1 Einleitung

Diese Leseprobe haben Sie beim edv-buchversand.de heruntergeladen.
Das Buch können Sie online in unserem Shop bestellen.
Hier zum Shop

Sie haben einen soliden Abschluss in Raketenwissenschaft? Nein? Nun ja, falls Ihre Rakete jedoch den Namen Apache Kafka trägt, dann haben Sie nach der Lektüre dieses Buches einen. Versprochen!

Zuallererst möchten wir, die Autoren dieses Buches, Ihnen jedoch danken. Wir danken Ihnen, dass Sie sich in Sachen Apache Kafka weiterbilden möchten und sind uns sicher, dass Sie mit den kommenden Inhalten Höhenflüge erleben und anschließend auch erzeugen werden. Um gleich mal bei unserer Metapher mit der Rakete zu bleiben.

Es ist dabei ganz gleich, ob Sie vor dem Aufschlagen dieses Buches noch nie etwas über Apache Kafka gehört oder ob Sie schon einige Erfahrungen mit Kafka gesammelt haben. Oder ob Sie vielleicht sogar an einer unserer Schulungen teilgenommen haben. Wir sind uns in jedem Fall sicher, dass Sie in den kommenden Kapiteln viel Nützliches für sich mitnehmen können.

Liegt Ihre Expertise in der IT oder leiten Sie ein Team und möchten Apache Kafka dort einsetzen? Oder Sie tun es bereits und möchten nun verstehen, was Kafka konkret ist und wie Sie es bestmöglich nutzen können? Gratulation, denn auch Sie sind hier absolut richtig, denn Sie werden nach der Lektüre dieses Buches nicht nur einiges an Expertise gesammelt haben, um Kafka erfolgreich zu betreiben und einzusetzen, sondern auch, um Ihr Team dabei zu unterstützen, wenn es mit Fragen oder Sorgen rund um Kafka zu Ihnen kommt. Natürlich nur so lange, wie es überhaupt Sorgen und Probleme mit Kafka hat. Denn auch diese lassen sich für und mit Ihrem Team lösen.

Und selbstverständlich hoffen wir auch, dass Ihnen die Beschäftigung mit den Inhalten Spaß bereitet. Scheuen Sie sich bitte nicht, uns bei Fragen oder Anregungen zu kontaktieren. Sie erreichen uns per E-Mail unter

buch@streamcommit.de

oder im Web unter

http://streamcommit.de/buch.

1.1 Einführung in Apache Kafka

Aber was hat es denn nun überhaupt mit Apache Kafka auf sich? Diesem Programm, das quasi jeder namhafte (deutsche) Automobilhersteller einsetzt? Diese Software, dank derer wir mit unseren Schulungen und Workshops kreuz und quer durch Europa gereist sind und viele unterschiedliche Branchen – von Banken, Versicherungen, Logistik-Dienstleistern, Internet-Start-ups, Einzelhandelsketten bis hin zu Strafverfolgungsbehörden – kennengelernt haben? Warum setzen so viele unterschiedliche Unternehmen (und Behörden) Apache Kafka ein?

Wir teilen die Einsatzgebiete von Apache Kafka in zwei Kategorien ein: Die erste Gruppe von Anwendungsfällen ist die, für die Kafka einst gedacht war. Apache Kafka wurde nämlich anfangs bei LinkedIn entwickelt, um alle Events, die auf der LinkedIn-Website anfallen, in das zentrale Data Warehouse zu bewegen. LinkedIn war auf der Suche nach einem skalierbaren und auch bei sehr hoher Last performanten Messaging-System und hat letzten Endes Kafka dafür kreiert. So setzen heute sehr viele Unternehmen Kafka ein, um große Datenmengen von A nach B zu bewegen. Der Fokus liegt oft auf der benötigten Performance, der Skalierbarkeit Kafkas, aber natürlich auch auf der Zuverlässigkeit, die Kafka für die Zustellbarkeit und Persistierung der Nachrichten bietet.

Eine der Kernideen, die Kafka von klassischen Messaging-Systemen jedoch unterscheidet, ist, dass Kafka Daten auf Datenträgern persistiert. So können wir Daten in Kafka aufbewahren und einmal geschriebene Daten nicht nur mehrfach lesen, sondern auch Stunden, Tage oder gar Monate nachdem diese Daten geschrieben wurden.

Dies ermöglicht die zweite Kategorie der Apache-Kafka-Anwendungsfälle: Immer mehr Unternehmen nutzen Kafka als zentrales Werkzeug, um nicht lediglich Daten zwischen unterschiedlichsten Services und Anwendungen auszutauschen, sondern wenden Kafka als zentrales Nervensystem an, um mit Daten innerhalb von Unternehmen zu agieren.

Was aber meinen wir damit, dass Kafka das zentrale Nervensystem für Daten sein kann? Die Vision ist (wie in Bild 1.1), dass jedes Ereignis, das in einem Unternehmen stattfindet, in Kafka abgespeichert wird. Jeder andere Service (der natürlich die Berechtigung hat) kann nun auf dieses Ereignis asynchron reagieren und dieses Ereignis weiterverarbeiten.

Wir sehen zum Beispiel in vielen Unternehmen den Trend, dass es eine Trennung zwischen sogenannten *Altsystemen*, die für bestehende Geschäftsprozesse und Geschäftsmodelle essenziell sind, und der sogenannten *Neuen Welt* gibt, wo mit agilen Methoden neue Dienste entwickelt werden, die auch in Software abgebildet werden müssen. Oftmals setzen Unternehmen Kafka ein, um nicht nur als Schnittstelle zwischen alten und neuen Systemen zu agieren, sondern auch, um den neuen Services zu ermöglichen, untereinander Nachrichten auszutauschen.

