Willkommen zum Seminar

anderScore
trust in competence

Apache Kafka - Event Streaming mit Java

Jan Lühr (M.Sc. Computer Science)

- Senior Software Engineer
- Schwerpunkte
  - Pragmatic Architect
  - Integration und Migration
  - Web / Mobile Engineering
  - Clean Code
  - Trainings, Artikel, Vorträge
  - Network- and Security-Techniques
  - IT-Trainer
- Java, Spring, JEE, Kafka, Android, Microservices,…

Vorstellung

Victoria Wadewitz (B.Sc. Mathematik)

- Software Engineer
- Schwerpunkte
  - Web Engineering
  - Datenbankmodellierung und –design
- Java, Spring, Kafka, elasticsearch,…

Vorstellung anderScore GmbH

Individuelle Anwendungsentwicklung - Java Enterprise, Web, Mobile

- seit 2005 ♦ in Köln ♦ für alle Branchen ♦
- nach Aufwand & zum Festpreis
- ✓ Digitalisierung / Prozesse / Integration
- ✓ Migration
- ✓ Neuentwicklung
- ✓ Notfall / kritische Situation
- ➔ pragmatisch, zielgerichtet, zuverlässig

Goldschmiede anderScore

Kompletter SW Life Cycle

- Projektmanagement / agile Methodik
- Anforderungsanalyse
- Architektur & SW-Design
- Implementierung & Testautomation
- Studien & Seminare

Einführung neues Versicherungsprodukt

Onlinebanking

Naturschutz: Kartierung bedrohter Arten

µservices Automotive

Stabilisierung Retail

Migration Energieversorger

Schnittstellen Gesundheitswesen

Adressverwaltung Logistik

Nutzerservices Energieversorger

Börsenhandel Wertpapiere

Security Härtung

EAM & Refactoring Leasing

... und für Sie? Sprechen Sie uns an!

Apache Kafka – Event Streaming mit Java

04.10. – 05.10.22

Jan Lühr

Agenda

Begrüßung

1. Einführung
2. Paradigmen und Funktionsweise
3. Komponenten
4. Implementierung von Clients
5. Analyse und Transformation
6. Broker Operations
7. Best Practices
8. Ausblick

Lektion 1 - Einführung

Inhalt und Ziel

- Workshop
  - Einführung in grundlegende Kafka-Konzepte
  - Überwindung von Einstiegshürden
  - Funktionen und Features mit Aufgaben erarbeiten

- Zielgruppe
  - Softwareentwickler:innen, Architekt:innen und DevOps mit guten Java-Kenntnissen

- Voraussetzungen
  - Gute Java Kenntnisse
  - Sicherheit im Umgang mit einer IDE (z.B. IntelliJ)

Zeitplan

Beginn	09:00 Uhr
Kaffeepause	ca. 10:30 Uhr
Mittagspause	12:00 bis 13:00 Uhr
Ende	16:00 Uhr

Remote Training

- Video-Konferenz über Zoom
  - Bildschirmfreigabe für Folien
  - Lautsprecher + Mikrofon benötigt, Kamera empfehlenswert
  - Zugangsdaten, um Rechner per Web-Browser zu erreichen:

    https://connect.gfu.cloud/

    Benutzername: 55842

- Entwicklung: Remote Desktop Protocol (RDP) zur GFU
  - Praktische Übungen
  - Ubuntu VM
  - Aufschaltung über Zoom möglich

- Material auf GitHub

  https://github.com/anderscore-gmbh/kafka-22.10

Vereinbarungen

- Pausen
  - Gemeinsam zu vorgegebenen Zeiten
  - Individuell während der Übungen
- Erreichbarkeit Dozent
  - Zoom (Chat, Mikrofon)
  - E-Mail
  - Kamera aus: gerade nicht anwesend bzw. ansprechbar
- Regeln
  - Mikrofon möglichst aus (Hintergrundgeräusche)
  - Bei Fragen: "Hand heben" oder Chat
  - Wenn Übung fertig, selbst in Hauptsession zurückkehren

Material

- Kafka Quickstart
  - https://kafka.apache.org/quickstart
- Kafka Cheat Sheet
  - https://github.com/lensesio/kafka-cheat-sheet
- Kafka E-Book
  - https://www.confluent.io/resources/kafka-the-definitive-guide

Jetzt sind Sie dran!

- Name
- Vorwissen
- Erwartungen
- Themenwünsche

Ihre Umgebung

- Kafkacat
- JDK 11
- Maven
- Entwicklungsumgebung (z.B. IntelliJ)
- Docker (optional)
- Docker Compose (optional)

Aufgabe 0 – Hello World mit kafkacat/kcat

Aufgabenstellung:

- Installieren Sie `kafkacat` bzw. `kcat` auf Ihrem System
- Senden Sie eine Nachricht an `broker-1.k.anderscore.com:9092` – Topic: `HelloWorld`
- Konsumieren Sie alle Nachrichten des Topics und geben Sie diese aus

Hinweise:

- Ubuntu / Debian (vorinstalliert): `apt-get install kafkacat`
- Nachricht senden:

```
echo "Hallo Welt" | kafkacat -b broker-1.k.anderscore.com -t HelloWorld
```

- Nachricht konsumieren:

```
kafkacat -b broker-1.k.anderscore.com -t HelloWorld
```

Aufgabe 0 – Hello World mit kafka-console-producer/consumer

anderScore trust in competence

Kafka Archiv herunterladen und entpacken:

https://dlcdn.apache.org/kafka/3.2.1/kafka_2.13-3.2.1.tgz

Nachricht in das Topic schreiben:

```
$ bin/kafka-console-producer.sh --topic HelloWorld --bootstrap-server broker-1.k.anderscore.com:9092
This is my first event
This is my second event
```

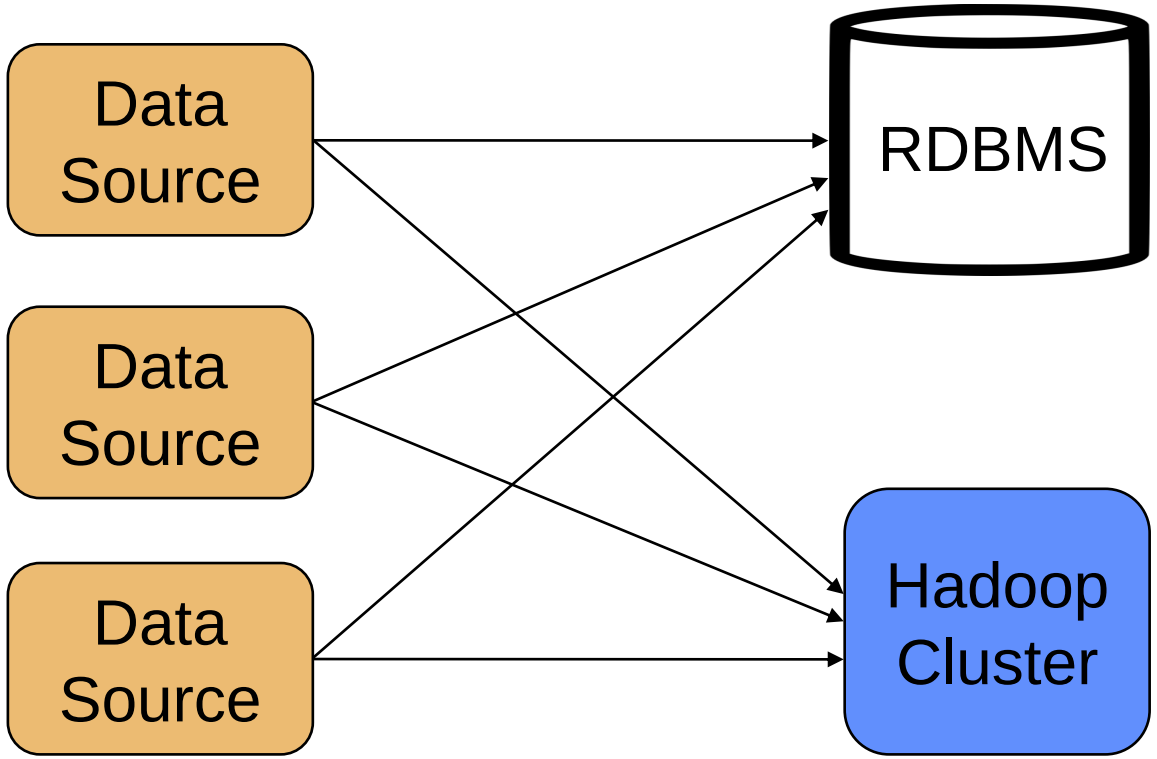
Nachricht aus Topic auslesen:

```
$ bin/kafka-console-consumer.sh --topic HelloWorld --from-beginning --bootstrap-server broker-1.k.anderscore.com:9092
```

Lektion 2 - Paradigmen und Funktionsweise von Kafka

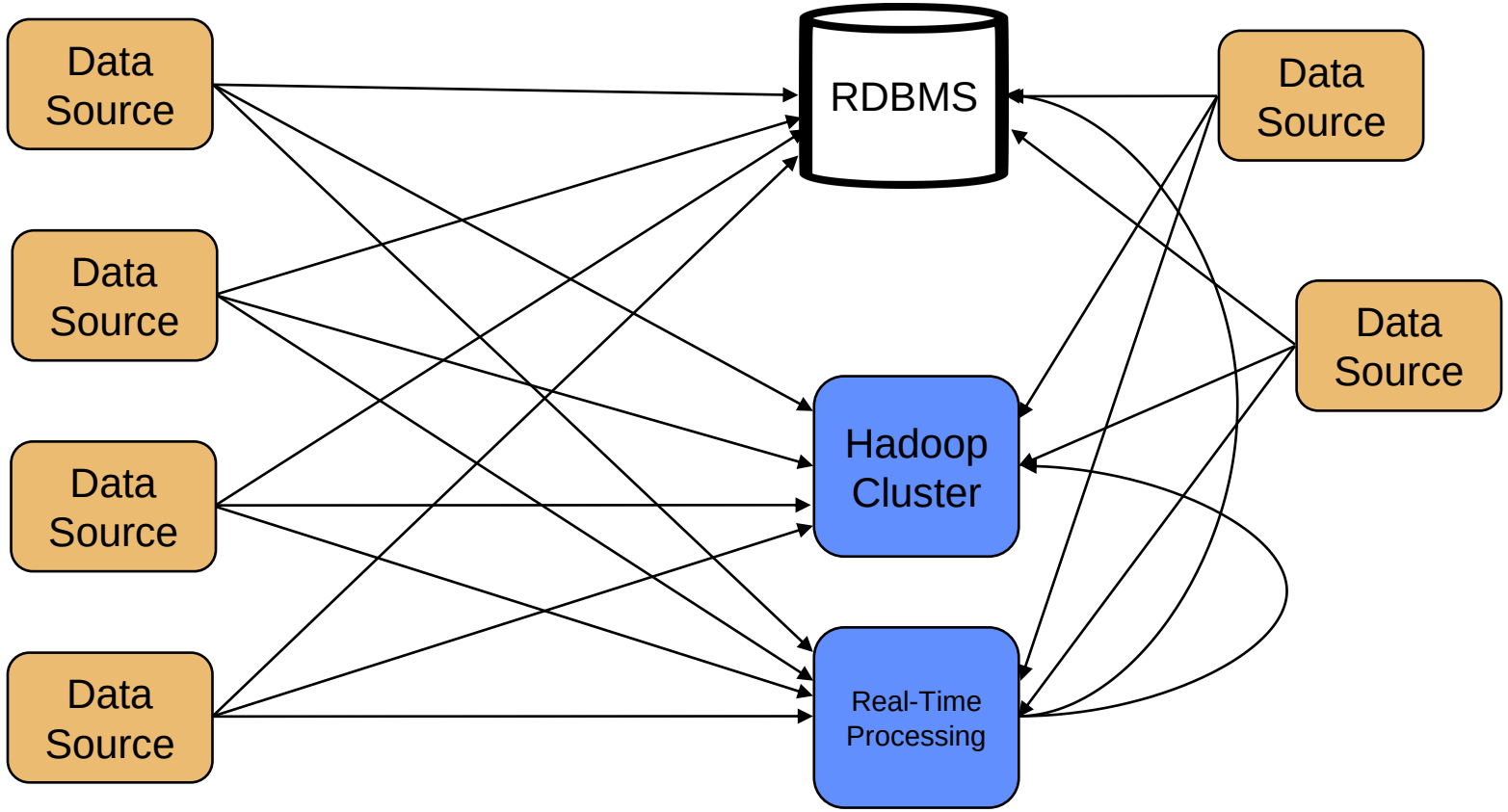
Was ist Kafka?

Einfache Datenverteilung:



Was ist Kafka?

Komplexe Datenverteilung:



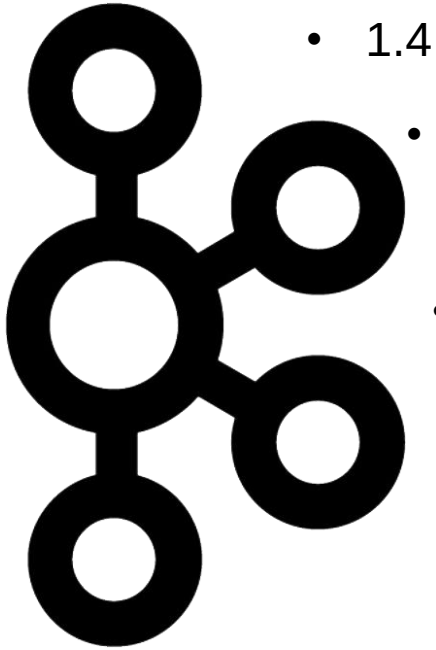
Motivation für Kafka

- Motivation
  - Komplexe Datenverteilung bewältigen
  - Batch-Verarbeitungsprozess verbessern
  - Zeitnahe Verarbeitung ermöglichen

- Ziele von Kafka
  - Pipelines vereinfachen
  - Data Stream Handling

- Ansatz
  - Stream Data Platform: Stream Processing statt Batch Processing
  - Middleware für persistente Logs / Streams
  - Ähnlich zu MQTT und Message Queuing

Apache Kafka



- Enstanden auf LinkedIn (2010)
  - Teil der Core-Architektur
  - 1.4 Milliarden Nachrichten pro Tag
  - Genutzt von:
    - IBM, zalando, airbnb, Cisco, Netflix, Paypal, Twitter…
  - Use Cases:
    - Event Verarbeitung (*quasi realtime*)
    - Log Aggregation
    - Metriken & Analyse
    - Messaging / µService Kommunikation
- Keine Realtime- bzw. Echtzeitverarbeitung (Werbeversprechen; im Sinne von Reordering Queues)

