11 Rezept: Messaging und Kafka

Dieses Kapitel zeigt die Integration von Microservices mithilfe einer Message-oriented Middleware (MOM). Eine MOM verschickt Nachrichten und stellt sicher, dass die Nachrichten beim Empfänger ankommen. MOMs sind asynchron. Sie implementieren also kein Request/Reply wie bei synchronen Kommunikationsprotokollen, sondern verschicken nur Nachrichten. Es gibt MOMs mit unterschiedlichen Eigenschaften wie hoher Zuverlässigkeit, niedriger Latenz oder hohem Durchsatz. MOMs haben eine lange Historie. Sie stellen die Basis zahlreicher geschäftskritischer Systeme dar.

Dieses Kapitel vermittelt:

- Zunächst zeigt der Text einen Überblick über die verschiedenen MOMs und ihre jeweiligen Unterschiede. So kann der Leser entscheiden, welche MOM seinen Anwendungsfall am besten unterstützt.
- Die Einführung in Kafka zeigt, warum Kafka für ein Microservices-System besonders geeignet ist und wie Event Sourcing (siehe Abschnitt 10.2) mit Kafka umgesetzt werden kann.
- Das Beispiel im Kapitel verdeutlicht auf Code-Ebene, wie ein Event-Sourcing-System mit Kafka praktisch aufgebaut werden kann.

11.1 Message-oriented Middleware (MOM)

Durch eine MOM werden Microservices entkoppelt. Ein Microservice schickt eine Nachricht an die MOM oder empfängt sie von der MOM. Dadurch kennen sich Sender und Empfänger nicht, sondern nur einen Kommunikationskanal. Service Discovery ist daher nicht notwendig: Sender und Empfänger finden sich durch den Kanal, auf dem sie Nachrichten austauschen. Ebenso ist eine Lastverteilung unkompliziert möglich: Wenn mehrere Empfänger sich für einen Kommunikationskanal registriert haben, dann kann eine Nachricht von einem der Empfänger verarbeitet werden und so die Last verteilt werden. Eigene Infrastruktur ist dafür nicht notwendig.

Allerdings ist eine MOM eine komplexe Software, durch die alle Kommunikation abgewickelt wird. Also muss die MOM hochverfügbar sein und einen

hohen Durchsatz bieten. MOMs sind generell sehr reife Produkte, aber die Sicherstellung einer ausreichenden Leistung auch unter widrigen Umständen erfordert viel Know-How beispielsweise bei der Konfiguration.

11.1.1 Spielarten von MOMs

In dem Bereich MOM spielen folgende Produkte eine Rolle:

- JMS (*https://jcp.org/aboutJava/communityprocess/final/jsr914/index.html*) (Java Messaging Service) ist eine standardisierte API für die Programmiersprache Java und Teil des Java-EE-Standards. Bekannte Implementierungen sind Apache ActiveMQ (*http://activemq.apache.org/*) oder IBM MQ (*http://www-03.ibm.com/software/products/en/ibm-mq*), das früher IBM MQSeries hieß. Es gibt allerdings noch viele weitere JMS-Produkte (*https://en.wikipedia.org/wiki/Java_Message_Service#Provider_implementations*). Java Application Server, die nicht nur das Web Profile sondern das vollständige Java-EE-Profil unterstützen, müssen eine JMS enthalten, sodass JMS oft schon ohnehin verfügbar ist.
- AMQP (*https://www.amqp.org/*) (Advanced Message Queuing Protocol) standardisiert keine API, sondern ein Netzwerk-Protokoll auf TCP/IP-Ebene. Das ermöglicht einen einfacheren Austausch der Implementierung. RabbitMQ (*https://www.rabbitmq.com/*), Apache ActiveMQ (*http://activemq.apache.org/*) und Apache Qpid (*https://qpid.apache.org/*) sind die bekanntesten Umsetzungen des AMQP-Standards. Auch bei AMQP gibt es weitere Alternativen (*https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol#Implementations*).

Darüber hinaus gibt es beispielsweise ZeroMQ (*http://zeromq.org/*), das keinem der Standards entspricht. MQTT (*http://mqtt.org/*) ist ein Messaging-Protokoll, das vor allem im Bereich Internet of Things eine Rolle spielt.

Es ist mit allen diesen MOM-Technologien möglich, ein Microservices-System aufzubauen. Gerade wenn eine bestimmte Technologie schon genutzt wird und Wissen über den Umgang vorhanden ist, kann eine Entscheidung für eine bekannte Technologie sinnvoll sein. Ein Microservices-System ist aufwendig zu betreiben. Der Einsatz einer bekannten Technologie ermöglicht eine willkommene Reduktion von Risiko und Aufwand. Die Anforderungen an Verfügbarkeit und Skalierbarkeit der MOM sind hoch. Eine bekanntes MOM kann helfen, diese Anforderungen auf einfache Art zu erfüllen.

11.2 Die Architektur von Kafka

Im Microservices-Umfeld ist Kafka (*https://kafka.apache.org/*) eine interessante Alternative. Neben typischen Zielen wie hohem Durchsatz und niedriger Latenz

kann Kafka durch Replikation den Ausfall einzelner Server kompensieren und mit einer größeren Anzahl Server skalieren.

11.2.1 Kafka speichert die Nachrichten-Historie.

Vor allem ist Kafka dazu in der Lage, auch eine umfangreiche Record-Historie zu speichern. Üblicherweise zielen MOMs darauf ab, Nachrichten an die Empfänger auszuliefern. Danach löscht die MOM die Nachricht, da diese die Zuständigkeit der MOM verlassen hat. Das spart Ressourcen. Es bedeutet aber auch, dass Ansätze wie Event Sourcing (siehe Abschnitt 10.2) nur möglich sind, wenn jeder Microservice die Event-Historie selber speichert. Kafka hingegen speichert Records dauerhaft. Kafka kann auch große Datenmengen bewältigen und auf mehrere Server verteilt werden. Außerdem hat Kafka Möglichkeiten für Stream Processing. Dabei transformieren Anwendungen die Records in Kafka.

11.2.2 Kafka: Lizenz und Committer

Kafka steht unter der Apache-2.0-Lizenz. Diese Lizenz räumt den Nutzern weitgehende Freiheiten ein. Das Projekt ist in der Apache Software Foundation organisiert, die mehrere Open-Source-Projekte verwaltet. Viele Committer arbeiten für die Firma Confluent, die auch kommerziellen Support, eine Kafka-Enterprise-Lösung und eine Lösung in der Cloud anbietet.

11.2.3 APIs

Kafka bietet für die drei unterschiedlichen Aufgaben eine MOM jeweils eine eigene API an:

- Die *Producer API* dient zum Senden von Daten.
- Den Empfang von Daten ermöglicht die *Consumer API*.
- Schließlich gibt es die *Streams API* zum Transformieren der Daten.