Das liegt daran, dass Altsysteme oftmals nicht den neuen Anforderungen unserer Kund-Innen gewappnet sind. Batch-Systeme können den Drang nach Informationen, die immer „Sofort“ verfügbar sein sollen, nicht erfüllen. Wer möchte zum Beispiel heutzutage noch einen Tag oder gar mehrere Wochen darauf warten, dass sich der Kontostand nach einer Kreditkartentransaktion aktualisiert? Wir erwarten mittlerweile, dass wir unsere Pakete in Echtzeit verfolgen können. Moderne Pkw produzieren Unmengen von Daten, die insbesondere für die Vorbereitung auf das autonome Fahren an die Konzernzentrale geschickt und ausgewertet werden sollen. Kafka kann all diese Unternehmen dabei unterstützen, von

einer batch-orientierten Verarbeitung hin zu einer Datenverarbeitung in (nahezu) Echtzeit zu gelangen.

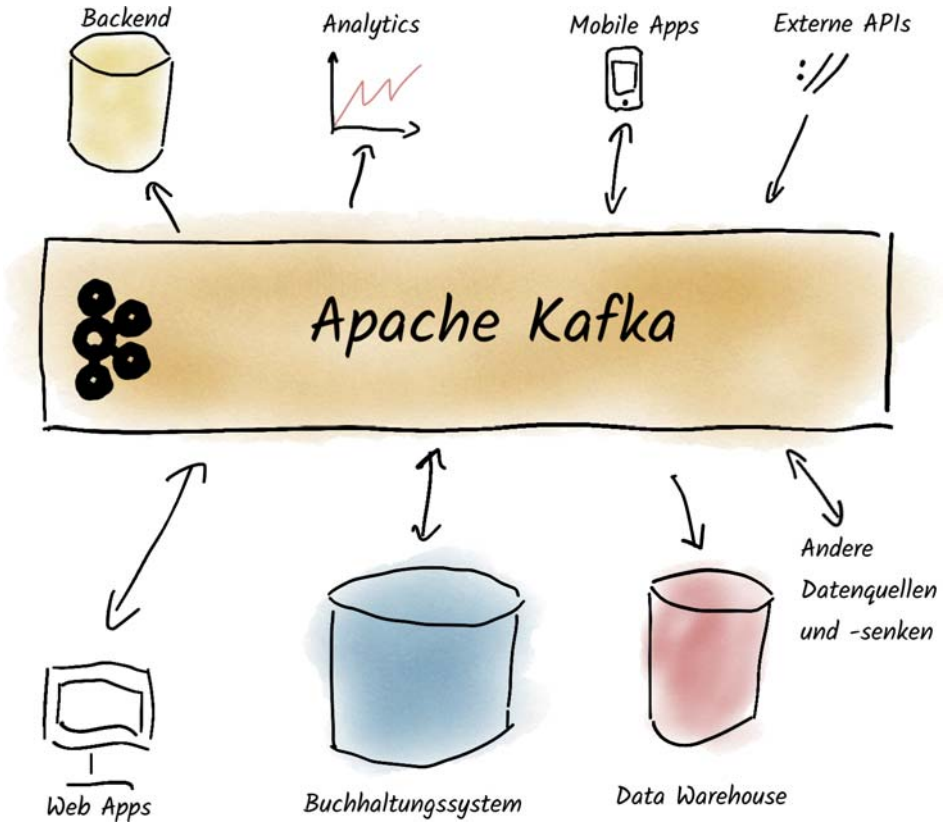

Bild 1.1 Kafka als zentrales Nervensystem für Daten im Unternehmen. Jedes Ereignis, welches im Unternehmen stattfindet, wird in Kafka gespeichert. Andere Services können auf diese Ereignisse asynchron reagieren und sie weiterverarbeiten.

Aber auch die Art und Weise, wie wir Software schreiben, verändert sich. Statt immer mehr Funktionalität in monolithische Services zu stecken und dann diese wenigen Monolithen miteinander mittels Integration zu verbinden, brechen wir unsere Services in *Microservices* auf, um unter anderem die Abhängigkeit zwischen den Teams zu reduzieren. Dafür benötigen wir jedoch eine Art und Weise für den Datenaustausch, die möglichst asynchron ist. Dadurch können Services, auch wenn ein Service gerade in Wartung ist, unabhängig von diesem weiter funktionieren. Wir benötigen Methoden für die Kommunikation, die es erlauben, dass sich die Datenformate in einem Service unabhängig von anderen Services entwickeln können. Auch hierbei kann uns Apache Kafka unterstützen.

Ein anderer Trend, der vor allem durch Virtualisierung und immer breiteren Einsatz von Cloud-Architekturen ausgelöst wurde, ist das Zurückgehen von Spezialhardware. Es gibt im Gegensatz zu anderen Messaging-Systemen keine *Kafka-Appliances*. Kafka läuft auf handelsüblicher Hardware und benötigt keine ausfallsicheren Systeme. Kafka selbst ist so ent-

wickelt, dass es gut mit Ausfällen von Teilsystemen zurechtkommt. Dadurch ist die Zustellung von Nachrichten zuverlässig, auch wenn in unserem Rechenzentrum vielleicht gerade Chaos ausbricht.