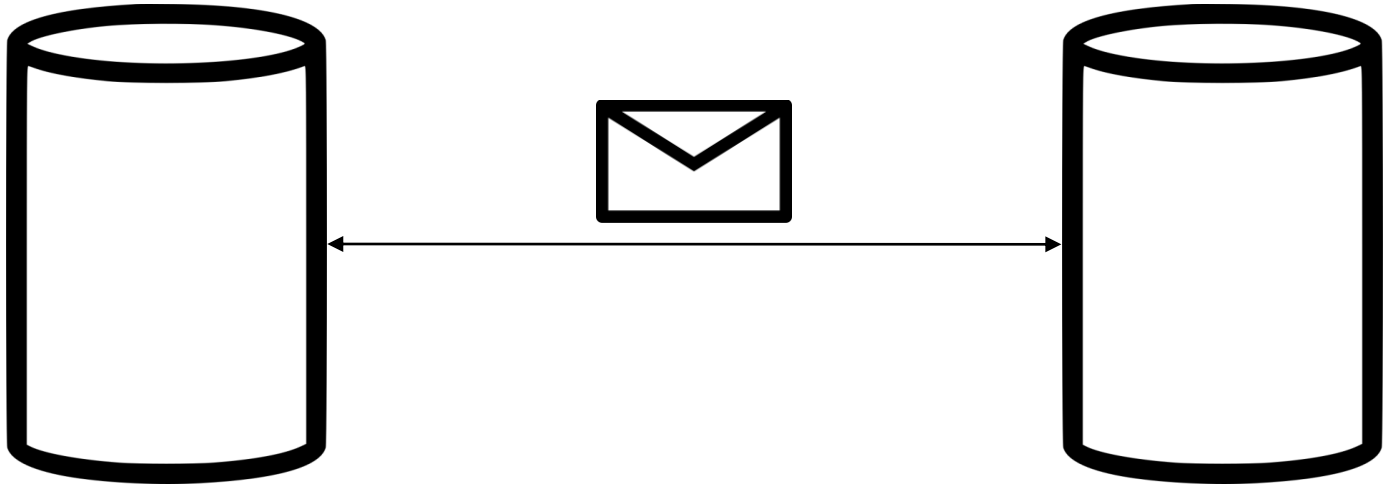
Kafka-Versionen

- Aktuell: Version 3.2.1
- Release Notes der Versionen unter

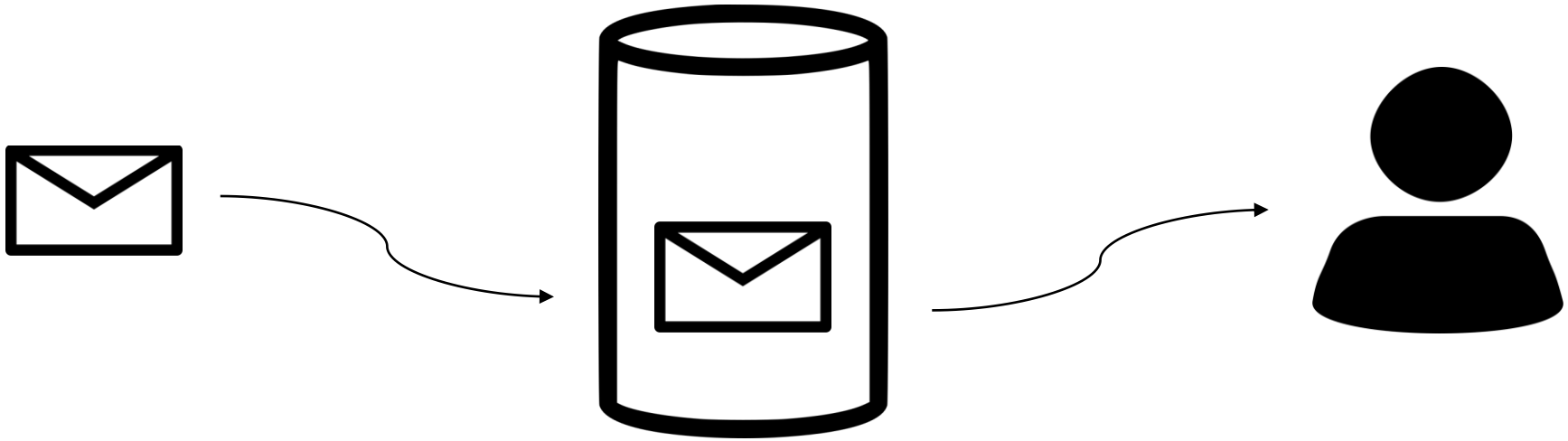
  https://archive.apache.org/dist/kafka/<version>/RELEASE_NOTES.html
- Upgrade auf neue Version und Changelog unter

  https://kafka.apache.org/28/documentation.html#upgrade

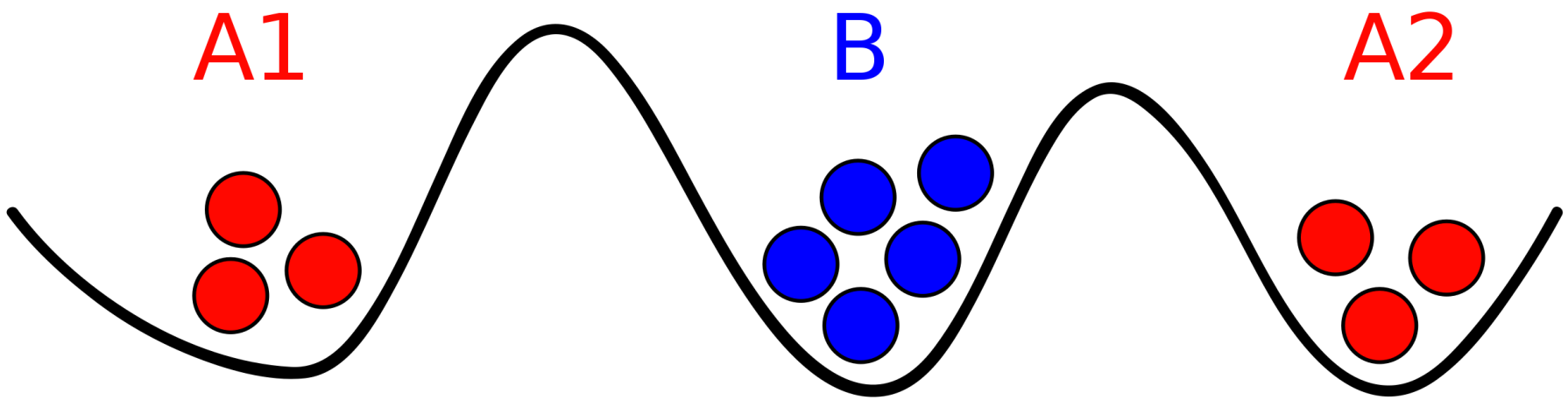
Synchrone Kommunikation



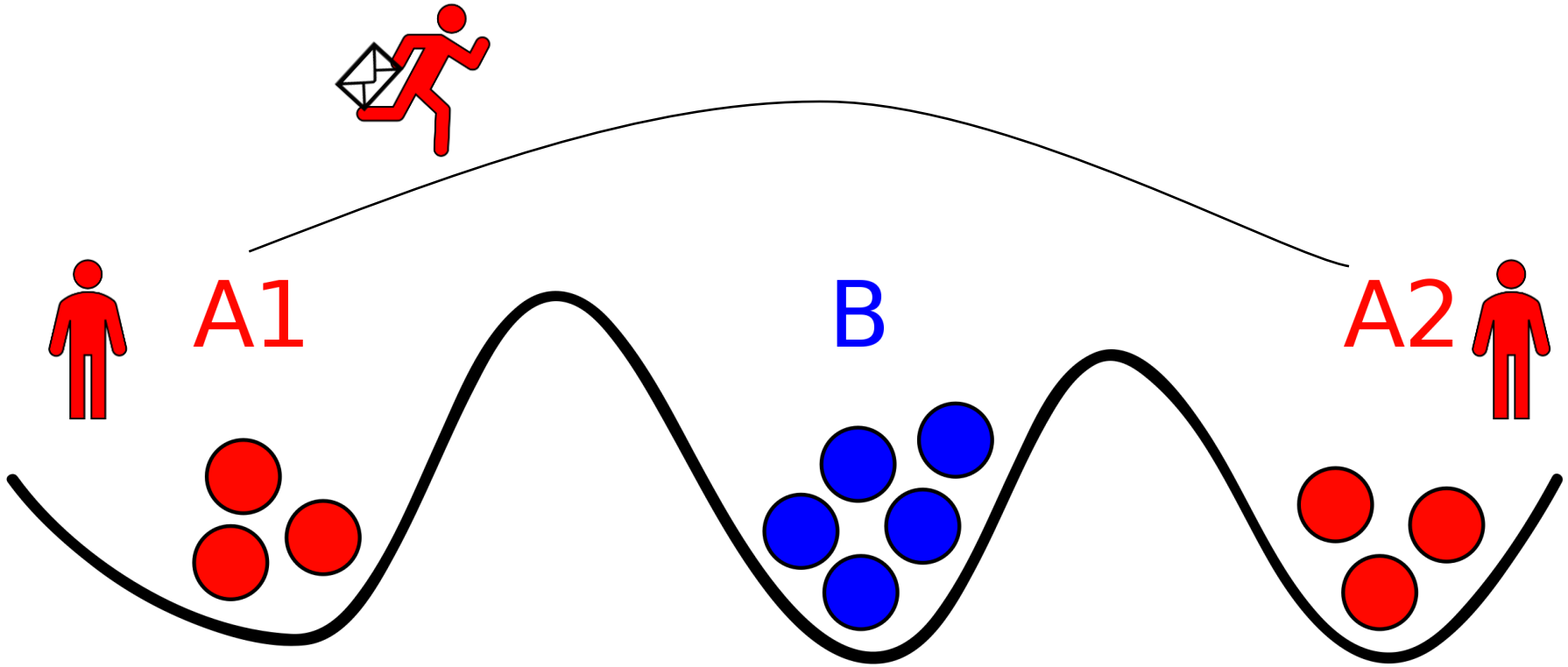
Asynchrone Kommunikation



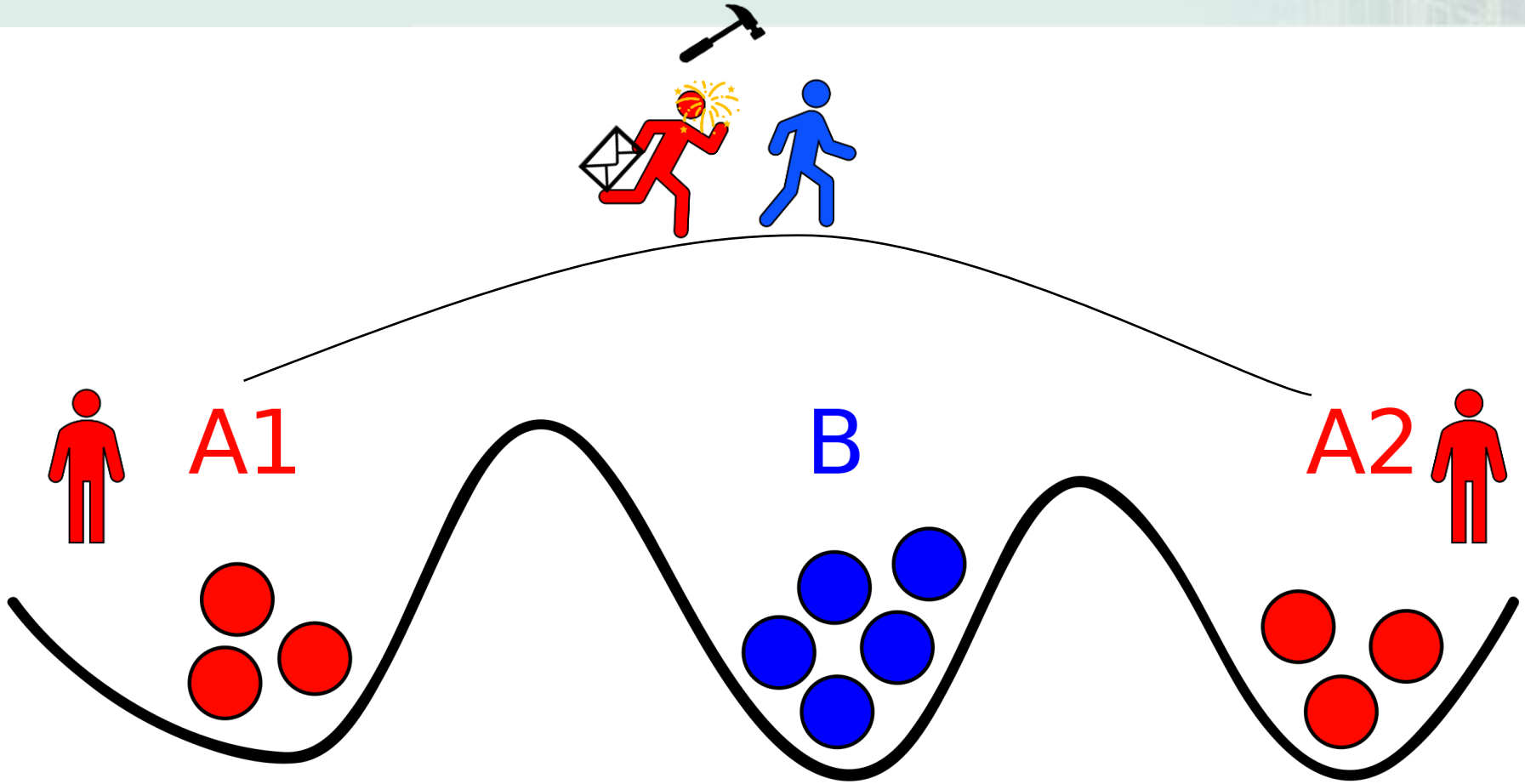
Two Generals Problem



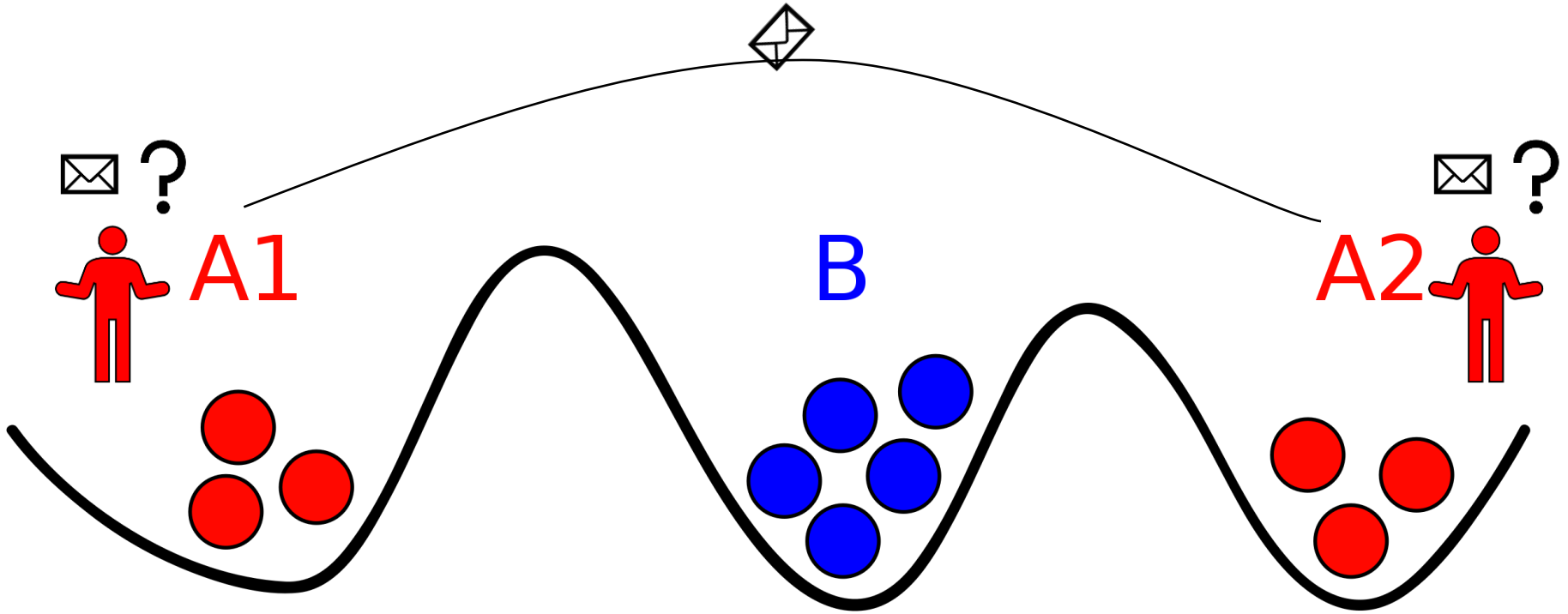
Two Generals Problem



Two Generals Problem



Two Generals Problem



Idempotenz

Definition:

Eine Operation, welche mehrfach hintereinander ausgeführt
das gleiche Ergebnis wie bei einer einzigen Ausführung liefert.

https://de.wikipedia.org/wiki/Idempotenz

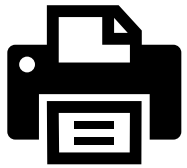
Semantik beim Aufruf

Was passiert bei einem Fehler?

- **At least once**
  - Mindestens ein Mal: Risiko von Duplikaten
- **At most once**
  - Maximal ein Mal: Kein Neuversuch beim Fehlschlag
- **Exactly once**
  - Genau ein Mal: In Praxis schwer zu erreichen

At Least Once Beispiel: Print Server

At Least Once

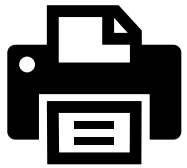


Situation: Druckfehler / Toner leer: Ausdruck zu hell!

1. Fehler wird behoben
2. Alle Seiten in der Warteschlange werden gedruckt
3. … sehr viel Papier

At Most Once Beispiel: Print Server

At Most Once

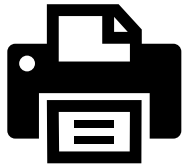


Situation: Druckfehler / Toner leer: Ausdruck zu hell!

