beispiel etwa die *orderId*, die sich durch *OrderPlaced*, *InventoryReserved*, *PaymentAuthorized* und *OrderShipped* zieht. Eine Bestellung kann sich über mehrere technische Traces erstrecken (eine HTTP-Anfrage zum Aufgeben, eine spätere zum Ändern, ein Scheduled Job für den Versand), die Correlation ID hält alles fachlich zusammen.
- *Causation ID:* Verweist auf das unmittelbar vorausgehende Event, das dieses Event ausgelöst hat. Damit lässt sich nicht nur erkennen, dass Events zusammengehören, sondern auch in welcher Kausalkette sie stehen.

Daneben gibt es Header, die als Key-Value-Paare an der Nachricht hängen. Header lassen sich lesen, ohne den Body zu deserialisieren, und genau das macht sie für bestimmte Zwecke wertvoll.

In Header gehört, was Infrastruktur und Tooling brauchen, bevor die Nachricht überhaupt bei der Anwendung ankommt: Trace-Information in *traceparent* und *tracestate*, eventuell der Event-Typ, um danach zu filtern, und ähnliche Routing- oder Observability-Attribute. Diese Werte werden von OpenTelemetry, Jaeger, Zipkin, Datadog und anderen Werkzeugen automatisch erkannt, sofern man bei den standardisierten Headernamen bleibt, statt eigene zu erfinden.

Eine Nachricht sollte für die Anwendung vollständig sein und nicht von Headerwerten abhängen, die durch Broker-Konfigurationen oder Bibliotheksversionen verloren gehen können. Innerhalb des Bodys hat sich eine Aufteilung in zwei Bereiche bewährt: Metadaten und Payload. In unserem Beispiel haben wir für die Metadaten einen einheitlichen Block definiert (Listing 3).

Listing 3: Typische Metadaten in einer Nachricht

```
{
  "metadata": {
    "eventId": "evt-789",
    "eventType": "PaymentAuthorized",
    "version": "1.1",
    "timestamp": "2026-04-29T12:00:00Z",
    "source": "PaymentService",
    "correlationId": "order-123",
    "causationId": "evt-456"
  },
  "payload": { ... }
}
```

Praktische Empfehlungen

Ein paar Dinge, die sich in der Praxis bewährt haben:

- Instrumentieren mindestens an den Grenzen, also beim Publizieren und beim Konsumieren. Das gibt schon einen guten Überblick über den Fluss, ohne jede Codezeile mit Tracing-Aufrufen zu durchsetzen.
- Die Observability vor dem Ernstfall testen: Test-Events durch das System schicken und prüfen, ob der Trace vollständig zusammenläuft, bevor ein Incident auftritt.

Ein kurzer Hinweis zu Dead Letter Queues: Wir haben sie in Teil 2 als Mittel der Fehlerbehandlung besprochen. Aus Operations-Sicht gilt: Eine Dead Letter Queue, deren Wachstum niemand bemerkt, ist schlimmer als gar keine. Alerting auf Queue-Größe und Tooling, mit dem Events nach einem Bugfix wieder eingespeist werden können, gehören zwingend dazu.

Skalieren: Consumer Lag als Steuersignal

Eines der Versprechen von EDA ist unabhängige Skalierbarkeit. Da Services zur Laufzeit entkoppelt sind, kann jeder auf seine eigene Last reagieren. Aber woran erkennen wir, dass skaliert werden muss, und wie automatisieren wir das?

In Request-Response-Systemen sind die üblichen Skalierungssignale CPU-Auslastung, Memory-Druck oder Request-Latenz. Diese Metriken sind auch in EDA noch nützlich, aber sie übersehen das Wesentliche: ob der Consumer mit dem ankommenden Eventstrom mitkommt.

Bei Kafka ist der Consumer Lag das eigentliche Signal, also die Differenz zwischen dem letzten Offset im Topic und dem zuletzt vom Consumer committeten Offset. Ein Lag von null bedeutet, dass der Consumer alles verarbeitet hat. Ein wachsender Lag bedeutet, dass Events schneller ankommen, als sie verarbeitet werden. Bei Message Queues ist es entsprechend die Anzahl der ungelesenen Nachrichten in einer Queue.

Im Kubernetes-Ökosystem hat sich für diese Aufgabe Kubernetes Event-Driven Autoscaling (KEDA) etabliert. KEDA stellt Scaler bereit, die Lag-Metriken aus Kafka, Pub/Sub und anderen Quellen lesen und Pods entsprechend hoch- und runterfahren, bis hinunter auf null Instanzen, wenn keine Last anliegt. Damit wird das Skalieren auf Lag-Basis zu einem Standardwerkzeug statt zu einer Eigenbaulösung.

Eine Einschränkung sollte man kennen: In Kafka ist die Skalierungseinheit die Partition. Mehr Consumer-Instanzen als Partitionen helfen nicht, denn überzählige Instanzen bekommen schlicht keine Partitionen zugeteilt. Wer hochskalieren können will, muss bei der Topic-Einrichtung eine ausreichende Partitionsanzahl vorsehen.