Wie aber erreicht Kafka diese Zuverlässigkeit und Performance? Wie können wir Kafka für unsere Anwendungsfälle verwenden, und was ist beim Betrieb Kafkas zu beachten? Die Antworten auf diese Fragen und vieles mehr möchten wir Ihnen auf der Reise durch dieses Buch geben.

Wir finden, dass sich Kafka auf gewisse Weise gut mit einer Rakete vergleichen lässt: Sie ist in aller Munde, ist zuverlässig, performant, aber auch bekannt für ihre komplexe Bauart. Wir werden die Raketenanalogie ein wenig ausreizen und Sie mit unserer Kafka-Rakete auf eine Mission mitnehmen.

Deswegen möchten wir in Kapitel 2 die Konzepte unserer Apache Kafka-Rakete vorstellen. Ziel einer jeden Rakete ist nämlich nie die Rakete selbst. Es geht vielmehr darum, unsere Nutzlast in den Orbit, zum Mond oder gar zum Mars zu befördern. Bei Kafka ist nie das Ziel, Kafka selbst einzusetzen, sondern unser Business zu unterstützen, indem wir Nachrichten zuverlässig und performant zwischen Systemen austauschen. Wie diese Nachrichten aussehen, wie wir diese Nachrichten einteilen und verschicken können, entdecken wir im Kapitel 2.1 „Payload“. Wie auch bei echten Raketen können wir Kafka nicht erfolgreich auf lange Zeit betreiben, wenn wir die Grundlagen und die Hintergründe des Systems nicht verstehen. Kafka ist nämlich ein verteiltes Log, und was dies bedeutet, das erfahren wir im Kapitel 2.2 „Kafka als verteiltes Log“. Dieses verteilte Log muss zwei Kernanforderungen erfüllen: Nachrichten zuverlässig zustellen – und dies darüber hinaus sehr schnell. Wir schauen uns das genauer in den Kapiteln 2.3 „Zuverlässigkeit“ und 2.4 „Performance“ an.

In Kapitel 3, nämlich dem Kafka Deep Dive, gehen wir weiter in die Tiefe. Wir sehen uns im Kapitel 3.1 „Verbindung zu Kafka“ an, wie unsere Kafka-Clients sich mit Kafka verbinden, im Kapitel 3.2 „Nachrichten produzieren und persistieren“, wie Nachrichten produziert und gespeichert werden. Im Kapitel 3.3 „Nachrichten konsumieren“ verdeutlichen wir, wie wir diese Nachrichten auch wieder lesen können. Wir setzen uns im Kapitel 3.4 „Nachrichten aufräumen“ auch damit auseinander, wie wir Nachrichten wieder löschen, und im Kapitel 3.5 „Cluster-Management“, wie Kafka als verteiltes System sich selbst koordiniert. Zu guter Letzt widmen wir uns in diesem Kapitel auch den Verarbeitungsgarantien und Transaktionen.

Im letzten Kapitel dieses Buches, „Kafka im Unternehmenseinsatz“, möchten wir darauf eingehen, wie Kafka erfolgreich in Unternehmen angewendet wird und was es abseits von Kafka für einen erfolgreichen Einsatz benötigt. Wir teilen mit Ihnen in diesem Kapitel auch einige Erfahrungen, die wir mit Kafka gemacht haben. Wir sehen uns im Kapitel 4.1 „Kafka-Ökosystem“ genauer das Ökosystem um Kafka an. Das heißt, wie wir Kafka mit anderen Systemen verbinden und Daten in Kafka verarbeiten. Für Architekten ist es immer wichtig zu wissen, wie sich Kafka mit anderen Systemen vergleicht. Das möchten wir im Kapitel 4.2 „Vergleich mit anderen Technologien“ schildern.

Zu guter Letzt haben wir im Kapitel 4.3 „Kafka-Referenzarchitektur“ noch weitere Empfehlungen für den Einsatz von Kafka gesammelt. Welche Hardware wird benötigt? Wo können

wir Kafka betreiben? Wie automatisieren wir den Betrieb? Die Antworten auf all diese Fragen werden besprochen.

Aber bevor wir uns in all die angekündigten Details stürzen, lassen Sie uns gemeinsam unsere Apache Kafka-Rakete starten und einen Blick darauf werfen, wie sie sich in einem ersten Testflug bewährt. Jede Raketenwissenschaft muss schließlich irgendwann einmal beginnen. Auch die Ihre.

In diesem Sinne: 3, 2, 1 … Zündung!

1.2 Erste Schritte mit Kafka

Bevor wir uns in die Details stürzen, starten wir gemeinsam unsere *Kafka-Rakete* und wagen einen ersten Testflug. Wir gehen davon aus, dass Kafka schon installiert ist. Wir haben die genaue Installationsanleitung im Anhang beschrieben.

Bevor wir ergründen, warum die Kafka-Rakete überhaupt fliegen kann und wie sie es tut, zünden wir gemeinsam die Triebwerke und machen einen kleinen Testflug, wie in Bild 1.2 dargestellt.