1. Fehler wird behoben
2. Keine Seite in der Warteschlage wird gedruckt
3. … hin und her laufen.

Exactly Once Beispiel: Print Server

Exactly Once



Situation: Druckfehler / Toner leer: Ausdruck zu hell

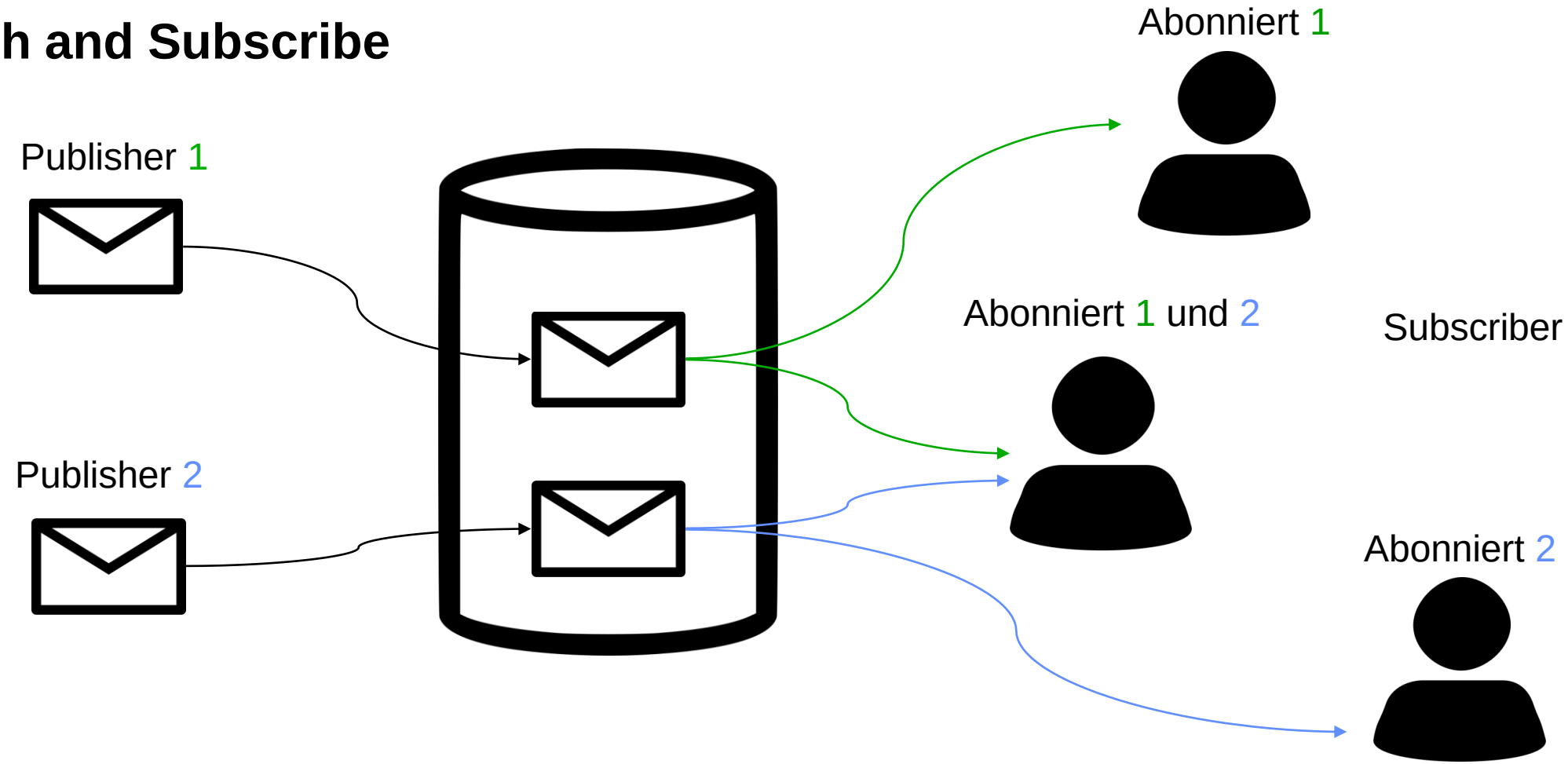
1. Fehler wird behoben
2. Genau die zu hellen Seiten werden gedruckt
3. … Perfekt ☺

Woher weiß der Drucker, welche Seiten zu hell sind?

- Idee: Seitennummer am Bedienfeld eingeben
- Im Allgemeinen: schwer umsetzbar
- Idee *auch bei Kafka*: Stand der Verarbeitung im Ergebnis speichern

https://www.confluent.io/de-de/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

Publish and Subscribe



Active Polling

Konzept:

- Es wird nicht automatisch an alle Subscriber gesendet (vgl. Observer Pattern)
- Subscriber senden zyklisch Anfragen
- Existiert neuer Inhalt, wird er als Antwort zurückgeliefert

Event Sourcing

Konzept:

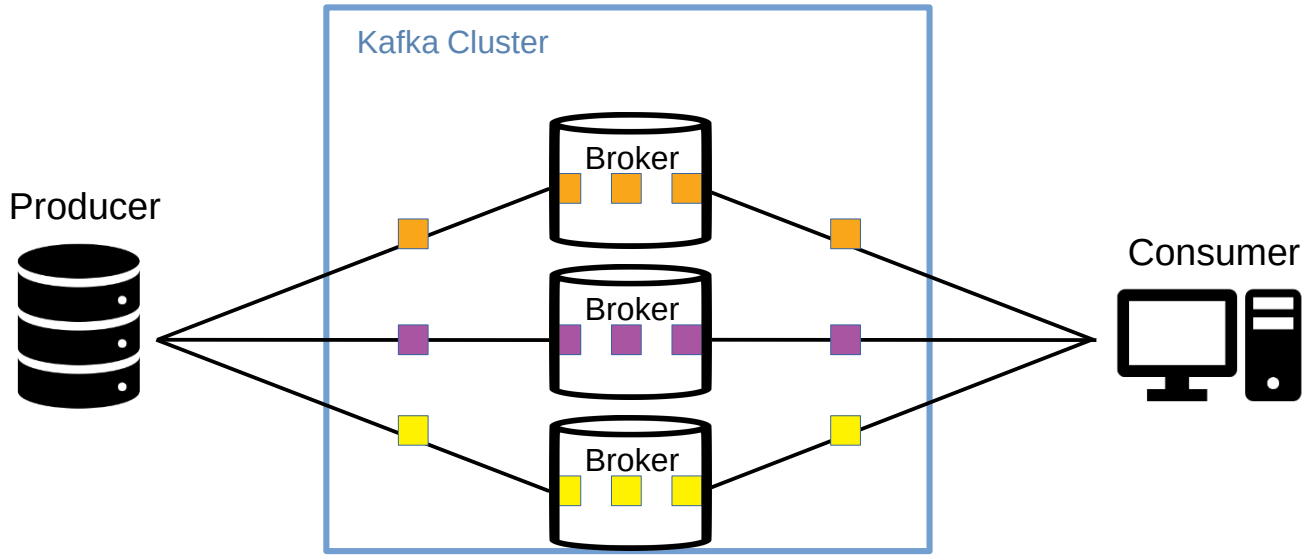
- Veränderung eines Zustandes = Event
- Nach Empfang neuer Daten werden alte nicht gelöscht
- Neue Events werden kontinuierlich im Event Store an alte angehängt
- Kafka: Retention Policy

Zweck:

- Kein Informationsverlust
- Analysemuster
- Macht Kafka zu einem Hybrid aus Datenbank und Messaging System

High-Level-View of Kafka

- *Producer* senden Nachrichten an den Kafka-Cluster
- *Consumer* lesen Nachrichten vom Kafka-Cluster
- *Broker* sind die Speicher- und Nachrichtenkomponenten des Kafka-Clusters
- Die kleinste Einheit an Daten in Kafka sind *Messages,* sie werden gruppiert in sogenannten *Topics*



Aufgabe 1 – Hello Kafka World

Kafka Quickstart oder **Docker Broker**

Aufgabenstellung Quickstart:

- Laden Sie das aktuelle Kafka Release aus:

  http://kafka.apache.org/quickstart
- Starten Sie das Kafka Environment in der Linux Shell
- Legen Sie einen Topic an
- Schreiben Sie einige Nachrichten in das Topic
- Lesen Sie die Nachrichten aus

Aufgabe 1.1 - Hinweise Quickstart

anderScore trust in competence

Hinweise

- Download Link Kafka:

https://dlcdn.apache.org/kafka/3.2.1/kafka_2.13-3.2.1.tgz

```
$ tar -xzf kafka_2.13-3.2.1.tgz
$ cd kafka_2.13-3.2.1
```

- Zookeeper starten:

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

- Kafka Broker starten:

```
$ bin/kafka-server-start.sh config/server.properties
```

Aufgabe 1.1 - Hinweise Quickstart

anderScore trust in competence

Topic erzeugen:

```
$ bin/kafka-topics.sh --create --topic HelloWorld --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092
```

Topic prüfen:

```
$ bin/kafka-topics.sh --describe --topic HelloWorld --bootstrap-server localhost:9092
Topic: HelloWorld PartitionCount:1 ReplicationFactor:1 Configs:
Topic: HelloWorld Partition: 0 Leader: 0 Replicas: 0 Isr: 0
```

Aufgabe 1.1 - Hinweise Quickstart

Nachricht in das Topic schreiben:

```
$ bin/kafka-console-producer.sh --topic HelloWorld --bootstrap-server localhost:9092
This is my first event
This is my second event
```

Nachricht aus Topic auslesen:

```
$ bin/kafka-console-consumer.sh --topic HelloWorld --from-beginning --bootstrap-server localhost:9092
```

Aufgabe 1.2 - Docker

Aufgabenstellung Docker:

- Setzen Sie einen eigenen Kafka Broker mit Docker gemäß folgendem Tutorial auf: https://medium.com/big-data-engineering/hello-kafka-world-the-complete-guide-to-kafka-with-docker-and-python-f788e2588cfc
- Starten Sie die Kafka Shell
- Legen Sie einen Topic an
- Initialisieren Sie einen Producer, der eine „Hello World“ Nachricht in den Topic schreibt
- Initialisieren Sie einen Consumer von einem anderen Kafka Terminal, welcher Nachrichten aus dem Topic liest

Aufgabe 1.2 - Hinweise Docker

Hinweise:

- Klonen Sie das *kafka-docker* Projekt und initialisieren Sie die Umgebung mit docker-compose:

  https://github.com/wurstmeister/kafka-docker

```
> git clone https://github.com/wurstmeister/kafka-docker.git
> cd kafka-docker

# Updaten Sie KAFKA_ADVERTISED_HOST_NAME in 'docker-compose.yml',
# Ändern Sie den Wert zu: 172.17.0.1
> vi docker-compose.yml
> docker-compose up -d

# Optional: Scalen Sie das Cluster, indem Sie mehr Broker hinzufügen
> docker-compose scale kafka=3

# Sie können die laufenden Prozesse mittels folgendem Befehl checken:
> docker-compose ps

# Zerstören Sie das Cluster, wenn Sie damit fertig sind:
> docker-compose stop
```

Aufgabe 1.2 - Hinweise Docker

Kafka Shell

- Mit folgendem Befehl hochfahren

```
> ./start-kafka-shell.sh <DOCKER_HOST_IP/KAFKA_ADVERTISED_HOST_NAME>
# Wie im Beispiel:
> ./start-kafka-shell.sh 172.17.0.1
```

Einen 'Hello' Topic anlegen

- In der Kafka Shell:

```
> $KAFKA_HOME/bin/kafka-topics.sh --create --topic test --partitions 4 --replication-factor 1 --bootstrap-server `broker-list.sh`

> $KAFKA_HOME/bin/kafka-topics.sh --describe --topic test --bootstrap-server `broker-list.sh`
```

- Hinweis: Eventuell muss *broker-list.sh* im Container der Kafka Shell erst noch angelegt werden. Sie finden die Datei im *kafka-docker* Git Repository.

Aufgabe 1.2 - Hinweise Docker

Hello Producer

- Producer initialisieren und eine Nachricht in den Topic schreiben:

```
> $KAFKA_HOME/bin/kafka-console-producer.sh --topic=test --broker-list=`broker-list.sh`
>> Hello World!
>> I'm a Producer writing to 'hello-topic'
```

Hello Consumer

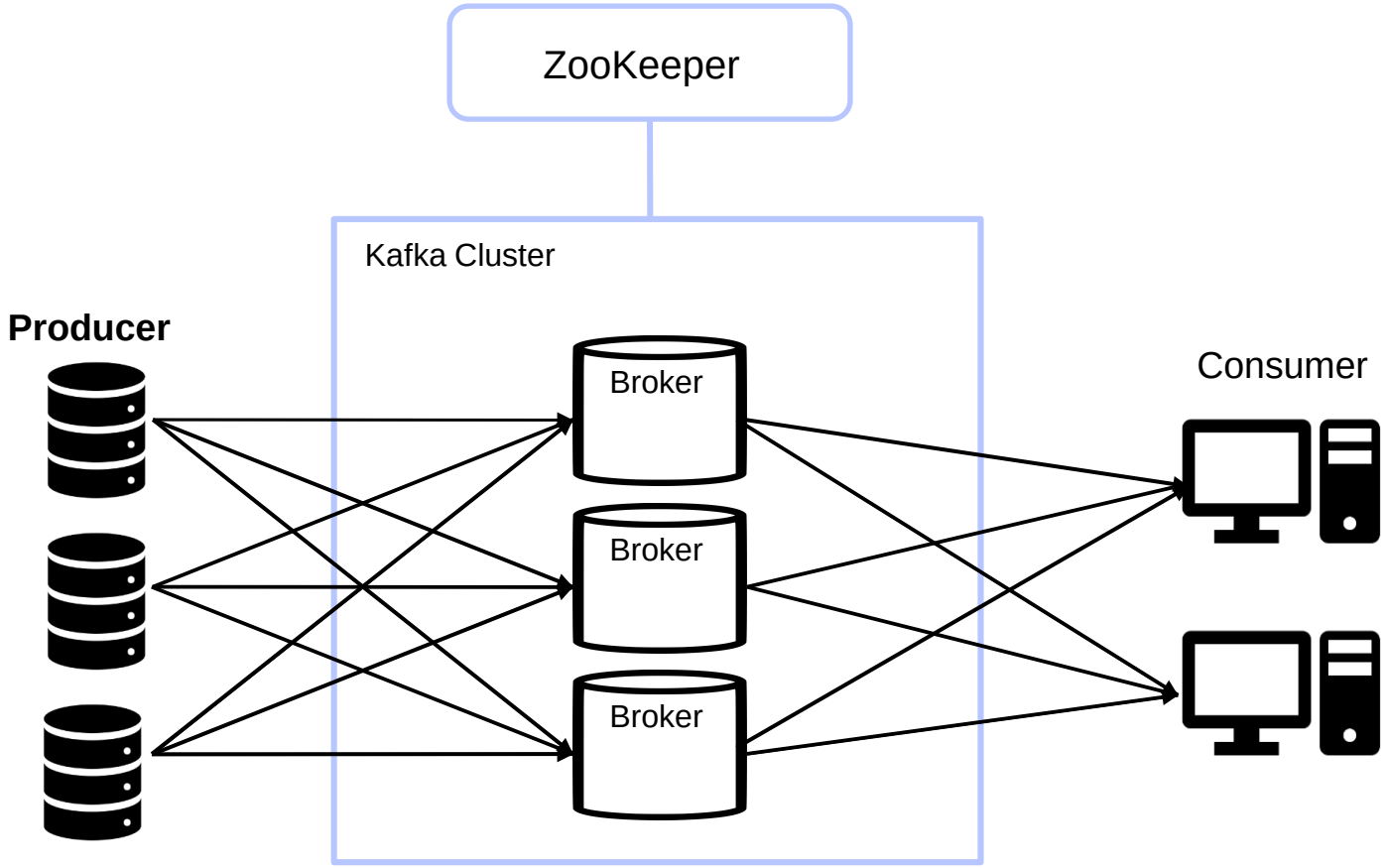
- Consumer von einem anderen Kafka Terminal initialisieren, welcher Nachrichten vom Topic liest:

```
> $KAFKA_HOME/bin/kafka-console-consumer.sh --topic=test --from-beginning --bootstrap-server `broker-list.sh`
```