Kafka ist in Java geschrieben. Die APIs werden mit einem sprachneutralen Protokoll angesprochen. Es gibt Clients (*https://cwiki.apache.org/confluence/display/KAFKA/Clients*) für viele Programmiersprachen.

11.2.4 Records

Kafka organisiert Daten in *Records*. Sie enthalten den transportierten Wert als *Value*. Kafka behandelt den *Value* als eine Black Box und interpretiert die Daten nicht. Außerdem haben *Records* einen Schlüssel (*Key*) und einen Zeitstempel (*Timestamp*). Kafka speichert darüber hinaus keine Metadaten zu den Records. Es gibt also keine weiteren Header, wie dies bei vielen anderen MOMs der Fall ist.

Ein Record könnte eine neue Bestellung oder eine Änderung an einer Bestellung sein. Der Key kann dann aus der Identität des Records und der Änderung zusammengesetzt werden.

11.2.5 Topics

Topics fassen Records zusammen. Wenn sich Microservices in einem ECommerce-System für neue Bestellungen interessieren oder neue Bestellungen anderen Microservices bekannt machen wollen, könnten sie dazu einen Topic nutzen. Neue Kunden wären ein anderer Topic.

11.2.6 Partitionen

Topics sind in *Partitionen* unterteilt. Wenn ein Producer einen neuen Record erstellt, wird er an eine Partition angehängt. Kafka speichert für jeden Consumer den Offset für jede Partition. Dieser Offset zeigt an, welchen Record in der Partition der Consumer zuletzt gelesen hat.

Partitionen erlauben Producern das Anhängen neuer Records. Producer profitieren davon, dass das Anhängen von Daten zu den effizientesten Operationen auf einem Massenspeicher gehört. Außerdem sind solche Operationen sehr zuverlässig und einfach zu implementieren.

Die Zuordnung von Records zu Partitionen erfolgt üblicherweise durch einen Hash des Keys. Man kann aber eine eigene Funktion implementieren.

Für Partitionen gilt, dass die Ordnung der Records erhalten bleibt: Die Reihenfolge, in der die Records in die Partition geschrieben werden, ist auch die Reihenfolge, in der Consumer die Records lesen. Über Partitionen hinweg ist die Ordnung nicht garantiert. Partitionen sind also auch ein Konzept für parallele Verarbeitung. Das Lesen in einer Partition ist linear, über Partitionen hinweg ist es parallel.

Mehr Partitionen haben verschiedene Auswirkungen (*http://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/*). Typisch sind Hunderte von Partitionen.

11.2.7 Commit

Wenn Consumer einen Record verarbeitet haben, committen sie einen neuen Offset. So weiß Kafka jederzeit, welche Records welcher Consumer bearbeitet hat und welche noch bearbeitet werden müssen. Natürlich können die Consumer Records committen, bevor sie tatsächlich bearbeitet sind. Dann kann es vorkommen, dass Records nicht bearbeitet werden.

Ein Consumer kann ein Batch von Records committen, wodurch eine bessere Performance möglich wird, weil weniger Commits notwendig sind. Dann können aber Duplikate auftreten. Das passiert, wenn der Consumer abstürzt, nachdem er

nur einen Teil eines Batches bearbeitet hat, aber den Batch noch nicht committet hat. Beim Neustart würde die Anwendung den kompletten Batch erneut lesen, weil Kafka beim letzten committeten Record wieder aufsetzt und damit am Anfang des Batches.

Kafka unterstützt auch Exactly once (*https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/*), also eine garantiert einmalige Zustellung.

11.2.8 Polling

Die Consumer pollen die Daten. Mit anderen Worten: Sie holen sich neue Daten ab und verarbeiten sie. Bei einem Push würden ihnen die Daten hingegen zugestellt werden. Polling scheint wenig elegant zu sein. Ohne einen Push der Records zu den Consumern sind die Consumer aber davor geschützt, unter zu viel Last zu geraten, wenn gerade eine große Zahl von Records geschickt wird und verarbeitet werden muss. Die Consumer können selbst entscheiden, wann sie die Records verarbeiten. Bibliotheken wie das im Beispiel verwendete Spring Kafka pollen im Hintergrund neue Records. Der Entwickler muss nur die Methoden implementieren, die Spring Kafka mit den neuen Records aufrufen muss. Das Polling ist dann vor dem Entwickler verborgen.

11.2.9 Records, Topics, Partitionen und Commits im Überblick

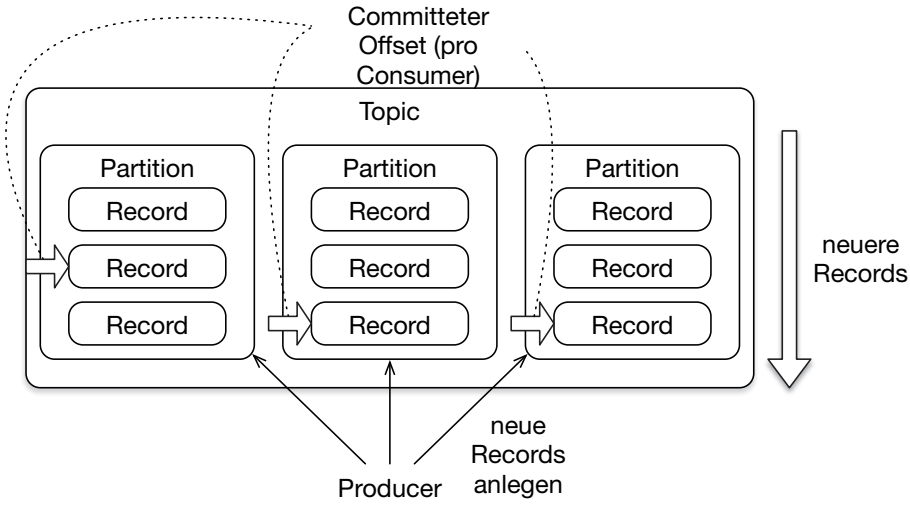

Abb. 11–1 *Partitionen und Topics in Kafka*

Abbildung 11–1 zeigt ein Beispiel: Der Topic ist in drei Partitionen aufgeteilt, die jeweils drei Records enthalten. Unten in der Abbildung befinden sich die neueren

Records, sodass dort der Producer neue Records anlegt. Der Consumer hat für die erste Partition den neuen Record noch nicht committet, aber für alle anderen Partitionen.

11.2.10 Replikation

Partitionen speichern Daten und können über Server verteilt werden. Jeder Server bearbeitet dann einige Partitionen. Das erlaubt Lastverteilung. Außerdem können die Partitionen repliziert werden. Dann liegen die Daten auf mehreren Servern. So kann Kafka ausfallsicher gemacht werden.