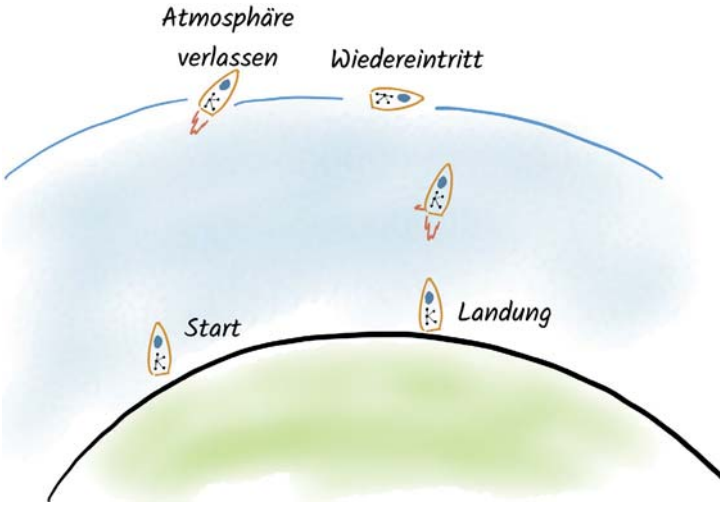

Bild 1.2 Unser Beispielflug mit unserer Apache Kafka-Rakete

Für unseren Flug ins All starten wir unsere (natürlich wiederverwendbare) Rakete, verlassen kurz die Atmosphäre, um den Wiedereintritt zu üben, und landen die Rakete sanft und sicher auf einem Landeplatz.

Wir möchten hierzu in Kafka alle Flugphasen erfassen und erstellen dazu als Erstes ein *Topic*. Topics sind insofern ähnlich zu Tabellen in Datenbanken, als dass wir in Topics eine Ansammlung von Daten zu einem bestimmten Thema abspeichern. In unserem Fall sind es Flugdaten, daher nennen wir das Topic entsprechend *flugdaten*:

```
$ kafka-topics.sh \
    --create \
    --topic flugdaten \
    --partitions 1 \
    --replication-factor 1 \
    --bootstrap-server
localhost:9092
Created topic flugdaten.
```

Mit dem Befehl `kafka-topics.sh` verwalten wir unsere Topics in Kafka. Hier weisen wir Kafka mit dem Argument `--create` an, das Topic `flugdaten` (`--topic flugdaten`) zu erstellen. Zunächst starten wir mit einer *Partition* (`--partitions 1`) und ohne die Daten zu replizieren (`--replication-factor 1`). Als Letztes geben wir an, zu welchem *Kafka-Cluster* sich `kafka-topics.sh` verbinden soll. In unserem Fall nutzen wir unser lokales Cluster, welches standardmäßig auf dem Port `9092` hört (`--bootstrap-server localhost:9092`). Der Befehl bestätigt uns die erfolgreiche Erstellung des Topics. Sollten wir hier Fehler bekommen, liegt dies oft daran, dass Kafka noch nicht gestartet und somit nicht erreichbar ist, oder daran, dass das Topic schon existiert.

Wir haben jetzt also einen Ort, an dem wir unsere Daten abspeichern können. Der Bordcomputer der Rakete schickt uns kontinuierlich Aktualisierungen zum Flugzustand der Rakete. Für unsere Simulation nutzen wir dafür das Kommandozeilenwerkzeug `kafka-console-producer.sh`. Dieser *Producer* und auch andere nützliche Werkzeuge werden mit Kafka direkt ausgeliefert. Der Producer verbindet sich zu Kafka, nimmt Daten von der Kommandozeile entgegen und schickt sie als Nachrichten in ein Topic (konfigurierbar über den Parameter `--topic`). Schreiben wir die Nachricht *`Countdown gestartet`* in unser eben erstelltes Topic *flugdaten*:

```
$ echo "Countdown gestartet" | kafka-console-producer.sh \
    --topic flugdaten \
    --bootstrap-server=localhost:9092
```

Unsere Bodenstation möchte diese Daten jetzt lesen und zum Beispiel auf einem großen Bildschirm ausgeben, damit wir sehen, ob die Rakete wirklich so funktioniert, wie wir das von ihr erwarten. Schauen wir uns also an, was bisher passiert ist. Um unsere eben gesendete Nachricht wieder zu lesen, starten wir den `kafka-console-consumer.sh`, welcher ebenfalls Teil der Kafka-Familie ist:

```
$ timeout 10 kafka-console-consumer.sh \
    --topic flugdaten \
    --bootstrap-server=localhost:9092
Processed a total of 0 messages
```

Wenn wir den `kafka-console-consumer.sh` starten, läuft der standardmäßig weiter, bis wir ihn aktiv (zum Beispiel mit **STRG+C**) abbrechen. Das wäre auch das gewünschte Verhalten, wenn wir den aktuellen Zustand der Rakete wirklich irgendwo anzeigen wollten. In unserem Beispiel nutzen wir den Befehl `timeout`, damit der Consumer automatisch nach spätestens zehn Sekunden abbricht. Bei dem Consumer müssen wir wieder angeben, welches Topic dieser benutzen soll (`--topic flugdaten`).

Etwas überraschend wird keine Nachricht angezeigt. Dies liegt daran, dass der `kafka-console-consumer.sh` standardmäßig am Ende des Topics zu lesen beginnt und nur neue Nachrichten ausgibt. Um auch bereits geschriebene Daten anzuzeigen, müssen wir das Flag `--from-beginning` benutzen:

```
$ timeout 10 kafka-console-consumer.sh \
    --topic flugdaten \
    --from-beginning \
    --bootstrap-server=localhost:9092
Countdown gestartet
Processed a total of 1 messages
```

Diesmal sehen wir die Nachricht *Countdown gestartet*! Was ist also passiert? Mit dem Befehl `kafka-topics.sh` haben wir das Topic *flugdaten* in Kafka erstellt und mit dem `kafka-console-producer.sh` die Nachricht *Countdown gestartet* produziert. Diese Nachricht haben wir dann mit dem `kafka-console-consumer.sh` wieder gelesen. Dieser Datenfluss ist in Bild 1.3 dargestellt. Ohne weitere Angaben fängt der `kafka-console-consumer.sh` immer am Ende an zu lesen, das heißt, wenn wir alle Nachrichten lesen möchten, müssen wir das Flag `--from-beginning` benutzen.