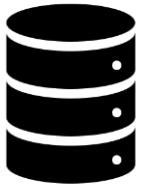
Lektion 3 – Komponenten

Kafka Komponenten

- **Producer**
- Broker
- Consumer
- Zookeeper



Kafka Komponenten: Producer

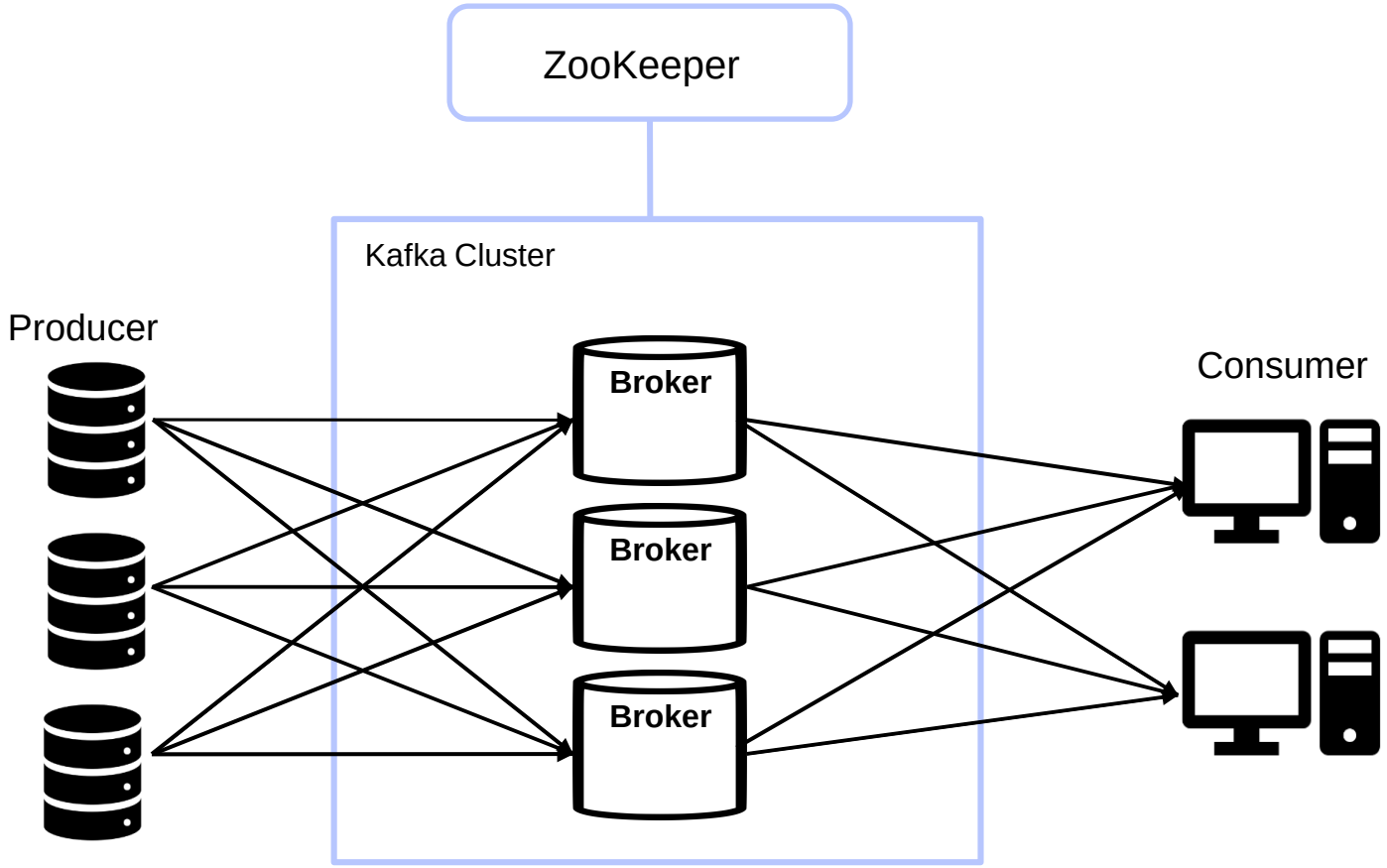


- Teil einer Anwendung
- Senden Daten an den Kafka Cluster
- Sharding der Messages (Partitionierung)
  - Über Hash Key oder Round Robin
  - Alternativ auch über eigene Strategie
  - Load Balancing: Verteilung der Last über Broker
  - Semantic Partitioning: User spezifischer Key

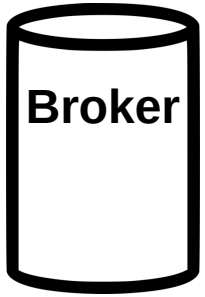
- Anbindung über:
  - Nativ: Java, C/C++, Python, Go, .Net, JMS
  - REST (Confluent)
- Zusätzlich existieren Implementierungen für viele andere Sprachen

Kafka Komponenten

- Producer
- **Broker**
- Consumer
- Zookeeper

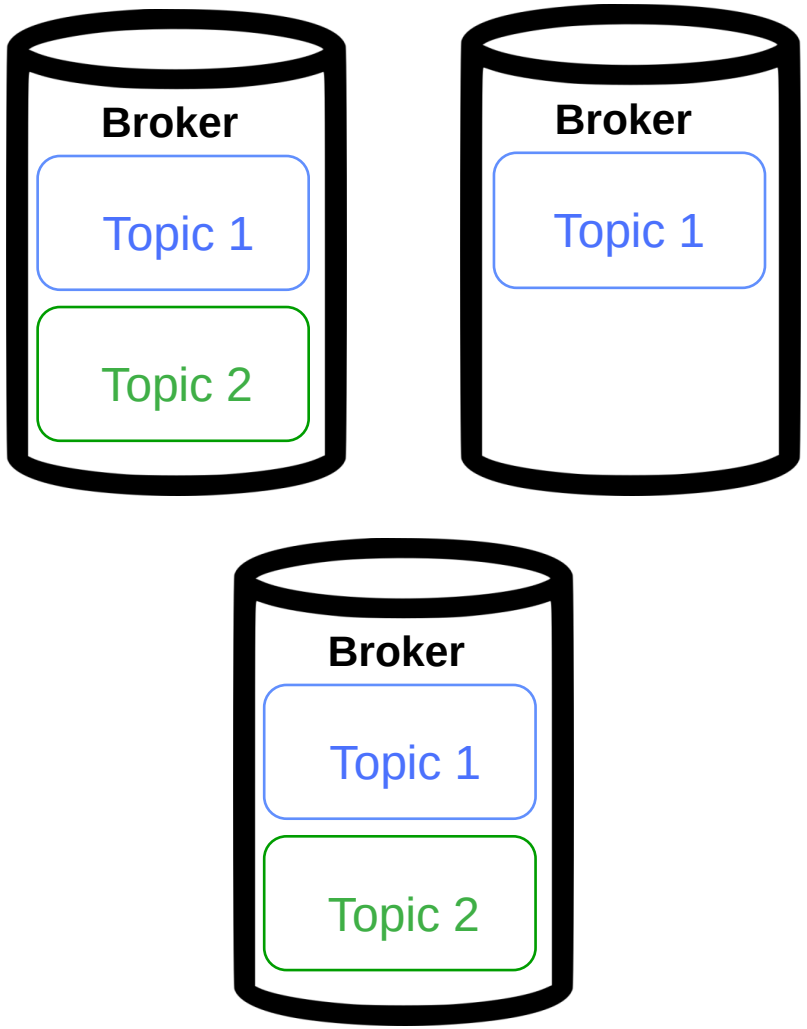


Kafka Komponenten: Broker



- Server Prozess (aktiv)
- Storage und Messaging Komponente
- Empfängt und speichert Nachrichten
- Message-Speicher: Direkt auf Hard Disk (Zero Copy)
- Existiert mehrfach pro Cluster
- Bedient Client Prozesse
- Weist Clients individuelle Offsets zu
- Verwaltet mehrere Topics und Partitionen

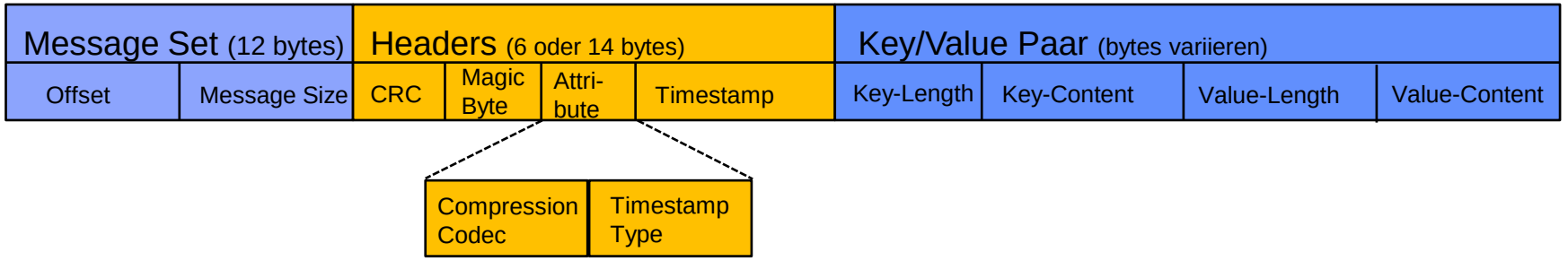
Kafka Komponenten: Topic



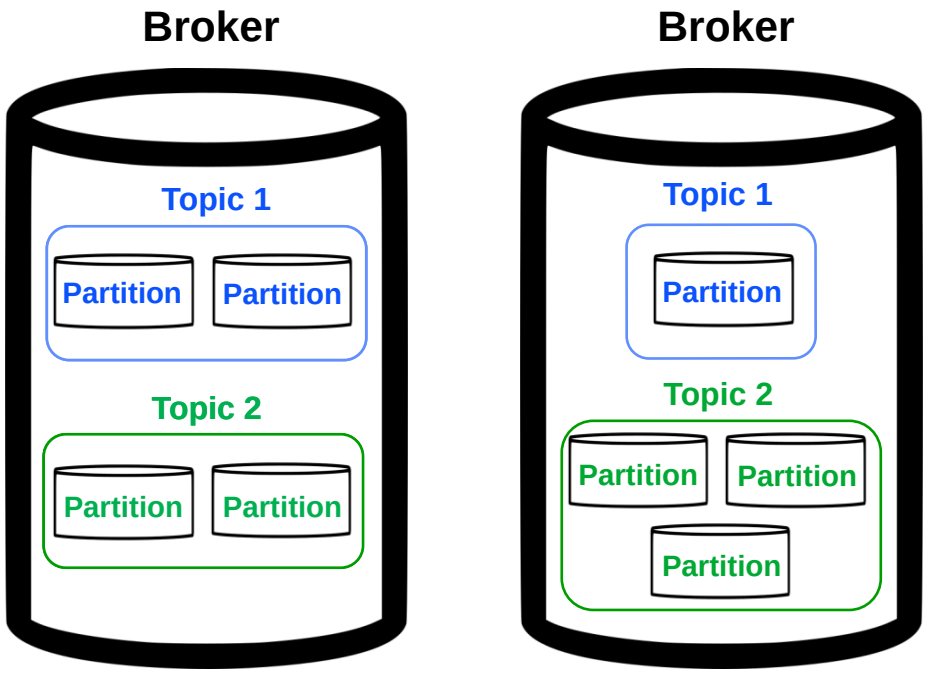
- Logischer Name eines Streams
- Gruppiert Messages
- „Beliebig“ viele pro Cluster
- Von Kafka verwaltet
- Von Entwickler(in) administriert
- Cleanup-policy: compact vs. delete
- Segment Size: head vs. tail
- Besteht aus mehreren Partitions

Kafka Komponenten: Message

- Eine Message ist ein **Key-Value** Paar; Key und Value können von beliebigem Datentyp sein
- Messages enthalten Daten und Metadaten:
  - Key/Value Paar
  - Offset
  - Timestamp
  - Compression type

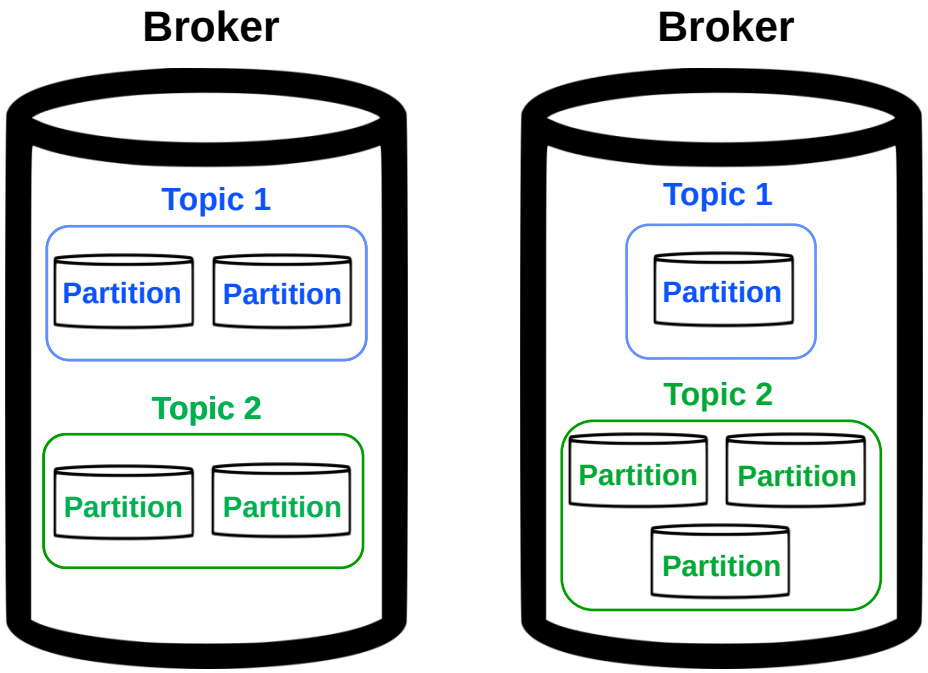


Kafka Komponenten: Partition

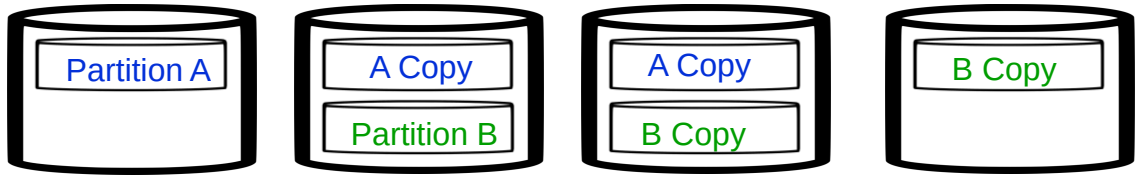


- Jede Partition enthält eine Teilmenge der Nachrichten einer Topic; üblich: Message Key zur Zuordnung einer Partition
- Lokale Ordnung innerhalb einer Partition, append-only
- Identifier innerhalb einer Partition: Offset einer Message, wachsend
- Replication zwischen Brokern möglich → Fault Tolerance

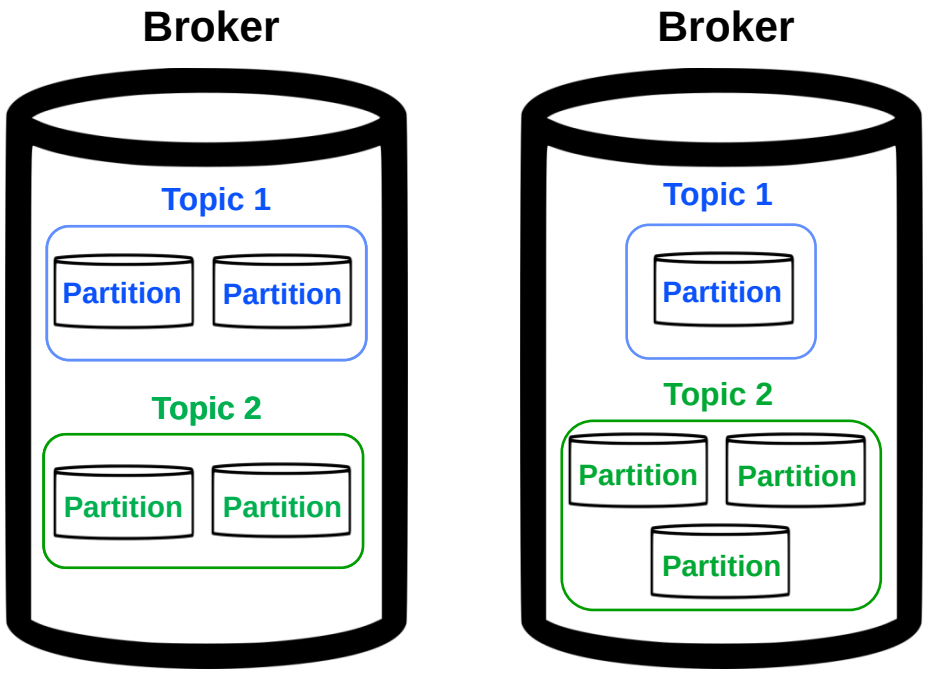
Kafka Komponenten: Partition



- Jede Partition enthält eine Teilmenge der Nachrichten einer Topic; üblich: Message Key zur Zuordnung einer Partition
- Lokale Ordnung innerhalb einer Partition, append-only
- Identifier innerhalb einer Partition: Offset einer Message, wachsend
- Replication zwischen Brokern möglich → Fault Tolerance