Die Anzahl »N« der Replicas kann konfiguriert werden. Beim Schreiben lässt sich festlegen, wie viele In-Sync-Replicas Änderungen committen müssen. Bei N=3 Replicas und zwei In-Sync-Replicas bleibt der Cluster einsatzfähig, wenn eines der drei Replicas ausfällt. Es kann dann nämlich immer noch auf zwei Replicas geschrieben werden. Beim Ausfall einer Replica gehen keine Daten verloren, da jedes Schreiben mindestens auf zwei Replicas erfolgreich gewesen sein muss. Selbst beim Verlust einer Replica müssen die Daten also noch mindestens auf einer weiteren Replica gespeichert sein. Kafka bietet damit bezüglich des CAP-Theorems verschiedene Kompromisse an (siehe Abschnitt 10.2).

11.2.11 Leader und Follower

Die Replikation ist so umgesetzt, dass ein Leader schreibt und die restlichen Replicas als Follower schreiben. Der Producer schreibt direkt an den Leader. Mehrere Schreiboperationen können in einem Batch zusammengefasst werden. Es dauert dann länger, bis ein Batch vollständig ist und die Änderungen tatsächlich abgespeichert werden. Dafür erhöht sich der Durchsatz, weil es effizienter ist, mehrere Datensätze auf einmal zu speichern.

11.2.12 Schreiben wiederholen

Wenn eine Schreiboperation nicht erfolgreich war, kann der Producer über die API festlegen, dass die Übertragung erneut versucht wird. Die Standardeinstellung ist, dass das Verschicken eines Records nicht wiederholt wird. Dadurch können Records verloren gehen. Wenn die Übertragung mehrfach erfolgt, kann es vorkommen, dass der Record trotz des Fehlers erfolgreich übertragen wurde. In diesem Fall würde es ein Duplikat geben, womit der Consumer umgehen können muss. Eine Möglichkeit ist es, den Consumer so zu entwickeln, dass er idempotente Verarbeitung anbietet. Damit ist gemeint, dass der Consumer in demselben Zustand ist, egal wie oft der Consumer einen Record verarbeitet (siehe Abschnitt 10.3). Beispielsweise kann der Consumer bei einem Duplikat feststellen, dass er den Record bereits bearbeitet hat und ihn ignorieren.

11.2.13 Consumer Groups

Consumer sind in Consumer Groups organisiert. Jede Partition hat genau einen Consumer in der Consumer Group. Ein Consumer kann für mehrere Partitionen zuständig sein.

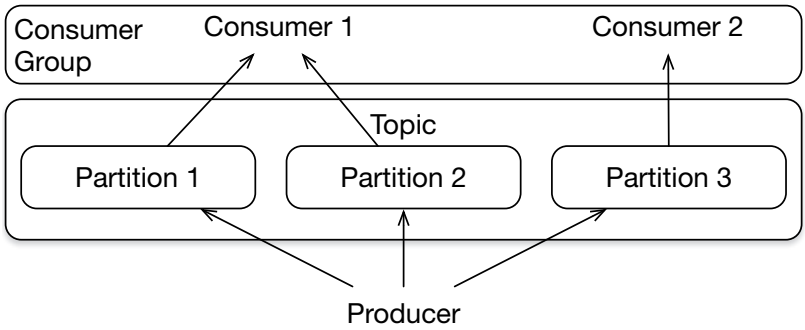

Abb. 11–2 *Consumer Groups und Partitionen*

Ein Consumer bekommt dann also die Nachrichten von einem oder mehreren Partitionen. Abbildung 11–2 zeigt ein Beispiel: Der Consumer 1 bekommt die Nachrichten aus den Partitionen 1 und 2. Der Consumer 2 erhält die Nachrichten aus Partition 3.

Wenn ein Consumer eine Nachricht mit einem Key bekommt, erhält er auch alle Nachrichten mit demselben Key, weil alle zur selben Partition gehören. Ebenso ist die Reihenfolge der Nachrichten pro Partition festgelegt. Somit können Records parallel bearbeitet werden und gleichzeitig ist die Reihenfolge bei bestimmten Records garantiert. Das gilt natürlich nur, wenn die Zuordnung stabil bleibt. Wenn zum Beispiel zur Skalierung neue Consumer zu der Consumer Group dazukommen, dann kann sich die Zuordnung ändern.

Die maximale Anzahl der Consumer in einer Consumer Group ist gleich der Anzahl der Partitionen, weil jeder Consumer mindestens für eine Partition verantwortlich sein muss. Idealerweise gibt es mehr Partitionen als Consumer, um bei einer Skalierung noch weitere Consumer hinzufügen zu können.

Wenn jeder Consumer alle Records aus allen Partitionen bekommen soll, dann muss es für jeden Consumer eine eigene Consumer Group mit nur einem Mitglied geben.

11.2.14 Persistenz

Kafka ist eine Mischung aus einem Messaging-System und einem Datenspeicher: Die Records in den Partitionen können von Consumern gelesen und von Producern geschrieben werden. Sie werden dauerhaft gespeichert. Die Consumer speichern lediglich ihren Offset.

Ein neuer Consumer kann daher alle Records verarbeiten, die jemals von einem Producer geschrieben worden sind, um so seinen eigenen Zustand auf den aktuellen Stand zu bringen.

11.2.15 Log Compaction

Allerdings führt das dazu, dass Kafka mit der Zeit immer mehr Daten speichern muss. Einige Records werden jedoch irgendwann irrelevant. Wenn ein Kunde mehrfach umgezogen ist, will man vielleicht nur noch den letzten Umzug als Record in Kafka halten. Dazu dient Log Compaction. Dabei werden alle Records mit demselben Key bis auf den letzten entfernt. Deswegen ist die Wahl des Keys sehr wichtig und muss aus fachlicher Sicht betrachtet werden, um auch mit Log Compaction noch alle relevanten Records verfügbar zu haben.

11.3 Events mit Kafka

Systeme, die über Kafka kommunizieren, können recht einfach Events austauschen (siehe auch Abschnitt 10.2):

- Die Records werden dauerhaft gespeichert. Also kann ein Consumer die Historie auslesen und so seinen Zustand wieder aufbauen. Dazu muss der Consumer die Daten nicht lokal speichern, sondern kann sich auf Kafka verlassen. Das bedeutet allerdings, dass alle relevanten Informationen im Record gespeichert sein müssen. Abschnitt 10.2 hat die Vor- und Nachteile dieses Ansatzes diskutiert.
- Wenn ein Event durch ein neues Event irrelevant wird, können die Daten durch Log Compaction von Kafka gelöscht werden.
- Durch Consumer Groups kann ein ausgewählter Consumer einen Record bearbeiten. Das vereinfacht die Zustellung beispielsweise, wenn eine Rechnung geschrieben werden soll. Dann sollte nämlich nur ein Consumer eine Rechnung schreiben und nicht mehrere Consumer parallel mehrere Rechnungen erstellen.