Integration mit Legacy- und Drittsystemen

Kein nicht triviales System steht für sich allein. Es gibt Drittanbieter, Legacy-Systeme, Partner sowie externe APIs. Und nur ein Bruchteil davon kommuniziert über Events. Wie integrieren wir diese Welten, ohne unsere Event-getriebene Architektur zu verwässern?

Hilfreiche Leitbilder sind Event-driven Core und Request-Response Shell: Im Inneren des Systems kommunizieren die Services über Events. An den Rändern, dort wo wir mit der Außenwelt interagieren, dürfen wir auch Request-Response sprechen.

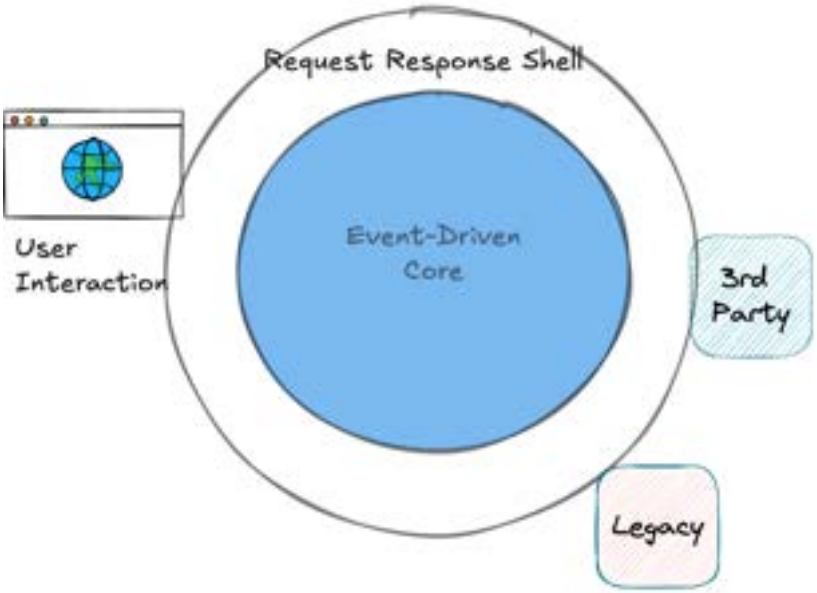


ABB. 2 Event-driven Core und Request-Response Shell: Synchrone Kommunikation findet an den Rändern statt, der Kern bleibt Event-getrieben

Ausgehende Aufrufe an externe Systeme

Wenn unser System einen externen HTTP oder gRPC Endpoint aufrufen muss, kapseln wir diesen Aufruf in einen Service – etwa für ein Zahlungs-Gateway oder ein ERP-System. Er wird durch ein internes Event ausgelöst, macht den synchronen Aufruf nach außen und publiziert das Ergebnis als Event zurück in den Kern. Der Rest des Systems sieht nur Events; der synchrone Aufruf ist auf eine kleine Insel beschränkt.

Wenn das externe System komplexere Konversationen verlangt, etwa mehrere Schritte mit Korrelationen, führt das zurück zu Teil 3: einem lokalen Workflow innerhalb einer Subdomain. Eine Workflow-Engine kann hier ihre Berechtigung haben, denn sie ist genau für diese Art zustandsbehafteter Konversation gebaut. Nach außen kommuniziert der kapselnde Service trotzdem weiterhin über Events. Dass intern ein Workflow läuft, ist ein Implementierungsdetail.

Legacy-Systeme als Eventquelle

Schwieriger als der ausgehende Fall ist meist der eingehende: Wir wollen Änderungen in einem Legacy-System als Events in unseren Kern bekommen. Da gibt es im Wesentlichen drei Wege mit deutlich unterschiedlichen Charakteristika.

Der erste, konzeptionell sauberste Weg ist Application Layer Emit: Wir modifizieren die Legacy-Anwendung selbst, sodass sie bei relevanten Zustandsänderungen Events publiziert. In der Praxis ist das selten realistisch. Legacy-Systeme sind oft nicht modifizierbar, schwer testbar oder bekommen Updates aus Quellen, von denen die Anwendungsschicht nichts weiß (Batch-Jobs, manuelle SQL-Statements, andere Integrationen).

Der zweite Weg ist klassisches Change Data Capture (CDC), etwa mit Debezium [5]. Wir hängen uns an das Transaktionslog der Datenbank und bekommen jede Zeile mit, die sich ändert. Der Vorteil: Es ist keine Änderung am Legacy-System nötig und wir bekommen wirklich jede Mutation. Der Nachteil: Wir erhalten Tabellenstrukturen und Spaltennamen statt Geschäftsereignisse. Wenn wir CDC ungefiltert verwenden, koppeln wir unsere Event-Verträge an Datenbankinterna. Hier ist zusätzlich unbedingt ein Anti-Corruption Layer notwendig, der die Datenbankänderungen in unsere Event-Sprache übersetzt. Das ist aber nicht immer einfach umzusetzen, da sich die Events oft nicht aus einer einzelnen Änderung ergeben.