Kafka Komponenten: Partition

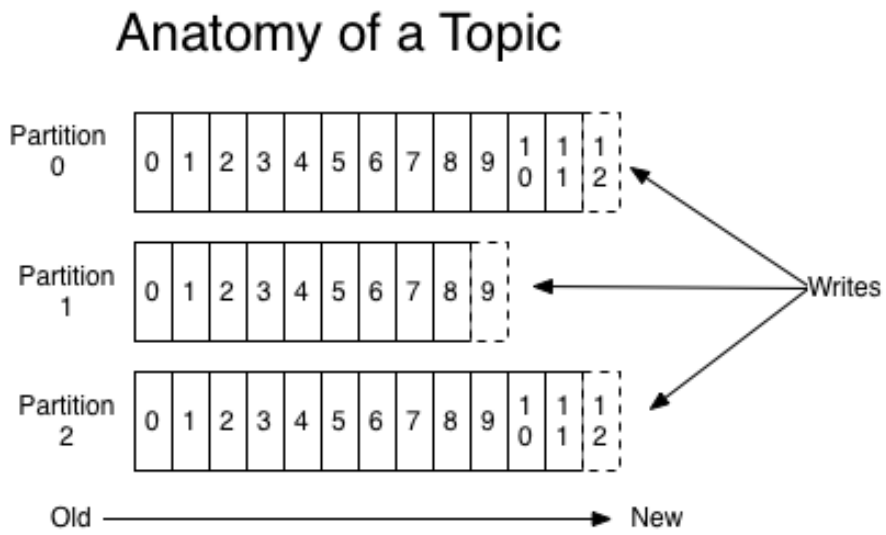


- Jede Partition enthält eine Teilmenge der Nachrichten einer Topic; üblich: Message Key zur Zuordnung einer Partition
- Lokale Ordnung innerhalb einer Partition, append-only
- Identifier innerhalb einer Partition: Offset einer Message, wachsend
- Replication zwischen Brokern möglich → Fault Tolerance
- Clients lesen nur vom Leader
- Head: in-memory
- Tail: persistent (Deletion & Compaction)
- Drift konfigurierbar

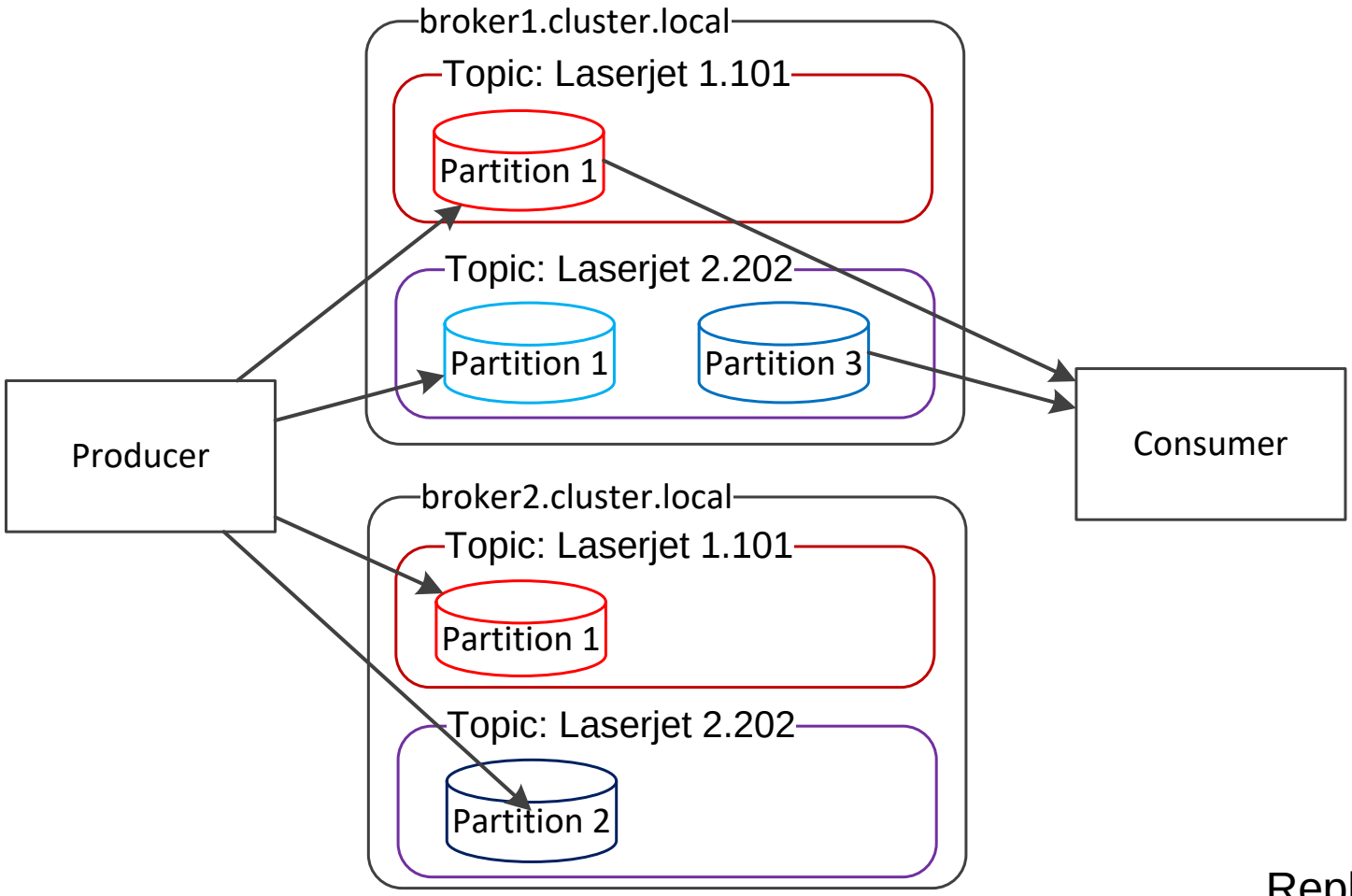
Anatomie eines Topics

Aufbau:

- Cluster besteht aus 1...n Brokern
- Ein oder mehrere Producer können in eine oder mehrere Topics schreiben
- Messages einer Topic sind in über 1...m Partitions in verschiedenen Brokern verteilt



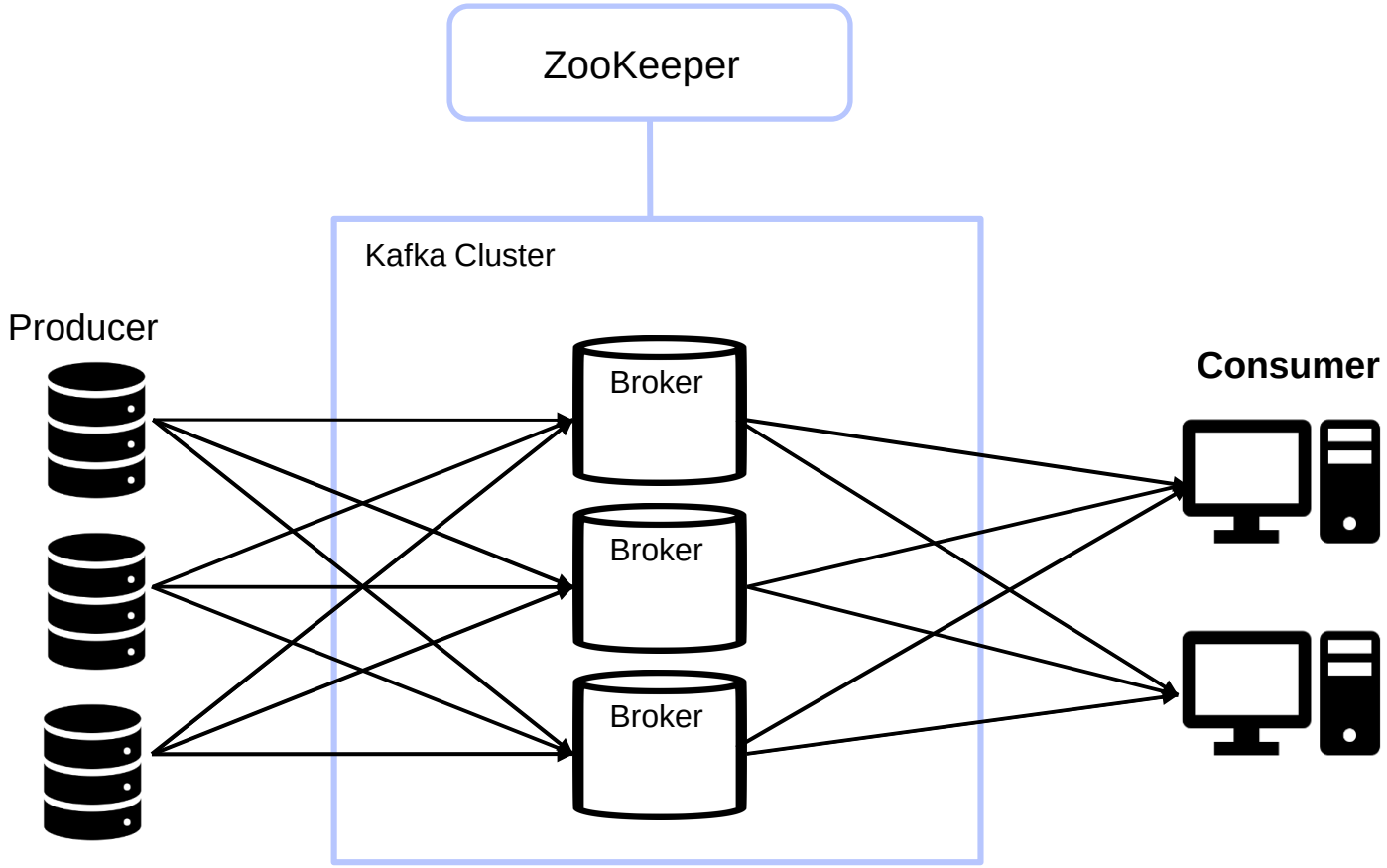
Partitionen: Printer Beispiel



Repliziert: Laserjet 1.101
Sharding: Laserjet 2.202

Kafka Komponenten

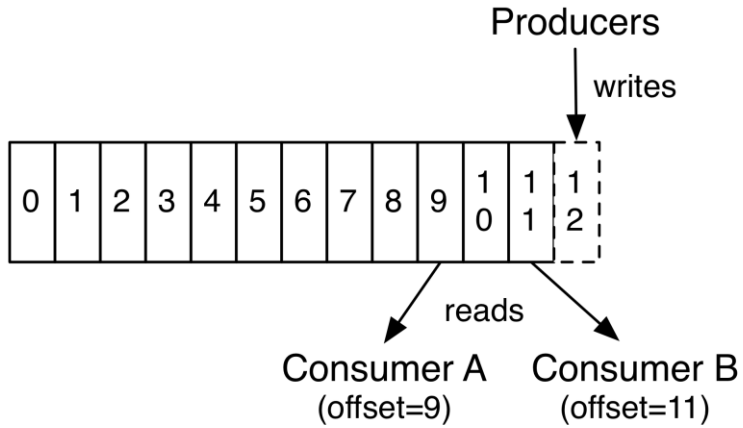
- Producer
- Broker
- **Consumer**
- Zookeeper



Kafka Komponenten: Consumer

- Abruf von Messages via **pull, single-threaded**

- **Consumer Offset**
  - *Nächste* zu lesende Nachricht
  - Pro Consumer und Partition
  - Speicher: spezielles internes Topic (oder extern)
  - Commit: automatisch (default: 5 sec – Obacht!) oder manuell



- **Semantik**:
  - At least once: Nachricht bearbeiten *danach* Commit
  - At most once: Commit, *danach* Nachricht bearbeiten
  - Exactly once: Offset im Zielsystem speichern (lokale Transaktion)

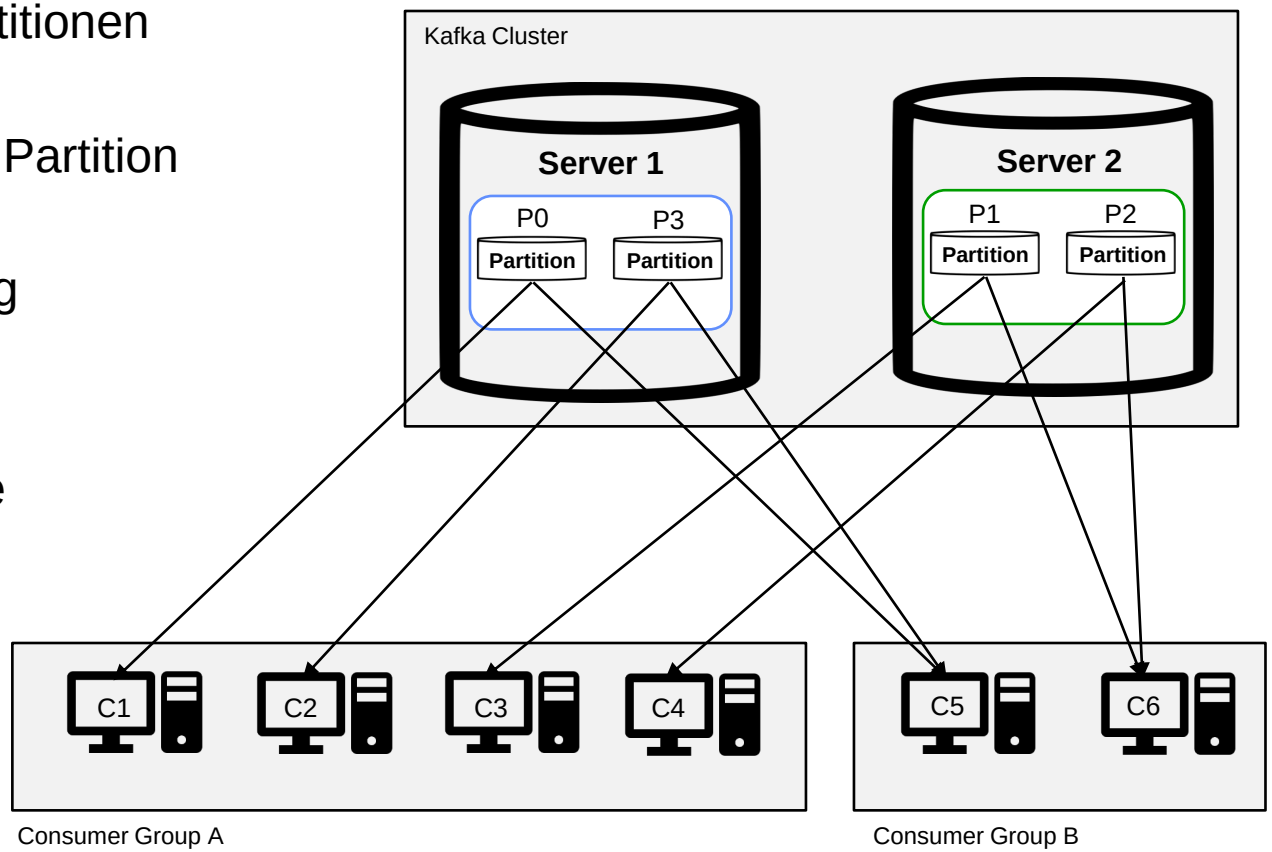
- **Verschiedene Consumer**
  - Gleichzeitiges Lesen möglich
  - Default: Alle Nachrichten im Topic an alle Consumer
  - Spezialfall: *Consumer Group*