11.3.1 Events verschicken

Der Producer kann die Events zu unterschiedlichen Zeiten verschicken. Die einfachste Option ist es, das Event zu verschicken, nachdem die eigentliche Aktion stattgefunden hat. Also bearbeitet der Producer zunächst eine Bestellung, bevor er mit einem Event die anderen Microservices über die Bestellung informiert. Es kann dann vorkommen, dass der Producer zwar die Daten in der Datenbank ändert, aber das Event nicht verschickt, weil er beispielsweise zuvor abstürzt.

Der Producer kann die Events aber auch verschicken, bevor die Änderung an den Daten tatsächlich stattfindet. Wenn also eine neue Bestellung eintrifft, verschickt der Producer zunächst das Event, bevor er die Daten in der lokalen Datenbank speichert. Das ist nicht sonderlich sinnvoll, da Events eigentlich eine Information über ein Ereignis darstellen, das bereits passiert ist. Schließlich kann bei der Aktion ein Fehler auftauchen. Dann ist das Event schon verschickt, obwohl die Aktion nie stattgefunden hat.

Technisch hat das Verschicken der Events vor der eigentlichen Aktion außerdem den Nachteil, dass die eigentliche Aktivität verzögert wird, weil erst das Event verschickt wird, bevor die eigentliche Aktion stattfindet. Das kann zu einem Performance-Problem führen.

Es ist auch denkbar, die Events in einer lokalen Datenbank zu sammeln und in einem Batch zu schicken. Dann kann das Schreiben der geänderten Daten und das Erzeugen der Daten für das Event in der Datenbank in einer Transaktion erfolgen. Die Transaktion kann absichern, dass entweder die Änderung der Daten stattfindet und ein Event in der Datenbank erzeugt wird oder beides nicht stattfindet. Außerdem erreicht diese Lösung einen höheren Durchsatz, weil Batches in Kafka genutzt werden können. Allerdings ist die Latenz höher. Eine Änderung findet sich erst in Kafka wieder, wenn der nächste Batch geschrieben worden ist.

11.4 Beispiel

Das Beispiel in diesem Abschnitt orientiert sich an dem Beispiel für Events aus dem Abschnitt 10.2 (siehe Abbildung 11–3). Der Microservice *microservice-kafka-order* ist dafür zuständig, die Bestellung zu erfassen. Er schickt die Bestellungen an einen Kafka Topic. Zwei Microservices lesen die Bestellungen aus: Der Microservice *microservice-kafka-invoicing* schreibt für eine Bestellung eine Rechnung, der Microservice *microservice-kafka-shipping* liefert die bestellten Waren aus.

11.4.1 Datenmodell für die Kommunikation

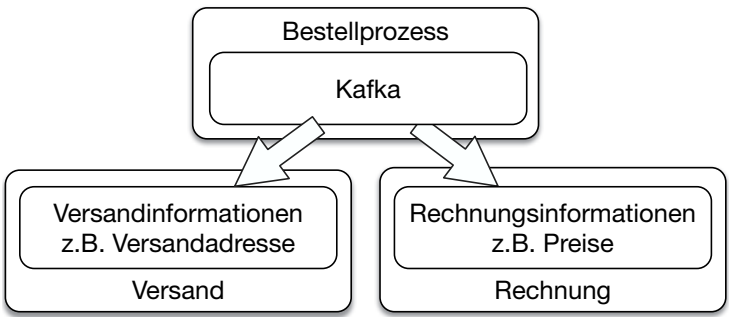

Abb. 11–3 *Beispiel für Kafka*

Die beiden Microservices *microservice-kafka-invoicing* und *microservice-kafka-shipping* benötigen unterschiedliche Informationen. Der Rechnung-Microservice benötigt die Rechnungsadresse und Informationen über die Preise der bestellten Waren. Der Versand-Microservices benötigt die Lieferadresse und kommt ohne Preise aus.

Beide Microservices lesen die notwendigen Informationen aus demselben Kafka Topic und denselben Records. Nur welche Daten sie aus den Records auslesen, unterscheidet sich. Das ist technische einfach möglich, weil die Daten über die Bestellungen als JSON ausgeliefert werden. Überflüssige Felder können die beiden Microservices einfach ignorieren.

11.4.2 Domain-Driven Design und Strategic Design

In der Demo sind die Kommunikation und die Konvertierung der Daten mit Absicht einfach gehalten. Sie entsprechen dem DDD-Pattern *Publish Language*. Es gibt ein standardisiertes Datenformat, aus dem alle Systeme die jeweils notwendigen Daten auslesen. Bei einer großen Anzahl an Kommunikationspartnern kann das Datenmodell unübersichtlich groß werden.

Dann könnte *Customer/Supplier* genutzt werden: Die Teams, die für Versand und Rechnungen zuständig sind, schreiben dem Bestellung-Team vor, welche Daten eine Bestellung enthalten muss, um den Versand und die Erstellung der Rechnung zu ermöglichen. Das Bestellung-Team stellt dann die notwendigen Daten bereit. Die Schnittstellen können dann sogar getrennt werden. Das scheint ein Rückschritt zu sein: Schließlich bietet Published Language eine gemeinsame Datenstruktur, die alle Microservices nutzen können. In Wirklichkeit ist es aber eine Vermischung der beiden Modelle, die zwischen Versand und Rechnung auf der einen Seite und Bestellung auf der anderen Seite festgelegt worden sind. Eine Trennung dieses einen Modells in zwei Modelle für die Kommunikationen zwischen Rechnung und Bestellung bzw. Lieferung und Bestellungen macht deutlich, welche Daten für welche Microservices relevant sind, und erleichtert es, die Auswirkungen von Änderungen abzuschätzen. Die beiden Datenmodelle können unabhängig voneinander weiterentwickelt werden. Das dient dem Ziel von Microservices, Software einfacher änderbar zu machen und die Auswirkungen einer Änderung zu begrenzen.

Die Patterns Customer/Supplier und Published Language kommen aus dem Strategic Design des Domain-Driven Design (DDD) (siehe Abschnitt 2.1).