Daher gewinnt ein dritter Weg gerade an Aufmerksamkeit: Semantic Change Capture. Werkzeuge wie Drasi [6] erlauben es, Abfragen über eine oder mehrere Tabellen zu definieren und Änderungen am Ergebnis dieser Abfragen als Events zu publizieren. Damit lässt sich die Lücke zwischen Datenbankinterna und Domänensprache schon an der Integrationsgrenze überbrücken.

Day Two

Wer schon einmal ein System nicht nur gebaut, sondern über Jahre betrieben hat, weiß: Die Entwicklung war der kleinere Teil. Viele Herausforderungen kommen erst mit der Zeit. Schemas wachsen, neue Konsumenten tauchen auf, Legacy-Systeme müssen integriert werden und die Last ändert sich.

EDA hat hier eine eigene Charakteristik. Die zeitliche Entkopplung, die wir in Teil 1 als Vorteil eingeführt haben, bedeutet auch: Events, die wir heute publizieren, leben möglicherweise länger als der Code, der sie erzeugt hat. Das macht Schema-Evolution und Versionierung wichtiger als in einer reinen Request-Response-Welt. Gleichzeitig zahlt sich genau diese Entkopplung aus, wenn wir neue Services hinzufügen oder bestehende ersetzen wollen, denn sie können sich unabhängig ins System einklinken.

Die Investitionen, die wir in den ersten drei Teilen besprochen haben, zahlen sich in dieser Lebensphase aus: Aussagekräftige Event-Namen aus der gemeinsamen Sprache erleichtern jede spätere Diskussion. Eine saubere Verteilung von Verantwortlichkeiten zwischen Producer und Consumer macht eine Fehleranalyse möglich. Ein choreografischer Ansatz erlaubt es Teams, ihre Services unabhängig weiterzuentwickeln.

Manche Themen haben wir in dieser Serie bewusst nicht behandelt. Event-Sourcing etwa, also das Prinzip, den Zustand eines Systems vollständig aus seiner Event-Historie wiederherzustellen, ist ein eigenes großes Feld, das eine eigene Behandlung verdient. Es nutzt viele der hier vorgestellten Konzepte, dient aber einem ganz anderen Ziel und spielt sich innerhalb eines Services ab, nicht in der Kommunikation zwischen Services.

EDA ist kein Selbstzweck, sondern ein Werkzeug, um Teams und Services über die Zeit voneinander zu entkoppeln. Wer das verinnerlicht, hat die wichtigste Entscheidung schon getroffen, bevor die erste Zeile Code geschrieben ist. Der Rest, von der Transactional Outbox über die Choreografie bis zur Schema-Evolution, ist Handwerk. Vielleicht ein etwas aufwendiges, aber lohnendes Handwerk.

Damit endet diese Serie. In vier Teilen haben wir uns von der einzelnen Komponente, die Events publiziert, über das Konsumieren von Events und das Zusammenspiel mehrerer Services bis zum Betrieb über Jahre hinweg vorgearbeitet. Wer all diese Themen einmal durchdacht hat, hat ein solides Fundament, um Event-getriebene Systeme nicht nur zu bauen, sondern auch zu betreiben und weiterzuentwickeln.

Lutz Hühnken ist ein erfahrener Softwarearchitekt und Manager mit über zwei Jahrzehnten Erfahrung in verteilten Systemen und Messaging-Technologien. Heute begleitet er als unabhängiger Berater Organisationen unter anderem durch die technischen und organisatorischen Herausforderungen bei der Einführung von Event-driven Architecture.

https://www.huehnken.de

Links & Literatur

[1] Beispielapplikation zur Artikelserie: https://codeberg.org/lutzh/javamagazin-eda-series-2026

[2] buf: https://github.com/bufbuild/buf

[3] Apicurio Registry: https://github.com/apicurio/apicurio-registry

[4] Schemas in Pub/Sub: https://docs.cloud.google.com/pubsub/docs/associate-schema-topic

[5] Debezium: https://debezium.io

[6] Drasi: https://drasi.io

iSAQB ACCREDITED Training Provider

Softwarearchitekt:in in vier Wochen mit Zertifikat

Grundlagen der Softwarearchitektur verstehen
Tragfähige Architekturen entwickeln
Dokumentation meistern & im Team anwenden

NEU

Das Software Architecture Camp Foundation FLEXIBLE

- Lerne, wann & wo du willst – auch neben dem Job
- Baue Praxis-Skills mit echtem Expertenfeedback auf
- Werde fit für Projekte und Prüfung

Software Architecture CAMP Foundation Flexible

14. September – 9. Oktober 2026 | online
9. November – 4. Dezember 2026 | online

software-architecture-camp.de

Präsentiert von: entwickler.de akademie entwickler.de