https://docs.confluent.io/platform/current/clients/consumer.html

Kafka Komponenten: Consumer Groups

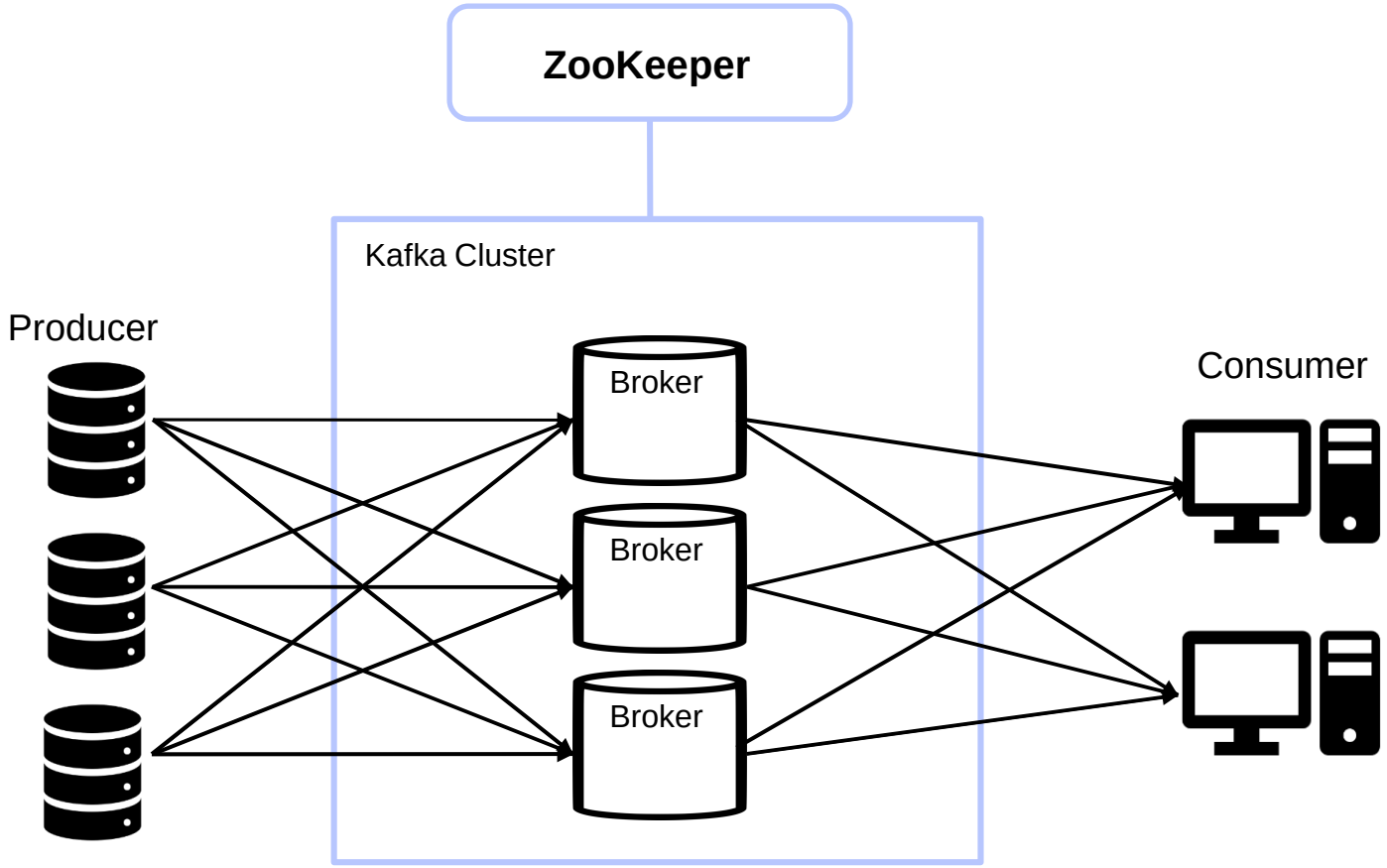
- Gruppierung: **Consumer Group**
  - Mehrere Consumer zusammenfassen → Consumer Group
  - Jeder Consumer bearbeitet nur **Subset** der Partitionen
  - Eindeutige Group ID
  - Jeder Consumer in einer Group braucht eigene Partition
    - Ein Subset von Partitionen
  - Automatisches Error-Handling & Load-Balancing

- Scaling
  - Maximal ein Consumer pro Partition pro Gruppe
  - **#Consumer ≤ #Partitions**
  - #Partition ändern: Schwer möglich
  - Besser: Neues Topic bei Release-Wechsel (API-Versionierung)



Kafka Komponenten

- Producer
- Broker
- Consumer
- **Zookeeper**



Zookeeper

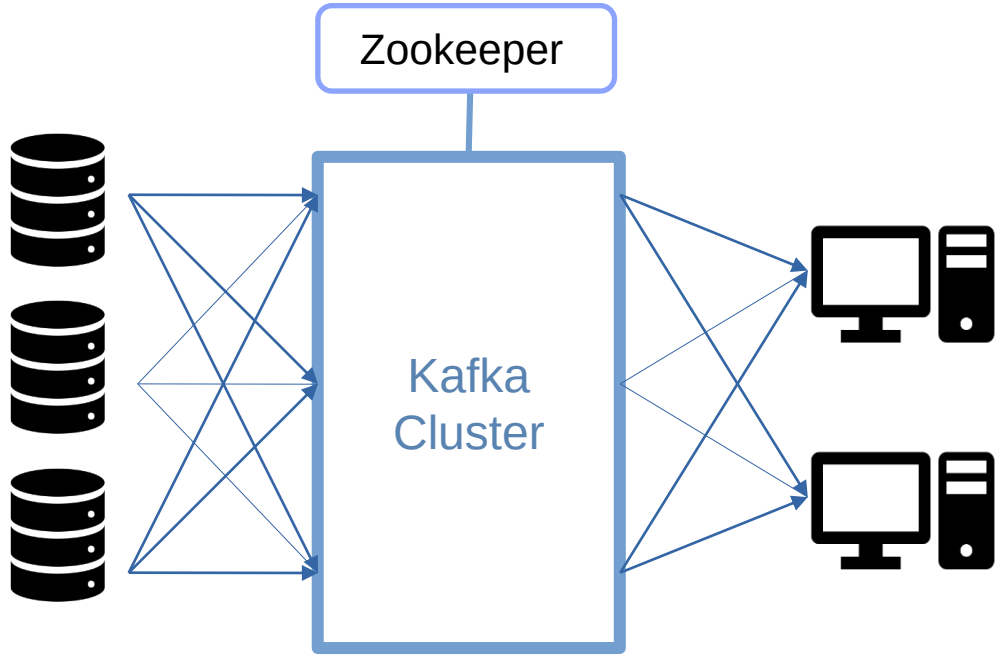
- Was ist **Zookeeper**?
  - Open Source von Apache
  - Ermöglicht verteilte Koordination
  - Kümmert sich um Konfigurationsinformationen
  - Bietet verteilte Synchronisation

- Besteht aus drei oder fünf Servern im Quorum
  - Quorum:
    - Eine replizierte Gruppe von Servern in der gleichen Applikation nennt man Quorum
    - Im replicated mode haben alle Server im Quorum eine Kopie der gleichen Config Datei

Zookeeper in Kafka

Kafka Broker nutzen **Zookeeper** für:

- Cluster Management
- Fehlerfindung und Wiederherstellung
- Speicherung von Access Control Lists (ACL)



Aufgabe 2 – Topics und Partitionen

Aufgabenstellung:

- Erstellen Sie ein Topic mit 2 Partitionen auf dem Broker, welchen Sie in Aufgabe 1 eingerichtet haben
  - Alternativer Broker: `broker-1.k.anderscore.com`
- Welche Funktion und Auswirkung haben die Parameter:
  - `segment.ms` und `segment.bytes`
  - `cleanup.policy` = `delete` (oder `compact`)
  - `retention.ms` oder `retention.bytes`
  - `min.cleanable.dirty.ratio`
- Schreiben Sie die Zahlen 0…42 in das Topic und lesen Sie sie daraus.
  Wie sind die Zahlen geordnet?

Aufgabe 2 - Hinweise

Hinweise:

- Verwenden Sie die Kafka Command Line Tools um das Topic anzulegen
- Topic erzeugen:

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic myTopic --partitions 2 --replication-factor 1
```

Aufgabe 2 - Hinweise

Nachricht senden:

```
for zahl in `seq 0 42`; do echo $zahl | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic myTopic; done
```

Hinweis: Schneller geht‘s auch mit `kafkacat` statt `kafka-console-producer.sh`, da kein JVM-Process pro Iteration erzeugt wird – kafka-console-producer.sh bringt jedoch mehr Optionen

Nachricht Lesen:

```
bin/kafka-console-consumer.sh  --bootstrap-server localhost:9092 --from-beginning --topic myTopic
```

Lektion 4 – Implementierung von Clients

Java APIs für Kafka

Java Producer API

- Kafka Client, welcher Datensätze zum Kafka Cluster published
  - Thread Safe
  - Dependency Injection Scope: Singleton
  - Pufferung bei Verbindungsverlust

Java Consumer API

- Kafka Client, welcher Datensätze aus einem Kafka Cluster konsumiert
  - Transparenz bei Fehlern von Brokern
  - Passt sich an Migrationen von Partitionen im Cluster an
  - Interagiert mit Broker und erlaubt Zugriff auf dessen Consumer Groups

Producer Eigenschaften

Name	Beschreibung
bootstrap.servers	Liste der Broker host/port Paare für initiale Verbindung zum Cluster
key.serializer/ value.serializer	Klasse zur Serialisierung von Keys/ Values. Muss das Serializer Interface implementieren
acks	Anzahl an Bestätigungen (Acknowledgements), die der Producer benötigt, bevor der Request fertig ist. **acks=0:** Producer wartet nicht auf Bestätigungen vom Server **acks=1:** Producer wartet, bis der Datensatz auf den Leader geschrieben wurde **acks=all:** Producer wartet, bis alle in-sync Replikate das Erhalten der Datensätze bestätigt haben

```
Properties props = new Properties();
props.puts(setting, value);
```

Java Producer API

anderScore trust in competence

Erstellung eines Producers:

Klasse

```
public class KafkaProducer<K,V>
```

Wichtige Eigenschaften und Senden

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "all");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
for (int i = 0; i < 100; i++)
    producer.send(new ProducerRecord<String, String>("my-topic", Integer.toString(i), Integer.toString(i)));
producer.close();
```

Nachricht an Kafka senden

- Die `send()` Methode ist non-blocking
  - Gibt Datensatz in einen Buffer von wartenden Datensätzen über und gibt sofort ein Return zurück
  - Effizienz: Batched einzelne Datensätze zusammen
  - Falls nötig kann mit `.send(record).get()` ein block geforced werden

```
ProducerRecord<String, String> record = new ProducerRecord<String, String>(“my-topic”, “myKey”, “myValue”);

Future<RecordMetadata> metadata =  producer.send(record);
producer.close();
```

- **Retries**:
  - Wie oft versucht der Producer bei einem Fehler Records erneut zu senden
  - Kann zu **Änderung in der Reihenfolge** der Nachrichten führen!
  - Anzahl der Connections kann angepasst werden

```
retry.backoff.ms=100          (Pause zwischen retries)
retries=600                   (Anzahl retries)

# default ist 5
set max.in.flight.requests.per.connection=1
```

Buffer beim Versenden

Default Linger:

- Buffer sendet sofort, auch bei ungenutztem Space
- Um die Anzahl an Requests zu verringern, kann Wartezeit konfiguriert werden
- Erhöht Effizienz bei minimaler Latenz

```
# größer als 0
linger.ms = 1
```

Buffer Größe:

- Gesamtmenge an Speicher, welcher dem Producer für den Buffer zu Verfügung gestellt wird
- Wenn der Buffer voll ist, werden Send Requests geblockt (TimeoutException)

```
buffer.memory
max.block.ms
```

Callbacks

- `send(record)` **äquivalent zu** `send(record, null)`

→ **Callback in zweitem Parameter überliefern**

```
ProducerRecord<byte[],byte[]> record = new ProducerRecord<String,String>("the-topic", key, value);
producer.send(record,
                new Callback() {
                    public void onCompletion(RecordMetadata metadata, Exception e) {
                        if(e != null) {
                           e.printStackTrace();
                        } else {
                            System.out.println("The offset of the record we just sent is: " + metadata.offset());
                        }
                    }
                });
```

- **Reaktion auf Fehler**:
  - Fehler: Metadata ist `null`
  - Kein Fehler: Exception ist `null`

Consumer Eigenschaften

Name	Beschreibung
bootstrap.servers	Liste der Broker host/port Paare für initiale Verbindung zum Cluster
key.deserializer/ value.deserializer	Klasse zur Deserialisierung von Keys/ Values. Muss das Deserializer Interface implementieren
group.id	Zeigt an, zu welcher Consumer Group der Consumer gehört
enable.auto.commit	Bei true triggered der Consumer offset commits

```
Properties props = new Properties();
props.setProperty(setting, value);
```

Java Consumer API

anderScore trust in competence

Erstellung eines Consumers:

Klasse

```
public class KafkaConsumer<K,V>
```

Wichtige Eigenschaften und Polling

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");
props.setProperty("group.id", "test");
props.setProperty("enable.auto.commit", "true");
props.setProperty("auto.commit.interval.ms", "1000");
props.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.setProperty("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("foo", "bar"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(),
record.value());
}
```

Nachricht aus Kafka lesen

- **Die `poll()` Methode gibt alle verfügbaren Nachrichten zurück**
  - Bis zu der maximalen Größe per Partition

```
# default 1048576 bytes
max.partition.fetch.bytes
```

- Eine zu hohe Anzahl an Partitionen kann extreme Mengen an Daten zurückgeben
  - Gesamtmenge von Datensätzen in einem einzelnen Poll kann reduziert werden (Chunking)

```
max.poll.records
```

Aufgabe 3 – Producer und Consumer

Aufgabenstellung:

- Erzeugen Sie ein neues Java Projekt und binden Sie den Kafka Client ein
- Verbinden Sie sich mit Ihrem Kafka Broker
- Implementieren Sie einen Consumer: Lesen Sie alle Nachrichten aus dem Topic „HelloWorld“ aus
- Geben Sie die Nachrichten auf der Konsole aus
- Implementieren Sie einen Producer

Aufgabe 3 - Hinweise

- Im einfachsten Fall können Sie Kafka-Events ohne weitere Frameworks konsumieren.
- Hierzu müssen die entsprechenden Bibliotheken in das Projekt angebunden werden.

```
# Maven dependencies
<dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>3.2.3</version>
</dependency>
```

- Erstellen Sie ein neues Maven Projekt z.B. mit:

```
mvn archetype:generate -DgroupId=gs -DartifactId=kafka \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DinteractiveMode=false
```

- Fügen Sie dann die Maven Dependency der Datei pom.xml hinzu
- Nutzen Sie die folgenden Klassen:

```
org.apache.kafka.clients.consumer.KafkaConsumer
org.apache.kafka.clients.producer.KafkaProducer
```

Einbindung in Spring Boot

Spring for Kafka

- Version 2.7 seit 2016
- Basiert auf Kernkonzepten von Spring (z.B. DI, Annotationen, Templates)
- Wird verwendet, um Kafka basierte Messaging Lösungen zu entwickeln
- Bietet Template für High-Level Abstraktion für das Senden von Messages
- Bietet Support für Message-driven POJOs
- Es kann plain Java zum Versenden und Empfangen von Messages genutzt werden
- Java mit Konfiguration
- Oder am simpelsten mit **Spring Boot**

Kafka mit Spring Boot - Beispiel

anderScore trust in competence

Dependencies

- spring-kafka JAR und alle seine Dependencies
- Am einfachsten mit Maven

```
<dependency>
   <groupId>org.springframework.kafka</groupId>
   <artifactId>spring-kafka</artifactId>
   <version>2.9.1</version>
</dependency>
```

Spring Boot Beispiel:

- Anwendung sendet 3 Nachrichten an ein Topic
- Nachrichten werden empfangen
- Anwendung wird beendet

Spring Boot - Beispiel

Voreinstellungen

- Es wird Group Management benutzt, um Topics und Partitions Consumern zuzuordnen
- Hierfür muss eine Gruppe erstellt werden:

```
spring.kafka.consumer.group-id=foo
```

- Der Container könnte nach dem Verstand der Nachrichten starten
- Es muss ein Offset eingestellt werden, um sicher zu gehen, dass die neue Consumer Group die Nachrichten auch bekommt:

```
spring.kafka.consumer.auto-offset-reset=earliest
```

Spring Boot - Beispiel

```
@SpringBootApplication
public class Application implements CommandLineRunner {

	public static Logger logger = LoggerFactory.getLogger(Application.class);

	public static void main(String[] args) {
		SpringApplication.run(Application.class, args).close();
	}

	@Autowired private KafkaTemplate<String, String> template;
	private final CountDownLatch latch = new CountDownLatch(3);

	@Override
	public void run(String... args) throws Exception {
		this.template.send("myTopic", "foo1");
		this.template.send("myTopic", "foo2");
		this.template.send("myTopic", "foo3");
		latch.await(60, TimeUnit.SECONDS);
		logger.info("All received");
	}