11.4.3 Technische Umsetzung der Kommunikation

Technisch ist die Kommunikation folgendermaßen implementiert: Die Java-Klasse *Order* aus dem Projekt *microservice-kafka-order* wird in JSON serialisiert. Aus diesem JSON beziehen die Klassen *Invoice* aus dem Projekt *microser-*

vice-kafka-invoicing und *Shipping* aus dem Projekt *microservice-kafka-shipping* ihre Daten. Sie ignorieren Felder, die in den Systemen nicht benötigt werden. Die einzige Ausnahme sind die *orderLines* aus *Order*, die in *Shipping shippingLines* und in *Invoice invoiceLine* heißen. Für die Umwandlung gibt es eine Methode *setOrderLine()* in den beiden Klassen, um die Daten aus JSON zu deserialisieren.

11.4.4 Datenmodell für die Datenbank

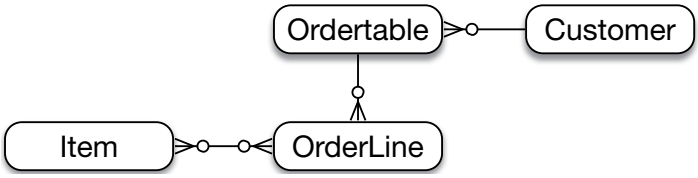

Abb. 11–4 *Datenmodellierung im System microservice-kafka-order*

Die Datenbank des Order-Microservice (siehe Abbildung 11–4) enthält eine Tabelle für die Bestellungen (Ordertable) und die einzelnen Posten in der Bestellungen (OrderLine). Waren (Item) und Kunden (Customer) haben ebenfalls eigene Tabellen.

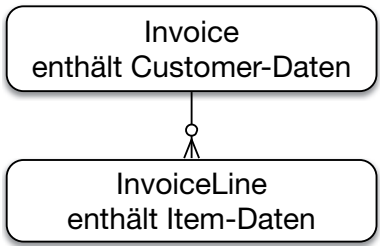

Abb. 11–5 *Datenmodellierung im System microservice-kafka-order*

Im Microservices *microservice-kafka-invoice* fehlen die Tabellen für Kunden und Waren. Die Daten der Kunden werden nur als Teil von *Invoice* gespeichert und die Daten der Waren als Teil der *InvoiceLine*. Die Daten in den Tabellen sind Kopien der Daten des Kunden und der Ware zu dem Zeitpunkt, als die Bestellung in das System übernommen wurde. Wenn also ein Kunde seine Daten ändert oder eine Ware den Preis ändert, beeinflusst das nicht die bisherigen Rechnungen. Das ist fachlich korrekt. Schließlich soll eine Preisänderung nicht auf bereits geschriebene Rechnungen durchschlagen. Diese Trennung der Änderungen kann sonst nur mit einer komplizierten Historisierung der Daten umgesetzt werden. Ebenfalls ist es mit dieser Modellierung sehr einfach, Rabatte oder Sonderangebote an die Rechnung zu übergeben. Es muss nur ein anderer Preis für eine Ware eingetragen werden.

Aus demselben Grund hat der Microservice *microservice-kafka-shipping* nur die Tabellen *Shipping* und *ShippingLine*. Daten für Kunden und Waren werden in diese Tabellen kopiert, so dass die Daten dort so hinterlegt sind, wie sie beim Auslösen der Lieferung waren.

Das Beispiel zeigt, wie Bounded Context die Modellierung vereinfacht.

11.4.5 Inkonsistenzen

Außerdem zeigt das Beispiel einen anderen Effekt: Die Informationen in dem System können inkonsistent sein. Bestellungen ohne Rechnungen oder Bestellungen ohne Lieferungen können vorkommen. Aber solche Zustände sind nicht dauerhaft. Irgendwann wird der Kafka Topic mit den neuen Bestellungen ausgelesen und die neuen Bestellungen erzeugen eine Rechnung und eine Lieferung.

11.4.6 Technischer Aufbau

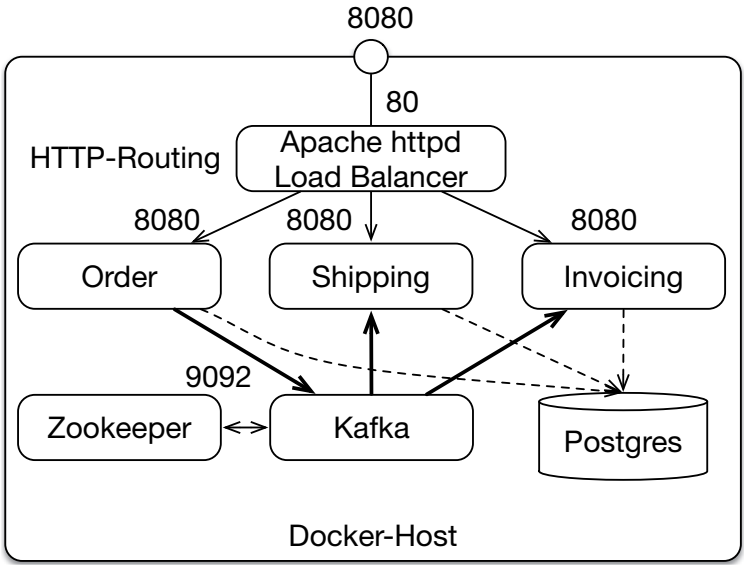

Abb. 11–6 *Überblick über das Kafka-System*

Abbildung 11–6 zeigt, wie das Beispiel technisch aufgebaut ist:

- Der *Apache httpd Load Balancer* verteilt eingehende HTTP-Requests. So kann es von jedem Microservice mehrere Instanzen geben. Das ist nützlich, um die Verteilung von Records auf mehrere Empfänger zu zeigen. Außerdem muss so nur der Apache-httpd-Server nach draußen sichtbar sein. Die anderen Microservices sind nur intern erreichbar.

- *Zookeeper* dient zur Koordination der Kafka-Instanzen und speichert unter anderem Informationen über die Verteilung der Topics und Partitionen. Das Beispiel nutzt das Image von *https://hub.docker.com/r/wurstmeister/zookeeper/*.
- Die *Kafka-Instanz* stellt die Kommunikation zwischen den Microservices sicher. Der Order-Microservice schickt die Bestellungen an die Shipping- und Invoicing-Microservices. Das Beispiel nutzt das Kafka-Image von *https://hub.docker.com/r/wurstmeister/kafka/*.
- Schließlich nutzen die Order-, Shipping- und Invoicing-Microservices dieselbe *Postgres-Datenbank*. Innerhalb der Datenbank-Instanz hat jeder Microservice ein eigenes, getrennte Datenbank-Schame. So sind die Microservices bezüglich der Datenbank-Schemas völlig unabhängig. Gleichzeitig reicht eine Datenbank-Instanz aus, um alle Microservices zu betreiben. Die Alternative wäre, jedem Microservice eine eigene Datenbank-Instanz zu geben. Das würde allerdings die Anzahl der Docker-Container erhöhen.

Im Abschnitt »Quick Start« in der Einleitung ist beschrieben, welche Software installiert sein muss, um das Beispiel zu starten.