Bild 1.3 In unserem Beispiel produzieren wir Daten mit dem `kafka-console-producer.sh` in das Topic flugdaten, und mit dem `kafka-console-consumer.sh` können wir diese Daten wieder lesen.

Interessanterweise, beziehungsweise anders als in vielen Messaging-Systemen, können wir Nachrichten nicht nur einmal lesen, sondern beliebig oft. Dies können wir nutzen, um zum Beispiel mehrere unabhängige Bodenstationen mit dem Topic zu verbinden, sodass alle die gleichen Daten lesen können. Oder es gibt unterschiedliche Systeme, die alle die gleichen Daten benötigen. Wir können uns vorstellen, dass wir nicht nur einen Anzeigebildschirm haben, sondern zusätzlich andere Services, wie zum Beispiel einen Service, der die Flugdaten mit aktuellen Wetterdaten abgleicht und entscheidet, ob etwas unternommen werden muss. Vielleicht möchten wir aber auch nach dem Flug die Daten analysieren und benötigen dafür die historischen Flugdaten. Dazu können wir den Consumer einfach mehrmals ausführen und bekommen jedes Mal das gleiche Ergebnis.

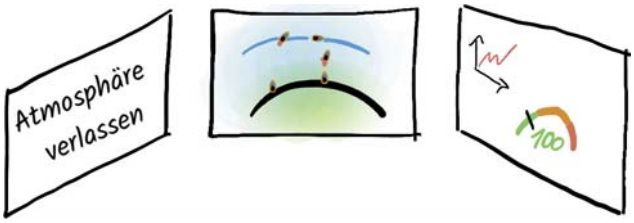

Bild 1.4 So könnte unser Kontrollzentrum für unser Beispiel aussehen.

Wir möchten jetzt aber gerne den aktuellen Zustand der Rakete in unserem Kontrollzentrum, welches in Bild 1.4 schemenhaft dargestellt ist, anzeigen, und zwar so, dass sich die Anzeige sofort aktualisiert, wenn es neue Daten gibt. Dazu starten wir den `kafka-console-consumer.sh` (ohne Timeout) in einem Terminal-Fenster. Sobald neue Daten verfügbar sind, holt sich der Consumer diese von Kafka und zeigt sie auf der Kommandozeile an:

```
# Nicht vergessen: STRG+C nutzen, um den Consumer zu stoppen
$ kafka-console-consumer.sh \
    --topic flugdaten \
    --bootstrap-server=localhost:9092
```

Um den Producer auf Raketen-Seite zu simulieren, starten wir jetzt den `kafka-console-producer.sh`. Der Befehl stoppt so lange nicht, bis wir **STRG+D** drücken und damit das *EOF-Signal* an den Producer senden:

```
# Nicht vergessen: STRG+D nutzen, um den Producer zu stoppen
$ kafka-console-producer.sh \
    --topic flugdaten \
    --bootstrap-server=localhost:9092
```

Der `kafka-console-producer.sh` schickt pro Zeile, die wir schreiben, eine Nachricht an Kafka. Das heißt, wir können nun Nachrichten in das Terminal mit dem Producer eintippen:

```
# Producer Fenster:
> Countdown beendet
> Liftoff
> Atmosphäre verlassen
> Vorbereitung Wiedereintritt
> Wiedereintritt erfolgreich
> Landung erfolgreich
```

Wir sollten diese auch zeitnah im Fenster mit dem Consumer sehen:

```
# Consumer Fenster:
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Stellen wir uns vor, dass ein Teil unserer Bodencrew im Homeoffice ist und den Flug von zu Hause aus verfolgen möchte. Dazu starten sie unabhängig voneinander ihren Consumer. Wir können das simulieren, indem wir in einem weiteren Terminalfenster einen `kafka-console-consumer.sh` starten, der alle Daten von Beginn an anzeigt:

```
# Fenster Consumer 2
$ kafka-console-consumer.sh \
    --topic flugdaten \
    --from-beginning \
    --bootstrap-server=localhost:9092
Countdown gestartet
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Wir sehen hier, dass Daten, die einmal geschrieben werden, parallel von mehreren Consumern gelesen werden können, ohne dass die Consumer miteinander reden müssen oder die Consumer sich vorher bei Kafka registrieren müssen. Kafka löscht erst einmal keine Daten, das heißt, dass wir auch später noch einen Consumer starten können, der historische Daten lesen kann.

Sagen wir nun, dass wir nicht nur eine Rakete gleichzeitig fliegen lassen möchten, sondern mehrere. Kafka hat damit kein Problem und kann ohne Weiteres Daten von vielen Producern gleichzeitig verarbeiten. Starten wir also einen weiteren Producer in einem weiteren Terminalfenster:

```
# Producer für Rakete 2
$ kafka-console-producer.sh \
    --topic flugdaten \
    --bootstrap-server=localhost:9092
> Countdown gestartet
```

Wir sehen alle Nachrichten von allen Producern in allen unseren Consumern in der Reihenfolge, in der die Nachrichten produziert wurden, auftauchen:

```
# Fenster Consumer 1
[…]
Landung erfolgreich
Countdown gestartet
# Fenster Consumer 2
[…]
Landung erfolgreich
Countdown gestartet
```

Nun ergibt sich aber das Problem, dass wir die Nachrichten von Rakete 1 und 2 nicht voneinander unterscheiden können. Wir könnten in jede Nachricht schreiben, von welcher Rakete die Nachricht geschickt wurde. Dazu aber später mehr.