	@KafkaListener(topics = "myTopic")
	public void listen(ConsumerRecord<?, ?> cr) throws Exception {
		logger.info(cr.toString());
		latch.countDown();
	}
}
```

Aufgabe 4 – Kafka mit Spring Boot

Aufgabenstellung:

- Erstellen Sie ein Spring Boot Projekt
- Binden Sie die Kafka Bibliotheken ein und konfigurieren Ihren Broker auf Port 9092
- Konsumiren Sie das Topic „Hallo Welt“ und geben Sie es auf der Konsole aus
- Senden Sie eine Nachricht an das Topic

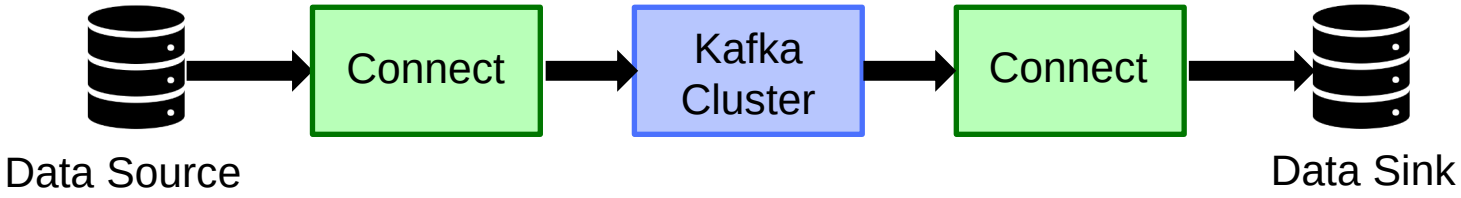
Aufgabe 4 - Hinweise

- **Properties**: `group-id` und `auto-offset-reset` anpassen
- Wie ein **neues Spring-Boot-Projekt** anzulegen ist, wird hier beschrieben:
  - https://spring.io/guides/gs/spring-boot/
  - https://start.spring.io

Lektion 5 – Analyse und Transformation

Kafka Connect

- Framework zum Streaming von Daten zwischen Kafka und externen Datensystemen
- Herausgeber: confluent
- Skalierbar, einfach und zuverlässig
- **Use cases:** Komplette SQL Datenbank zu Kafka streamen, Streamen von Kafka Topics zu Elasticsearch für Indexierung, …

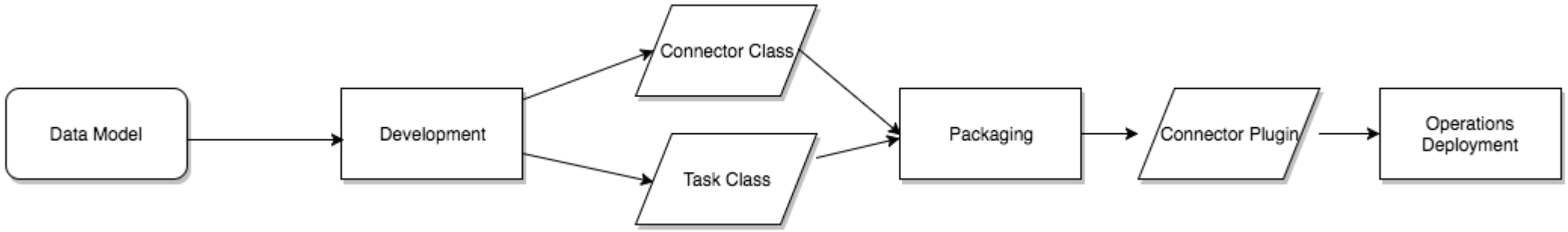


Kafka Connect vs. Producer/ Consumer API

- Im Kern ist Kafka Connect ein Kafka Client der die standard Producer und Consumer APIs benutzt
- Vorteile gegenüber „DIY“ Producer und Consumer:
  - Getestete Connectors für gebräuchliche Datenquellen
  - Unterstützt fault tolerance und automatisches load balancing
  - Keine komplizierte Implementierung, nur configuration files für Kafka Connect nötig
  - Einfach erweiterbar und anpassbar durch Entwickler

Kafka Connect

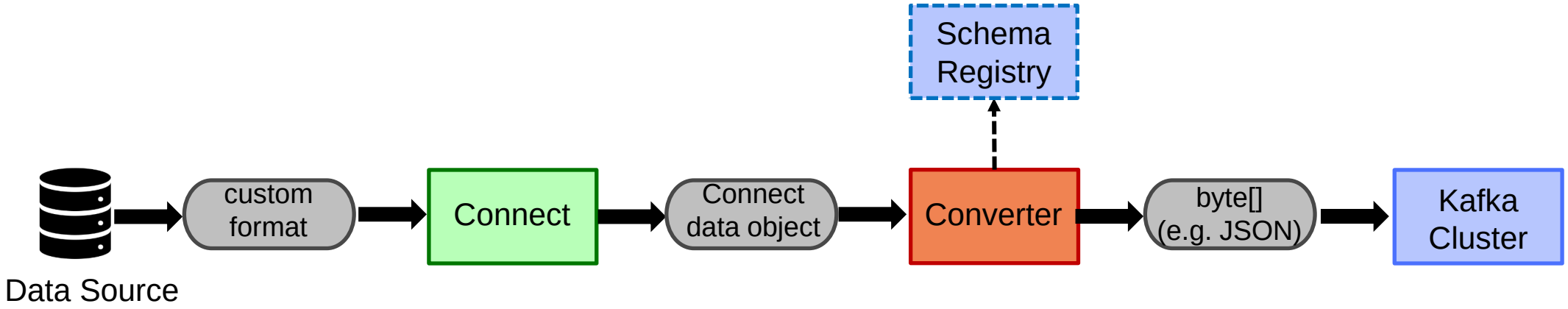
- **Connector:** Logical Job bzw. Plugin zum Managen vom Datenaustausch zwischen Kafka und anderem System
  - *Connector Sources:* Datenübertragung vom externen System zu Kafka (Producer Client)
  - *Connector Sinks:* Übertragung von Kafka Daten in ein externes System (Consumer Client)
- **Task:** Connector Jobs werden in Tasks zerlegt, die den Datenaustausch durchführen (stateless)
- **Workers:** Laufende Prozesse für Connectors und Tasks (standalone vs. distributed)



Quelle: https://docs.confluent.io/platform/current/connect/concepts.html#:~:text=handles%20connector%20errors-,Connectors,defined%20in%20a%20connector%20plugin.

Kafka Connect

- **Converter:** Stellt das Datenformat bereit, das von Kafka gelesen bzw. in Kafka geschrieben wird
  - Konverter zwischen Kafka und Data Source bzw. Data Sink (z.B JSON Converter)
  - Entkoppelt von Connectors: Beliebiger Connector kann für beliebiges Serialisationsformat verwendet werden
- **Transforms:** Anpassung von Nachrichten von oder zu einem Connector (z.B. Filter)
- **Dead Letter Queue**: Behandlung von Connector Fehlern *(auch für Consumer relevant!)*



Kafka ist eine Streaming Plattform

- **Stream** = unbeschränkte, kontinuierlich updatende Datenmenge
- Streams von Datensätzen publishen und abonnieren
  - ähnelt einer Message Queue oder einem Enterprise Messaging System
- **Kafka Streams API:** Java Bibliothek zum Aufbau verteilter Stream processing applications in Kafka-Clustern

→ **Einfache Anwendung**

- Speichern von Datensatz-Streams (fault tolerance)
- Verarbeiten von Datensätzen sobald diese auftreten

Anwendung von Streams

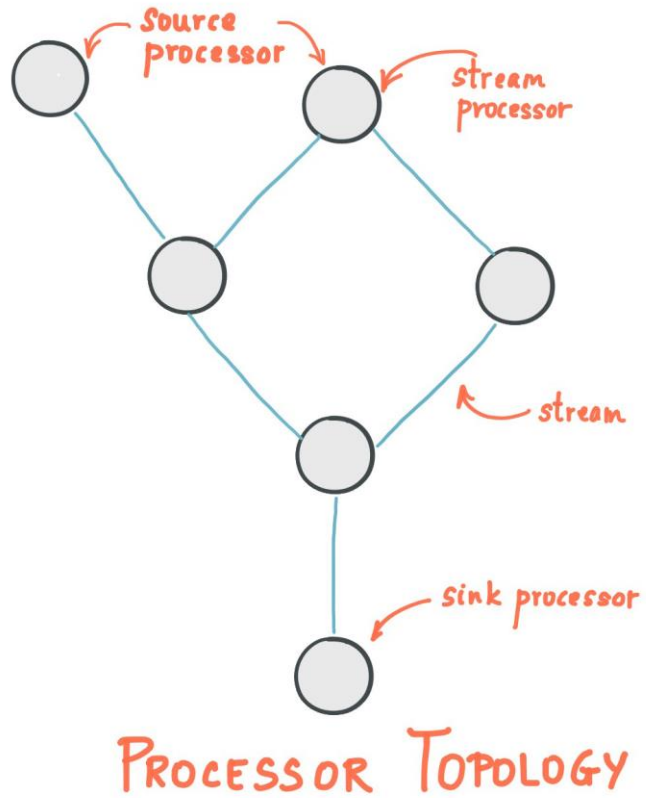
- Real Time Streaming Datapipelines
  - Zuverlässiger Austausch von Daten zwischen Systemen oder Anwendungen
- Real Time Streaming Anwendungen
  - Auf Daten Streams reagieren oder diese transformieren

Kafka Streams

- Ähnlich zu Spark Streaming, Apache Storm und Co.
- KafkaStreams DSL (map, flatMap, count, …) vs. low-level Processor API
- Abstraktion: KStream, KTable, ksqlDB

```java
public class WordCountApplication {

    public static void main(final String[] args) throws Exception {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, String> textLines = builder.stream("TextLinesTopic");
        KTable<String, Long> wordCounts = textLines
            .flatMapValues(textLine -> Arrays.asList(textLine.toLowerCase().split("\\W+")))
            .groupBy((key, word) -> word)
            .count(Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as("counts-store"));
        wordCounts.toStream().to("WordsWithCountsTopic", Produced.with(Serdes.String(), Serdes.Long()));

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();
    }

}
```

Quelle: https://kafka.apache.org/documentation/streams/



Quelle: https://kafka.apache.org/20/documentation/streams/core-concepts

KStream vs. KTable

- **KStream**
  - Abstraktion des Record Streams
  - Alle Messages werden als Insert betrachtet
- **KTable**
  - Abstraktion des Changelog Streams
  - Messages werden als Insert bzw. Update (sog. „Upsert“) betrachtet
- **Beispiel**: Summation
  - Nachricht 1: {id: 'apple', value: 1}
  - Nachricht 2: {id: 'apple', value: 2}
  - Ergebnis:
    - KStream: 3 (Summe der Records)
    - KTable: 2 (Update für Nachricht mit ID 'apple')

Aufgabe 5 – Kafka Streams

Beschreibung

- Kafka-Streams erlauben die „Echtzeit“-Auswertung und Transformation von Daten,
  die auf Kafka-Topics veröffentlicht werden
- In der Dokumentation wird die Verwendung zur Zählung der Worte im Stream verwendet:
  https://kafka.apache.org/documentation/streams/

Aufgabenstellung:

- Starten Sie das Beispiel aus der Dokumentation
- Zählen Sie die Wörter im Topic „Hello World“ auf dem Broker broker-1.k.anderscore.com (Port 9092)

Aufgabe 5 - Hinweise

anderScore trust in competence

Hinweise:

- Erzeugen Sie die Projektstruktur wie in Aufgabe 4
- Für Kafka-Streams wird eine weitere Maven Dependency benötigt:

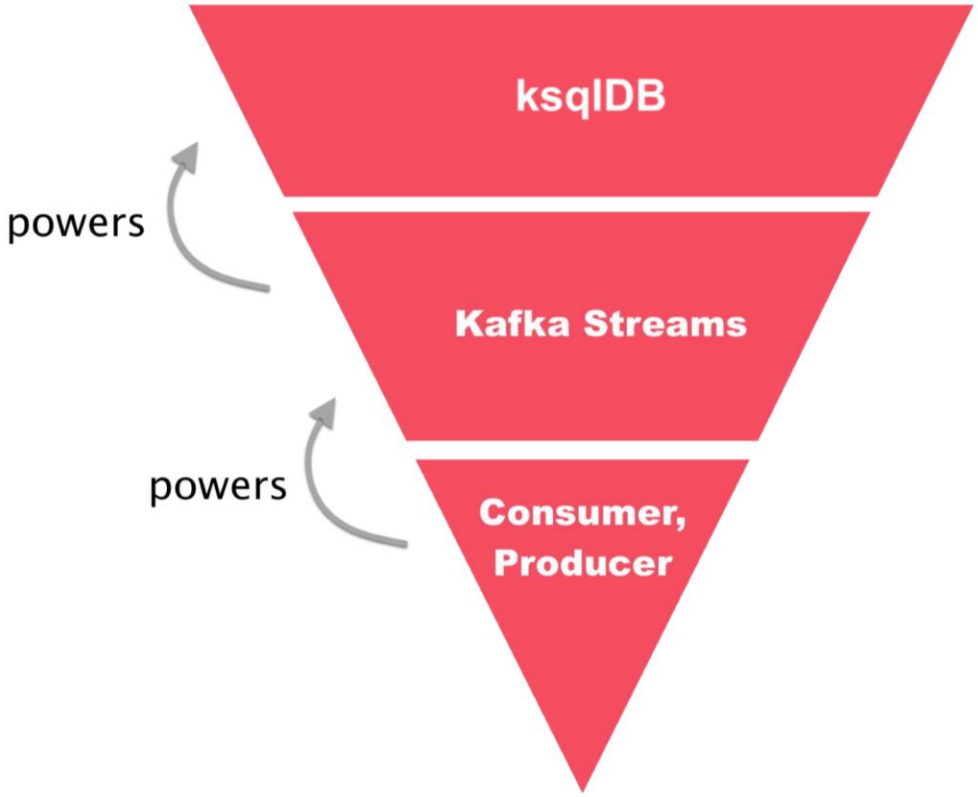
```
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-streams</artifactId>
<version>3.2.3</version>
</dependency>
```

- Die Streams aktualisieren sich, wenn Sie Nachrichten senden.
- Das Codebeispiel schreibt das Ergebnis in den Stream `WordsWithCountsTopic`, den Sie konsumieren müssen.
- Die Ergebnisse erscheinen innerhalb des eingestellten Intervalls – per default müssen Sie ein paar Sekunden warten.

Event Streaming Database

- Hilft beim Erstellen komplexer Stream Processing Anwendungen mit Kafka
- Abfragen, Lesen, Schreiben und Verarbeiten von Daten
- Deklarative Definition von Tabellen, Streams und Konnektoren
- Dynamische Joins
- Lightweight SQL Syntax
- SQL-like Interface
- Runtime: ksqlDB Engine
- Community Component von Confluent
- Alternative / Aufbau zu Kafkas Stream API
- Besser für:
  - Streaming ETL Pipelines
  - Reaktion auf kontinuierliche Real-Time Business Requests
  - Anomalien erkennen

ksqlDB



ease of use

```
CREATE STREAM, CREATE TABLE,
SELECT, JOIN, GROUP BY, SUM, …

KStream, KTable,
filter(), map(), flatMap(),
join(), aggregate(), …

subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

flexibility

Quelle: https://docs.ksqldb.io/en/latest/concepts/ksqldb-and-kafka-streams

ksqlDB - Beispiel

ksqlDB:

```
CREATE STREAM fraudulent_payments AS
SELECT fraudProbability(data) FROM payments WHERE fraudProbability(data) > 0.8
EMIT CHANGES;
```

Kafka Streams:

```
public class FraudFilteringApplication {
    StreamsBuilder builder = new StreamsBuilder();
    KStream<String, Payment> fraudulentPayments = builder
        .stream("payments-topic")
        .filter((customer ,payment) -> payment.getFraudProbability() > 0.8);
        .to("fraudulent-payments-topic");

    Properties config = new Properties();
    config.put(StreamsConfig.APPLICATION_ID_CONFIG, "fraud-filtering-app");
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092");

    new KafkaStreams(builder.build(), config).start();
}
```

Nach: https://docs.ksqldb.io/en/latest/concepts/ksqldb-and-kafka-streams

Lektion 6 – Broker Operations

Kafka Broker Konfigurieren

Config Files

Jeder Broker hat eine **Config Datei**

- `.property` file extension
- Kann in der Datei oder programmatisch angepasst werden
- Nutzt Key / Value Paare

Beispiel:

- /usr/local/kafka/config/server.properties

Essentielle Konfiguration

anderScore trust in competence

broker.id

- ID des Brokers (unique)
- Bekommt ID automatisch, wenn nicht explizit gesetzt
- Vermeidung von Konflikten zwischen automatisch und explizit gesetzten IDs:
  `reserved.broker.max.id +1`

log.dirs

- Verzeichnis der log Daten
- Wenn nicht genutzt, wird stattdessen das Verzeichnis in `log.dir` benutzt

zookeeper.connect

- Definiert den Zookeeper connection string
  `hostname:port`
- Um Verbindung zu anderen Zookeeper Nodes zu ermöglichen, falls eine Verbindung unterbrochen ist:
  `hostname1:port1, hostname2:port2…`