Das Beispiel findet sich unter *https://github.com/ewolff/microservice-kafka*. Um es zu starten, muss man zunächst mit `git clone https://github.com/ewolff/microservice-kafka.git` den Code herunterladen. Anschließend muss im Verzeichnis `microservice-kafka` das Kommando `mvn clean package` ausgeführt werden. Dann muss im Verzeichnis `docker` erst `docker-compose build` zum Erzeugen der Docker-Images und `docker-compose up -d` zum Starten der Umgebung ausgeführt werden. Der Apache httpd Load Balancer steht unter Port 8080 zur Verfügung. Wenn Docker lokal läuft, findet man ihn also unter *http://localhost:8080/*. Von dort aus kann man den Order-Microservice nutzen, um eine Bestellung anzulegen. Die Microservices Shipping und Invoicing sollten nach einiger Zeit die Daten der Bestellung anzeigen.

Unter *https://github.com/ewolff/microservice-kafka/blob/master/WIE-LAUFEN.md* steht eine umfangreiche Dokumentation bereit, die Schritt für Schritt die Installation und das Starten des Beispiels erläutert.

11.4.7 Key für die Records

Kafka überträgt die Daten in Records. Jeder Record enthält eine Bestellung. Der Key ist die ID der Bestellung mit Zusatz `created`, also beispielsweise `1created`. Es reicht nicht aus, nur die ID der Bestellung zu nutzen. Bei einer Log Compaction werden alle Records mit demselben Key gelöscht, außer dem letzten Record. Für einen Order kann es verschiedene Records geben. Ein Record kann die Erstellung des Orders signalisieren und andere Records die verschiedenen Updates. Dann sollte der Key mehr als nur die ID des Orders enthalten, um so beim Log Com-

pacting alle Records zu einem Order zu behalten. Wenn der Key der ID des Orders entspricht, würde bei einer Log Compaction nur noch der letzte Record übrig bleiben.

Allerdings hat dieser Ansatz den Nachteil, dass Records zu einer Bestellung in unterschiedlichen Partitionen und Consumern landen können, weil sie unterschiedliche Keys haben. Also können beispielsweise Records für dieselbe Bestellung parallel bearbeitet werden, was zu Fehlern führen kann.

Um dieses Problem zu lösen, müsste eine eigene Funktion implementiert werden, die alle Records für eine Bestellung einer Partition zuweist. Eine Partition wird von einem einzigen Consumer verarbeitet und die Reihenfolge der Nachrichten ist in einer Partition garantiert. Somit kann also sichergestellt werden, dass alle Nachrichten für eine Bestellung von einem Consumer in der richtigen Reihenfolge verarbeitet werden.

11.4.8 Alle Informationen über die Bestellung im Record mitschicken

Eine Alternative wäre es, als Key doch nur die ID der Bestellung zu nutzen. Um dem Problem der Log Compaction zu entgehen, kann man in jedem Record den vollständigen Stand der Bestellung mitschicken, sodass ein Consumer seinen Zustand aus den Daten in Kafka rekonstruieren kann, obwohl nur der letzte Record für eine Bestellung nach der Log Compaction erhalten bleibt. Das setzt jedoch ein Datenmodell voraus, das für alle Consumer alle Daten enthält. So ein Datenmodell zu entwerfen, kann sehr aufwendig sein. Außerdem kann es sehr kompliziert und schwer wartbar werden.

11.4.9 Aufteilung der Records auf Partitionen selber implementieren

Außerdem ist es möglich, mit einem Partitioner (*https://kafka.apache.org/0110/javadoc/org/apache/kafka/clients/producer/Partitioner.html*) eigenen Code für die Aufteilung von Records auf die Partitionen zu schreiben. Dadurch könnte man alle Records, die fachlich zusammengehören, in dieselbe Partition schreiben und vom selben Consumer bearbeiten lassen, obwohl sie unterschiedliche Keys haben.

11.4.10 Technische Parameter der Partitionen und Topics

Der Topic `order` enthält die Order-Records. Docker Compose konfiguriert den Kafka-Docker-Container aufgrund der Umgebungsvariable `KAFKA_CREATE_TOPICS` in der Datei `docker-compose.yml` so, dass der Topic `order` angelegt wird.

Der Topic `order` ist in fünf Partitionen aufgeteilt. Mehr Partitionen erlauben mehr Parallelität. In dem Beispiel-Szenario ist ein hoher Grad an Parallelität unwichtig. Mehr Partitionen benötigen mehr File Handles auf dem Server und

mehr Memory auf dem Client. Beim Ausfall eines Kafka-Knotens muss gegebenenfalls für jede Partition ein neuer Leader gewählt werden, was länger dauert, wenn es mehr Partitionen gibt. Das spricht für eine eher geringe Zahl von Partitionen, wie sie im Beispiel genutzt werden, um so Ressourcen zu sparen.

Die Anzahl der Partitionen in einem Topic kann nach der Anlage des Topics noch geändert werden. Dann ändert sich jedoch die Zuordnung von Records zu Partitionen. Das kann zu Problemen führen, weil die Zuordnung von Records zu Consumern nicht mehr eindeutig ist. Also sollte die Anzahl der Partitionen von Anfang an ausreichend hoch gewählt werden.

11.4.11 Keine Replikation im Beispiel

Eine Replikation über mehrere Server ist für eine Produktionsumgebung notwendig, um den Ausfall einzelner Server zu kompensieren. Für eine Demo ist die dafür notwendige Komplexität übertrieben, sodass nur ein Kafka-Knoten läuft.

11.4.12 Producer

Der Order-Microservice muss die Informationen über die Bestellungen an die anderen Microservices schicken. Dazu nutzt der Microservice das `KafkaTemplate`. Diese Klasse aus dem Spring-Kafka-Framework kapselt die Producer API und erleichtert das Verschicken der Records. Nur die Methode `send()` muss aufgerufen werden. Das zeigt der Codeausschnitt aus der Klasse `OrderService` im Listing.

```
public Order order(Order order) {
  if (order.getNumberOfLines() == 0) {
    throw new IllegalArgumentException("No order lines!");
  }
  order.setUpdated(new Date());
  Order result = orderRepository.save(order);
  fireOrderCreatedEvent(order);
  return result;
}

private void fireOrderCreatedEvent(Order order) {
  kafkaTemplate.send("order", order.getId() + "created", order);
}
```

Hinter den Kulissen konvertiert Spring Kafka die Java-Objekte zu JSON-Daten mithilfe der Jackson-Bibliothek. Weitere Konfigurationen wie beispielsweise die Konfiguration der JSON-Serialisierung sind in der Datei `application.properties` im Java-Projekt zu finden. In `docker-compose.yml` sind Umgebungsvariablen für Docker Compose definiert, die Spring Kafka auswertet. Das sind vor allem der Kafka Host und der Port. So können mit einer Änderung an `docker-compose.yml` der Docker-Container mit dem Kafka-Server umkonfiguriert werden und die Producer so angepasst werden, dass sie den neuen Kafka-Host nutzen.