Bevor wir weitermachen, sollten wir noch den Flug unserer zweiten Rakete abbrechen, da wir sie erst später starten lassen wollen:

```
# Producer für Rakete 2
[…]
> Countdown abgebrochen
```

Wir haben jetzt erfolgreich eine Rakete testweise gestartet und wieder gelandet. Dabei haben wir einige Daten in Kafka geschrieben. Dazu haben wir zuerst ein Topic *flugdaten* mit dem Kommandozeilenwerkzeug `kafka-topics.sh` erstellt, in welches wir alle Flugdaten für unsere Rakete schreiben. In dieses Topic haben wir mithilfe von `kafka-console-producer.sh` einige Daten produziert. In unserem Fall waren dies Informationen über den aktuellen Status der Rakete. Diese Daten konnten wir mithilfe des `kafka-console-consumer.sh` lesen und anzeigen. Wir sind sogar weiter gegangen und haben parallel mit mehreren Producern Daten produziert und mit mehreren Consumern Daten gleichzeitig gelesen. Mit dem Flag `--from-beginning` im `kafka-console-consumer.sh` waren wir in der Lage, auf historische Daten zuzugreifen. Wir haben so schon drei Kommandozeilenwerkzeuge kennengelernt, die mit Kafka ausgeliefert werden.

Nachdem wir diese Erfahrungen gesammelt haben, können wir nun alle geöffneten Terminals schließen. Producer beenden wir mit **STRG+D** und Consumer mit **STRG+C**. Dieses Beispiel soll nicht darüber hinwegtäuschen, dass Kafka überall da eingesetzt wird, wo größere Datenmengen verarbeitet werden. Aus unserer Schulungserfahrung wissen wir, dass Kafka bei vielen Automobilherstellern, Supermarktketten, Logistikdienstleistern und selbst in vielen Banken und Versicherungen intensiv eingesetzt wird.

Nachdem wir in diesem Kapitel einen ersten Überblick über Kafka erhalten und ein einfaches Testflug-Szenario durchexerziert haben, werden wir im nächsten Kapitel tiefer in die

Materie einsteigen und die Kafka-Architektur näher untersuchen. Wir werden uns anschauen, wie Kafka-Nachrichten strukturiert sind und wie genau sie in Topics organisiert werden. In diesem Zusammenhang werden wir uns auch mit der Skalierbarkeit und der Zuverlässigkeit von Kafka beschäftigen. Außerdem werden wir mehr über Producer, Consumer und das Kafka-Cluster selbst erfahren.

2.2 Kafka als verteiltes Log

Nachdem wir im ersten Kapitel unsere Rakete haben fliegen lassen und den Flug in Kafka nachvollzogen haben, fokussieren wir uns im folgenden Kapitel auf das Verständnis der Grundlagen. In unserem Raketenmodell ist dies die erste Stufe. Was ist Kafka für ein System? Warum funktioniert es so, wie es ist? Welche Auswirkungen hat ein System wie Kafka auf die IT-Architektur in meinem Unternehmen? Diese und weitere Fragen möchten wir in diesem Kapitel behandeln.

Wir kommen mit Kafka auch ohne die Antworten auf diese Fragen recht weit, doch spätestens abseits von Sandkastenprojekten ist es essenziell, die genaue Arbeitsweise zu verstehen. Für die meisten ITler sind die Kafka-Konzepte relativ unbekannt. Aber das Verstehen dieser Konzepte führt oft zu Aha-Effekten und Erkenntnissen, wie wir unsere IT-Architektur, die Entwicklung und den Betrieb noch weiter optimieren können.

2.2.1 Logs

Wir beschreiben Kafka gerne als ein Log. Obwohl der Begriff des Logs für viele im Zusammenhang mit Kafka neu ist, begegnen uns Logs ständig im (IT-)Alltag.

2.2.1.1 Was genau ist eigentlich ein Log?

Unsere Betriebssysteme produzieren System-Logs. Wenn wir für diese Systeme zuständig sind, nutzen wir die Logs regelmäßig, um den Status der Systeme zu überprüfen und in Fehlerfällen zu verstehen, wie es zu diesen Fehlern kommen konnte. Vielleicht setzen wir sogar Log-Überwachungssysteme ein, die bei bestimmten Ereignissen Alarm schlagen und uns notfalls (aber hoffentlich nicht) mitten in der Nacht wecken, damit wir auf die Fehler reagieren können.

Dasselbe gilt auch für unsere Applikationen und Services. Sie produzieren Log-Ereignisse, in denen wir sehen können, was gerade passiert, ob es zu Fehlern kommt oder ob alle Systeme einwandfrei laufen. Wir nutzen beides, System-Logs und Applikations-Logs, um Fehler zu analysieren und den Zustand des entsprechenden Systems zu überwachen. Hier sind Logs nützlich und essenziell, aber nicht der Kern unserer Systeme.

Anders sieht es bei Datenbanksystemen aus: Hier ist der Commit-Log zwar tief im System versteckt, aber eine der Hauptkomponenten des Systems. Im Commit-Log speichern Datenbanken jegliche Veränderungen an Daten. Werden Daten hinzugefügt, geändert oder gelöscht, schreibt die Datenbank diese Veränderung zuerst in den Commit Log und erst dann in die entsprechenden Tabellen. Sollte die Datenbank ausfallen, garantiert uns der Commit-Log, dass wir den letzten sauberen Zustand wiederherstellen können, ohne Daten zu verlieren. Ähnliches gilt auch für die Replikation, also den Cluster-Betrieb, von Datenbanken. Oft basiert diese Replikation darauf, das Commit-Log zwischen den Cluster-Einheiten auf dem aktuellen Stand zu halten, und jeder Server baut dann die Tabellen unabhängig voneinander basierend auf den Daten im replizierten Commit-Log auf.