11.4.13 Consumer

Die Consumer werden ebenfalls in `docker-compose.yml` und mit dem `application.properties` im Java-Projekt konfiguriert. Spring Boot und Spring Kafka bauen automatisch eine Infrastruktur mit mehreren Threads auf, in der Records ausgelesen und bearbeitet werden können. Im Code findet sich nur in der Klasse `OrderKafkaListener` eine Methode, die mit `@KafkaListener(topics = "order")` annotiert ist:

```
@KafkaListener(topics = "order")
public void order(Invoice invoice, Acknowledgment acknowledgment) {
  log.info("Revceived invoice " + invoice.getId());
  invoiceService.generateInvoice(invoice);
  acknowledgment.acknowledge();
}
```

Ein Parameter der Methode ist ein Java-Objekt, das die Daten aus dem JSON im Kafka-Record enthält. Bei der Deserialisierung findet die Datenkonvertierung statt: Invoicing und Shipping lesen nur die Daten aus, die sie jeweils benötigen. Die restlichen Informationen werden ignoriert. Natürlich ist es in einem realen System auch denkbar, komplexere Logik als ein Ausfiltern der relevanten Felder zu implementieren.

Der andere Parameter der Methode ist vom Typ `Acknowledgement`. Damit kann der Record committet werden. Bei einem Fehler kann der Code das Acknowledgement verhindern. Dann würde der Record noch einmal verarbeitet werden.

Die Verarbeitung der Daten im Kafka-Beispiel ist idempotent. Wenn ein Record verarbeitet werden soll, wird zunächst überprüft, ob die Datenbank Daten enthält, weil der Record schon verarbeitet worden ist. Dann ist der Record offensichtlich ein Duplikat und wird nicht ein zweites Mal verarbeitet.

11.4.14 Consumer Groups

Die Einstellung `spring.kafka.consumer.group-id` in der Datei `application.properties` in den Projekten `microservice-kafka-invoicing` und `microservice-kafka-shipping` definiert die Consumer Group, zu der die Microservices gehören. Alle Instanzen von Shipping oder Invoicing bilden jeweils eine Consumer Group. Genau eine Instanz von Shipping oder Invoicing bekommt daher jeweils einen Record zugestellt. So ist gewährleistet, dass eine Bestellung nicht parallel von mehreren Instanzen verarbeitet wird.

Mit `docker-compose up --scale shipping=2` können mehr Instanzen des Shipping-Microservice gestartet werden. Wenn man mit `docker logs -f mskafka_shipping_1` sich die Log-Ausgaben einer Instanz ansieht, so kann man erkennen, welche Partitionen dieser Instanz zugewiesen sind und dass sich die Zuweisung ändert, wenn man mehr Instanzen startet. Ebenso sieht man beim Erzeugen einer neuen Bestellung, welche der Instanzen einen Record verarbeitet.

Es ist auch möglich, sich den Inhalt des Topics anzuschauen. Dazu muss man zunächst mit `docker exec -it mskafka_kafka_1 /bin/sh` eine Shell auf dem Kafka-Container starten. Der Befehl `kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic order --from-beginning` zeigt den vollständigen Inhalt des Topics an. Da die Microservices jeweils zu einer Consumer Group gehören und die verarbeiteten Records committen, bekommen sie nur jeweils die neuen Records. Eine neue Consumer Group würde aber tatsächlich alle Records noch einmal verarbeiten.

11.4.15 Tests mit Embedded Kafka

In einem JUnit-Test kann ein Embedded-Kafka-Server genutzt werden, um die Funktionalität der Microservices zu überprüfen. Dann läuft ein Kafka-Server in derselben JVM (Java Virtual Machine) wie der Test. Es muss also keine Infrastruktur für den Test aufgebaut werden und es ist auch nicht notwendig, die Infrastruktur nach dem Test wieder abzubauen.

Im Wesentlichen sind dazu zwei Dinge notwendig:

- Es muss ein Embedded Kafka gestartet werden. Dazu dient eine Variable wie

  ```
  public static KafkaEmbedded embeddedKafka = new KafkaEmbedded(1, true, "order");
  ```

  , die mit der JUnit-Annotation `@ClassRule` versehen wird. Dadurch startet JUnit den Kafka-Server vor den Tests und fährt ihn nach den Tests wieder herunter.

- Mit

  ```
  System.setProperty("spring.kafka.bootstrap-servers",
   embeddedKafka.getBrokersAsString());
  ```

  wird die Sping-Boot-Konfiguration so angepasst, dass Spring Boot den Kafka-Server nutzt. Dieser Code steht in einer Methode, die mit `@BeforeClass` annotiert ist, sodass sie vor den Tests ausgeführt wird.

11.4.16 Avro als Datenformat

Avro (*http://avro.apache.org/*) ist ein Datenformat, das zusammen mit Kafka (*https://www.confluent.io/blog/avro-kafka-data/*) und Big-Data-Lösungen aus dem Hadoop-Bereich recht oft genutzt wird. Avro ist ein binäres Protokoll, bietet aber auch eine JSON-basierte Repräsentation an. Avro-Bibliotheken gibt es beispielsweise für Python, Java, C#, C++ und C.

Avro hat ein Schema. Jeder Datensatz wird zusammen mit seinem Schema gespeichert oder übertragen. Zur Optimierung kann auch eine Referenz auf ein Schema aus einem Schema-Repository übertragen werden. Dadurch ist klar, welches Format die Daten haben. Das Schema enthält eine Dokumentation der Fel-

der. Das stellt sicher, dass die Daten langfristig interpretiert werden können und die Semantik der Daten klar ist. Außerdem können die Daten beim Lesen in ein anderes Format konvertiert werden. Das erleichtert die Schema-Evolution (*https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html*). Neue Felder können hinzugefügt werden, wenn Default-Werte definiert sind, sodass sich alte Daten in das neue Schema konvertieren lassen, indem der Default-Wert verwendet wird. Wenn Felder entfernt werden, kann ebenfalls ein Default-Wert angegeben werden, sodass neue Daten in das alte Schema konvertiert werden können. Auch die Reihenfolge der Felder lässt sich änderen, weil die Namen der Felder abgespeichert werden.

Vorteil der Flexibilität bei der Schema-Migration ist, dass sehr alte Records mit aktueller Software und dem aktuellen Schema verarbeitet werden können und auf einem alten Schema basierende Software neue Daten verarbeiten kann. Solche Anforderungen hat Message-oriented Middleware (MOM) typischerweise nicht, weil Nachrichten nicht lange gespeichert werden. Erst mit einer langfristigen Speicherung der Records wird die Schema-Evolution zu einer größeren Herausforderung.

11.5 Rezept-Variationen

Das Beispiel schickt in den Records die Daten für das Event mit. Dazu gibt es Alternativen (siehe auch Abschnitt 11.4):

- Man schickt immer den kompletten Datensatz mit, also die vollständige Bestellung.
- In den Records könnte nur ein Schlüssel des betroffenen Datensatzes stehen. Der Empfänger kann sich dann die Informationen über den Datensatz selber abholen.
- Für jeden Client gibt es ein eigenes Topic. Die Records haben jeweils eine an den Client angepasste Datenstruktur.

11.5.1 Andere MOM

Die Alternative zu Kafka wäre eine andere MOM, was beispielsweise bei umfangreicher Erfahrung mit einer anderen MOM die bessere Lösung sein kann. Kafka unterscheidet sich von anderen MOMs dadurch, dass Kafka die Records langfristig speichert. Das ist jedoch nur bei Event Sourcing relevant. Und selbst dann kann jedes System die Events selber speichern. Eine Speicherung in der MOM ist also nicht unbedingt notwendig. Sie kann sogar schwierig sein, weil sich die Frage nach dem Datenmodell stellt.

Ebenso kann asynchrone Kommunikation mit Atom (siehe Kapitel 12) implementiert werden. In einem Microservices-System sollte es nur eine Lösung für asynchrone Kommunikation geben, um den Aufwand nicht zu groß werden zu

lassen. Sowohl die Nutzung von Atom als auch von Kafka oder einer anderen MOM sollte also vermieden werden.

Kafka kann mit Frontend-Integration (siehe Kapitel 7) kombiniert werden. Die Ansätze arbeiten auf unterschiedlichen Ebenen, sodass eine gemeinsame Nutzung kein großes Problem darstellt. Eine Kombination mit synchronen Mechanismen (siehe Kapitel 13) erscheint weniger sinnvoll, da die Microservices entweder synchron oder asynchron kommunizieren sollten. Die Kombination kann sinnvoll sein, wenn in einigen Situationen synchrone Kommunikation notwendig ist.

11.6 Experimente

- Ergänze das System mit einem zusätzlichen Microservice:
  - Als Beispiel kann ein Microservice dienen, der dem Kunden abhängig vom Wert der Bestellung einen Bonus gutschreibt oder die Bestellungen zählt.
  - Natürlich kannst du einen der vorhandenen Microservcies kopieren und entsprechend modifizieren.
  - Implementiere einen Kafka Consumer für den Topic `order` des Kafka Servers `kafka`. Er soll dem Kunden einen Bonus bei einer Bestellung gutschreiben oder die Nachrichten zählen.
  - Außerdem sollte der Microservice eine HTML-Seite mit den Informationen (Anzahl Aufrufe oder Bonus der Kunden) darstellen.
  - Packe den Microservice in ein Docker-Image ein und referenziere ihn im `docker-compose.yml`. Dort kannst du auch den Namen des Docker-Containers festlegen.
  - Erzeuge im `docker-compose.yml` einen Link vom Container `apache` zum Container mit dem neuen Service und vom Container mit dem neuen Service zum Container `kafka`.
  - Der Microservice muss von der Homepage aus zugreifbar sein. Dazu musst du in der Datei `000-default.conf` im Docker-Container `apache` einen Load Balancer für den neuen Docker-Container erzeugen. Nutze dazu den Namen des Docker-Containers. Dann musst du im `index.html` einen Link zu dem neuen Load Balancer integrieren.
- Es ist möglich, mehr Instanzen des Shipping- oder Invoicing-Microservice zu starten. Dazu kann `docker-compose up -d --scale shipping=2` oder `docker-compose up -d --scale invoicing=2` dienen. Mit`docker logs mskafka_invoicing_2` kann man die Logs betrachten. Dort gibt der Microservice dann auch aus, welche Partitionen er bearbeitet.
- Kafka kann auch Daten transformieren. Dazu dient Kafka Streams. Recherchiere diese Technologie!
- Aktuell nutzt die Beispielanwendung JSON. Implementiere eine Übertragung mit Avro (*http://avro.apache.org/*). Ein möglicher Startpunkt dazu kann *https://*

www.codenotfound.com/2017/03/spring-kafka-apache-avro-example.html sein.

- Log Compaction ist eine Möglichkeit, überflüssig Records aus einem Topic zu löschen. Die Kafka Documentation (*https://kafka.apache.org/documentation/#compaction*) erläutert dieses Feature. Um Log Compaction zu aktivieren, muss es beim Erzeugen des Topic eingeschaltet werden. Siehe dazu *https://hub.docker.com/r/wurstmeister/kafka/*. Stelle das Beispiel so um, dass Log Compaction aktiviert ist.

11.7 Fazit

Kafka bietet einen interessanten Ansatz für die asynchrone Kommunikation zwischen Microservices:

11.7.1 Vorteile

- Kafka kann die Records dauerhaft speichern, was den Einsatz von Event Sourcing in einen Szenarien vereinfacht. Zusätzlich gibt es Ansätze wie Avro, um Probleme wie Schema-Evolution zu lösen.
- Der Overhead für die Consumer ist gering. Sie müssen sich nur die Position in jeder Partition merken.
- Mit Partitionen hat Kafka ein Konzept für die parallele Verarbeitung und mit Consumer Groups ein Konzept, um die Reihenfolge der Records für Consumer zu garantieren. So kann Kafka die Zustellung an einen Consumer garantieren und gleichzeitig die Arbeit auf mehrere Consumer verteilen.
- Kafka kann die Zustellung von Nachrichten garantieren. Der Consumer committet, welche Records er erfolgreich bearbeitet hat. Bei einem Fehler committet er den Record nicht und versucht erneut, ihn zu verarbeiten.

Daher lohnt sich gerade für Microservices der Blick auf diese Technologie, auch wenn andere asynchrone Kommunikationsmechanismen sicher ebenso sinnvoll einsetzbar sind.

11.7.2 Herausforderungen

Allerdings hält Kafka auch einige Herausforderungen bereit:

- Kafka ist eine zusätzliche Infrastruktur-Komponente. Sie muss betrieben werden. Gerade bei Messaging-Lösungen ist die Konfiguration oft nicht sehr einfach.
- Wenn Kafka als Speicher für die Events genutzt wird, müssen die Records alle Daten enthalten, die alle Clients benötigen. Das ist wegen Bounded Context (siehe Abschnitt 2.1) oft nicht leicht umzusetzen.