So unterschiedlich diese Logs auch sein mögen, geben alle die Antwort auf die Frage *„Was ist passiert?“*. Die System- und Applikations-Logs beantworten diese Fragen dem Betreiber der Systeme, und die Commit Logs in Datenbanken beantworten diese Frage den anderen Servern im Datenbankcluster oder auch sich selbst, wenn das System abstürzt oder sonstige Fehler auftreten.

Wenn wir zurück an unser Beispiel aus dem ersten Kapitel denken, können wir auch logähnliche Datenstrukturen erkennen. Hier folgt zum Beispiel der Inhalt unseres Topics `flugdaten`:

```
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Der Grund für die Ähnlichkeiten ist einfach. Kafka basiert auf Logs als zentrale Datenstruktur. Im Gegensatz zu Datenbanken versteckt Kafka das Log nicht, sondern das Log wird zum Kernelement unseres Systems. Unsere Daten werden in Kafka in Logs gespeichert, und wir können die gleichen Muster und Erkenntnisse aus der Welt der System-, Applikations- und Datenbank-Logs nehmen und auf Kafka anwenden!

2.2.1.2 Grundlegende Eigenschaften eines Logs

Aber auch abseits der IT nutzen wir Logs sehr häufig. Das wohl einfachste Beispiel sind Tagebücher. Wenn wir im Tagebuch vorne anfangen und zum Beispiel jeden Tag eine Seite schreiben, ohne dazwischen Platz zu lassen, und dabei einen Kugelschreiber statt eines Bleistifts benutzen, erkennen wir die Eigenschaften eines Logs wieder:

Reihenfolge und Sortierung: Nachrichten in einem Log sind nach Zeit sortiert. Am Anfang des Logs steht die älteste Nachricht und am Ende die neueste Nachricht. Genauso ist es auch in unserem Tagebuch. Wir wissen (wenn wir keine Seiten frei lassen), dass der Eintrag auf Seite 10 älter ist als der Eintrag auf Seite 11.

Schreib- und Leserichtung: Neue Einträge schreiben wir immer ans Ende des Logs. Wenn wir das Log lesen, fangen wir meist auf einer Seite an und lesen erst die älteren Einträge und dann die neueren. Natürlich können wir im Tagebuch auch zurückblättern, aber dennoch ist die natürliche Leserichtung von alt nach neu.

Unveränderbarkeit: Sobald wir einen Eintrag geschrieben haben, können wir diesen nicht mehr ohne Weiteres ändern oder entfernen. Im Tagebuch ist dies noch möglich, indem wir auf den Seiten Anmerkungen hinzufügen, Wörter durchstreichen oder Seiten ausreißen. Aber zumindest fallen diese Änderungen meistens auf.

Eine weitere interessante Eigenschaft von Logs ist, dass wir sehr gut nachvollziehen können, wie der vom Log beschriebene Teil der Welt sich über die Zeit verändert hat, und damit können wir auch herausfinden, wie dieser Teil der Welt zu einem bestimmten Zeitpunkt ausgesehen hat.

Oft schauen wir uns System- oder Applikations-Logs mit der Frage *„Wie konnte es zu diesem Fehler kommen?“* an. Wenn wir die Einträge im Log chronologisch lesen, können wir oft nachvollziehen, was sich in der Zeit verändert hat, und oft können wir dadurch einen Hinweis finden, was schiefgelaufen ist. Dasselbe gilt auch für Datenbanken. Um herauszu-

finden, wie eine Tabelle zu einem bestimmten Zeitpunkt ausgesehen hat, fangen wir im Commit-Log ganz am Anfang an, arbeiten uns Eintrag für Eintrag vor, sehen so, wie sich die Tabelle über die Zeit verändert hat und warum wir genau die Einträge in der Tabelle sehen, die sie uns anzeigt.

Aber nicht nur das, Logs erlauben es uns, in der Zeit rückwärts zu reisen! Wenn wir den Zustand unserer Datenbank auf den Stand von vorgestern wiederherstellen möchten, können wir das Log von Anfang an bis zu den Einträgen von vorgestern durchgehen und sehen, wie die Datenbankwelt damals aussah.

Mit diesem Wissen können wir das Log etwas formaler definieren. Ein Log ist eine Liste von Ereignissen, auf der wir folgende zwei Operationen definieren:

Schreiboperation: Wir fügen neue Einträge an das Ende der Liste an.

Leseoperation: Wir beginnen bei einem bestimmten Eintrag und lesen von alt nach neu, üblicherweise bis zum Ende des Logs.

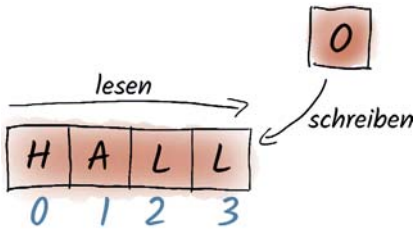

Bild 2.2 Ein Log ist eine Liste, an die wir ausschließlich Elemente hinten anfügen und sequenziell von einer Position (Offset) auslesen können. Wir können zum Beispiel zuerst alle Elemente ab Offset 0 lesen und uns merken, dass wir als Nächstes die Nachricht ab Offset 4 lesen möchten. Solange es keine neuen Einträge gibt, bekommen wir kein Ergebnis, aber wenn wir ein Element anfügen und es erneut versuchen, erhalten wir die neuen Einträge und merken uns wieder den Offset des Eintrages, den wir als Nächstes lesen möchten (hier in dem Beispiel dann die 4).

Meist sind unsere Logs zu groß und zu lang, um sie am Stück lesen zu können. Applikations-Logs werden gerne mehrere Gigabyte groß. Wie merken wir uns also, welche Einträge wir bereits gelesen haben und welche noch nicht? Wir nummerieren unsere Einträge durch. Der erste Eintrag bekommt die Position 0, der zweite die Position 1 und so weiter. In Kafka ist diese Position der *Offset*, den wir zuvor schon kennengelernt haben. Der Offset gibt einerseits die Position einer Nachricht im Log an (ähnlich einer Seitenzahl), aber er gibt auch an, welchen Eintrag wir als Nächstes lesen möchten (ähnlich einer Seitenzahl, die wir uns merken). Wie in einem Buch fangen wir meistens ganz vorne an zu lesen (Offset 0), dann lesen wir ein paar Seiten und merken uns die Seite (beziehungsweise den Offset), die wir als Nächstes aufschlagen möchten. Genau wie wir bei Büchern verwendet Kafka Lesezeichen. Die Systeme, die die Logs lesen, können zwar die Offsets im RAM vorhalten, aber dies ist nicht sehr zuverlässig. Das System könnte jederzeit ausfallen und müsste dann von Anfang an lesen. Stattdessen hilft uns Kafka dabei, die Offsets zu merken.

2.2.1.3 Die Rolle des Logs in Kafka

Da Logs in IT-Systemen meistens nicht eine statische Anzahl an Seiten haben, sondern ständig neue Einträge hinzukommen, haben sie üblicherweise auch kein Ende. In Kafka

merken wir uns den Offset, den wir als Nächstes lesen möchten, und fragen kontinuierlich den Kafka-Cluster an, ob es neue Einträge gibt. Gibt es neue Einträge, bekommen wir diese Einträge zurück und merken uns den nächsten Offset, den wir lesen möchten. Gibt es keine neuen Einträge, dann bekommen wir auch keine Daten und müssen unseren Offset auch nicht anpassen.

Das Interessante an einem Offset ist, dass er lediglich die Position einer Nachricht beschreibt, aber nicht deren Inhalt. Das ist genauso wie mit den Seitenzahlen im Buch. Es ist eine fortlaufende Nummerierung ohne weitere Bedeutung. Das ist wichtig zu verstehen, denn in Logs ist der Offset die einzige Möglichkeit, Daten zu adressieren. Wir können in einem Log nicht auf konkrete Elemente zugreifen.

Dies hat enorme Auswirkungen auf die Architektur unserer Services, die Kafka benutzen. Wir sollten Kafka nicht nutzen, um Anfragen über die Daten darüber ad hoc zu beantworten. Wir sollten Kafka auch nicht nutzen, um ähnlich wie in einem Key-Value Store auf Daten mit einem bestimmten Key zuzugreifen. Jede dieser Operationen würde unter Umständen eine Suche über alle Daten im Log provozieren.

Logs, beziehungsweise Kafka, sind nicht der Heilbringer, um alle unsere Datenbanken, Caches und Analytics-Werkzeuge durch ein System zu ersetzen. Aber Kafka kann uns dabei helfen, die Daten in unserem Unternehmen effektiver zu organisieren und zwischen den Systemen auszutauschen.

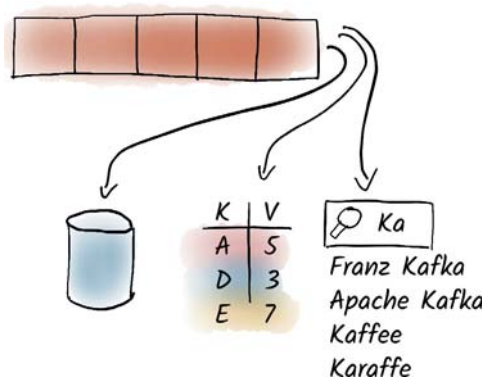

Bild 2.3 Ein Log ist eine perfekte Datenstruktur, um Daten zwischen Systemen auszutauschen. Üblicherweise arbeiten wir nicht direkt mit den Daten im Log, sondern speichern diese in einem Datenformat, welches am besten für unseren jeweiligen Anwendungsfall geeignet ist. Zum Beispiel können wir relationale Datenbanken nutzen, um über unsere Daten komplexe Abfragen auszuführen. Wenn wir schnell auf vorbereitete Daten zugreifen möchten, können wir zum Beispiel einen In-Memory Key-Value Store wie Redis benutzen. Sollten wir eine Suchfunktion über den Daten im Log anbieten wollen, können wir dafür eine Suchmaschine benutzen.

Statt zu versuchen, Kafka als zentrale Datenbank für unsere Services zu missbrauchen, nutzen wir Kafka als zentrale Datendrehscheibe und nutzen in unseren Services die beste Technologie für unseren Anwendungsfall. Wenn wir zum Beispiel einen Suchservice über unsere Astronauten-Stammdaten erstellen möchten, schreiben wir die Daten aus Kafka in eine Suchmaschine wie zum Beispiel *ElasticSearch*. Wenn wir Daten ad hoc auswerten möchten, könnten wir dafür eine relationale Datenbank wie PostgreSQL benutzen und Daten aus Kafka in die Datenbank schreiben.