Broker Konfiguration

- advertised.host.name
- advertised.listeners
- advertised.port
- auto.create.topics.enable
- auto.leader.rebalance.enable
- background.threads
- compression.type
- control.plane.listener.name
- delete.topic.enable
- host.name
- leader.imbalance.check.interval.seconds
- leader.imbalance.per.broker.percentage
- listeners
- log.flush.interval.ms
- log.flush.offset.checkpoint.interval.ms
- log.flush.start.offset.checkpoint.interval.ms
- log.retention.bytes/hours/minutes/ms
- log.roll.hours/ms
- log.roll.jitter.hours /ms
- log.segment.bytes
- log.segment.delete.delay.ms
- message.max.bytes
- min.insync.replicas
- num.io.threads
- num.network.threads
- num.recovery.threads.per.data.dir
- num.replica.alter.log.dirs.threads
- offsets.metadata.max.bytes
- offsets.commit.required.acks
- offsets.topic.segment.bytes
- port
- queued.max.requests
- quota.consumer.default
- quota.producer.defaults
- replica.fetch.min.bytes
- request.timeout.ms
- socket.receive.buffer.bytes
- socket.request.max.bytes
- transaction.max.timeout.ms
- unclean.leader.election.enable
- zookeeper.connection.timeout.ms
- zookeeper.max.in.flight.requests
- zookeeper.session.timeout.ms

- **Gesamte Liste:** https://kafka.apache.org/documentation/#brokerconfigs

Updaten der Broker Konfiguration

anderScore trust in competence

Live Update

Einige Configs können ohne Broker Restart geändert werden

- ready-only: Nur mit Neustart
- per-broker: Dynamisch für jeden Broker einzeln
- cluster-wide: Dynamisch gesamtes Cluster oder einzelne Broker

Beispiel Live Update:

- Broker id 0
- Anzahl der Log Cleaner Threads

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type brokers --entity-name 0 --alter --add-config log.cleaner.threads=2
```

Topics verwalten

- **Default:** Topics werden automatisch erstellt, wenn sie das erste mal von einem Client benutzt werden
  - Replication factor ist 1, eine einzelene Partition
  - In production environments: Ggf. auto.create.topics.enable deaktivieren
- **Topic manuell erstellen** (vgl. Aufgabe 2):

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic myTopic --partitions 2 --replication-factor 1
```

- **Topic ändern**:

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --alter --topic myTopic --partitions 3 --replication-factor 2
```

Achtung: Ändern der Anzahl an Partitionen kann zu Problemen in der Anwendungslogik führen!

Topics verwalten

- **Topic löschen:**

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic myTopic
```

- Alle Broker müssen delete.topic.enable auf true gesetzt haben
  - Default ist false, falls es nicht gesetzt ist, wird Löschkommando ignoriert
- Alle Broker müssen laufen, damit der Löschvorgang erfolgreich ist

Topics verwalten: Einstellungen

anderScore trust in competence

Topic Level Einstellungen: Beispiele

- cleanup.policy
- compression.type
- delete.retention.ms
- file.delete.delay.ms
- flush.messages
- flush.ms
- follower.replication.throttled.replicas
- index.interval.bytes
- leader.replication.throttled.replicas
- min./max.compaction.lag.ms
- max.message.bytes
- message.format.version
- message.timestamp.difference.max.ms
- message.timestamp.type
- min.cleanable.dirty.ratio
- min.insync.replicas
- preallocate
- retention.bytes
- retention.ms
- segmentet.bytes
- segment.index.bytes
- segment.jitter.ms
- segment.ms
- unclean.leader.election.enable
- message.downconversion.enable

- **Gesamte Liste:** https://kafka.apache.org/documentation/#topicconfigs

Topic Konfiguration

- Relevante Konfiguration
  - Server Default
  - Per-Topic Override

Der Override kann beim Erstellen eines Topics in der Kafka Shell eingestellt werden

- `--config`
- Eine oder mehrere Einstellungen

```
> bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-topic --partitions 1 \
    --replication-factor 1 --config max.message.bytes=64000 --config flush.messages=1
```

Overrides können auch später mit `add-config` konfiguriert werden

- Im folgenden Beispiel wird die `max.message.bytes` upgedated:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name my-topic
    --alter --add-config max.message.bytes=128000
```

- Overrides überprüfen mit `--describe`:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name my-topic --describe
```

- Override entfernen mit `--alter --delete-config`:

```
> bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --entity-name my-topic
    --alter --delete-config max.message.bytes
```

Lektion 7 – Best Practices

Best Practices

Producer

- Automatische Retries möglichst abschalten (Reordering)
- acks = all für “Konsistenz über Verfügbarkeit” (vgl. CAP Theorem)
- Mehrere Bootstrap Server angeben (Fehlertoleranz)

Consumer

- Nicht zu viele Daten auf einmal pollen (Netzwerklast, OutOfMemoryError)
- Auto-Commit beachten und möglichst abschalten
- Topics mit Consumer Offsets richtig konfigurieren (Compaction und Deletion!)
- At-least-once-Semantik meist beste Wahl
- Mehrere Bootstrap Server angeben (Fehlertoleranz)

Nach: https://media.ccc.de/v/froscon2018-2213-apache_kafka_lessons_learned

Best Practices

Broker

- Defaults setzen und bei Bedarf überschreiben (Fehlervermeidung)

Topics

- Deletion vs. Compaction **fachlich** entscheiden
- retention.ms wirkt nur bei Deletion, max.cleanable.dirty.ratio nur bei Compaction
- Automatisiert anlegen (z.B. Skript)
- Richtwert: ~ 10 bis 30 Partitionen pro Topic

Streams

- Dead Letter Queue vorsehen (z.B. Topic)

Nach: https://media.ccc.de/v/froscon2018-2213-apache_kafka_lessons_learned

Lektion 8 – Ausblick

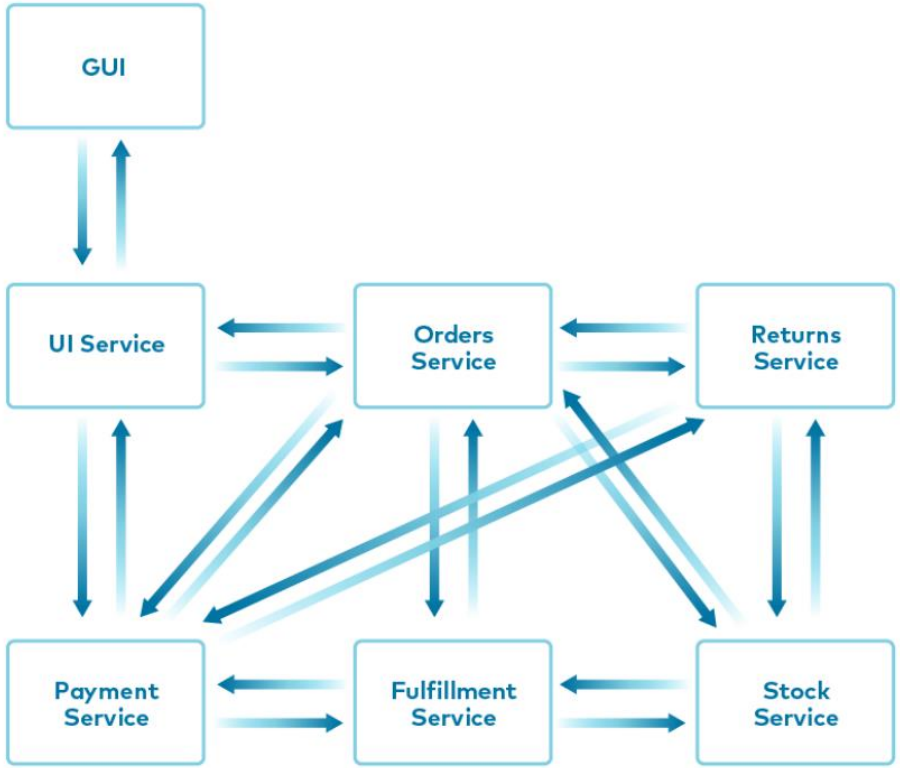
Microservice Architektur

Microservices:

- Fachlicher Schnitt (Bounded Contexts)
- Weniger Verantwortlichkeiten → leichter austauschbar
- Potential für bessere Skalierbarkeit und Fehlertoleranz
- Lose Kopplung zwischen Services

Beispiel:

- Simples Business System in Microservice Architektur
- Asynchrone Kommunikation über Messages
- Eventbasiert



Quelle: WP Microservices in the Apache Kafka Ecosystem Confluent 2017

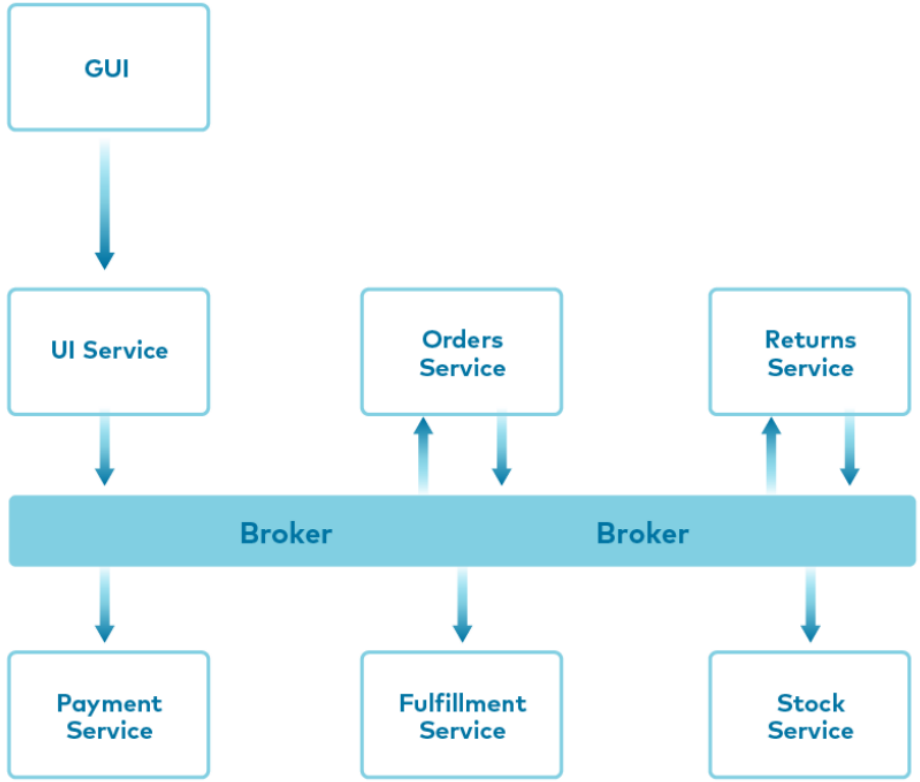
Microservices mit Kafka

Microservice Environment

- Event-basiert
- Request / Response basiert
- **Hybrider Ansatz:**
  - Service Discovery
  - Synchrone Vorgänge
  - Asynchrone, Event-basierte Flows
  - Anpassbarkeit
  - Skalierbarkeit

Praxisbeispiel:

- Asynchroner eMail Service mit Kafka Streams
- Order und Payment Stream joinen
- Ergebnis zu einem Lookup Table von Kunden joinen
- E-Mail zu jedem resultierenden Touple versenden



Quelle: WP Microservices in the Apache Kafka Ecosystem Confluent 2017

Apache Avro - Serialization

anderScore trust in competence

Apache Avro

- Data Serialization System (& RPC, Container)
- Vergleichbar mit: Thrift, Protocol Buffers
- Aktuell: 1.11.1
- Bindings: Java, Python, C, C++, C#, JavaScript, … (3rd party)
- Teil des Hadoop Projekts
- Serialization Format für persistente Daten
- Wire Format für Nodes und Clients

Schema für Daten in JSON

- Getrennt von den Nutzdaten
- Code-Generatoren (code-first, contract-first)
- Gemeinsames Schema für verschiedene µServices

Apache Avro – Beispiel für Schemata

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
  ]
}
```

Quelle: https://en.wikipedia.org/wiki/Apache_Avro

Apache Avro – Integration mit Kafka

Confluent Schema Registry (OpenSource)

- Historisierung, Java & RESTful API, Migration
- Kein Transport mit den Daten (Performance)
- Persistenz: Kafka-Topic
- Command Line Client

Avro als wire-format?

- Einheitliches Schema für Daten zwischen Systemen
- **Pro:**
  - Integration (confluent platform)
- **Contra:**
  - XML & JSON deutlich weiter verbreitet

Aufgabe 6 - Avro

Beschreibung:

- Austausch von Nachrichten zwischen zwei Services über Kafka
- Serialisierung und Deserialisierung über Avro mit JSON-Schemata
- Generierung von Java Klassen aus Metadaten

- Die Confluent Platform enthält eine Avro Schema Registry, mit der Schemas unternehmensweit verwaltet werden können.
- Die OpenSource-Ausgabe der Confluent-Platform ist auf `broker-2.k.anderscore.com` installiert.
- Weitere Infos: https://docs.confluent.io/current/schema-registry/docs

Aufgabe 6 - Avro

Aufgabenstellung:

- Unter folgendem Link liegt ein Producer, welcher Personendaten (SteuerID, Name, Vorname) sendet:

  https://github.com/anderscore-gmbh/kafka-22.10/tree/main/Aufgabe6-Producer

- Implementieren Sie einen Consumer, welcher die Daten deserialisiert und auf der Konsole ausgibt.

Schemata:
http://broker-2.k.anderscore.com:8081/subjects
http://broker-2.k.anderscore.com:8081/schemas/ids/1
http://broker-2.k.anderscore.com:8081/schemas/ids/2

Aufgabe 6 - Hinweise

Hinweise:

- Verwenden Sie den Consumer aus Aufgabe 4 als Vorlage
- Die Schemata können Sie aus dem Producer übernehmen oder aus der Registry laden.
- Beachten Sie die pom.xml des Producers zu Abhängigkeit und Generator-Konfiguration
- Simulieren Sie einen Tippfehler: Benennen Sie das Feld firstName in fristName um und generieren Sie den Avro-Code für den Consumer erneut. Was passiert beim Start?
- Tipp: Ändern Sie die Consumer Group bei Bedarf
- Ein Beispiel für den Consumer finden Sie unter: https://github.com/anderscore-gmbh/kafka-22.10/tree/main/Aufgabe6-Producer

Kafka und Testing

anderScore trust in competence

Topics und Partitions

- Senden von Messages an einen Topic, Auswerten der recordMetadata
- Konsumieren eines Topics, Validieren der Messages
- Command Line, Kafka Client API (low-level)

Producer und Consumer

- Klassische Unit Tests mit Mocks
- Programmatische Integration Tests mit *EmbeddedKafkaCluster*
- Programmatische Integration Tests mit *Spring Boot*
- Deklarative Integration Tests mit *Zerocode*

Microservices

- End-to-end Tests mit *Testcontainers*

Kafka und Testing

anderScore trust in competence

Beispiel: Spring Boot

```
@SpringBootTest
@DirtiesContext
@EmbeddedKafka(partitions = 1, brokerProperties = { "listeners=PLAINTEXT://localhost:9092", "port=9092" })
class EmbeddedKafkaIntegrationTest {

    @Autowired
    private KafkaConsumer consumer;

    @Autowired
    private KafkaProducer producer;

    @Value("${test.topic}")
    private String topic;

    @Test
    public void givenEmbeddedKafkaBroker_whenSendingtoSimpleProducer_thenMessageReceived()
      throws Exception {
        producer.send(topic, "Sending with own simple KafkaProducer");
        consumer.getLatch().await(10000, TimeUnit.MILLISECONDS);

        assertThat(consumer.getLatch().getCount(), equalTo(0L));
        assertThat(consumer.getPayload(), containsString("embedded-test-topic"));
    }
}
```

Quelle: https://www.baeldung.com/spring-boot-kafka-testing

Beispiel: Zerocode

```
{
    "name": "produce_to_kafka",
    "url": "kafka-topic:people-address",
    "operation": "produce",
    "request": {
        "recordType" : "JSON",
        "records": [
            {
                "key": "id-lon-123",
                "value": {
                    "id": "id-lon-123",
                    "postCode": "UK-BA9"
                }
            }
        ]
    },
    "verify": {
        "status": "Ok",
        "recordMetadata" : "$NOT.NULL"
    }
}
```

Quelle: https://github.com/authorjapps/zerocode/wiki/Kafka-Testing-Introduction

Weiterführende Literatur zum Thema

- https://dzone.com/articles/a-quick-and-practical-example-of-kafka-testing
- https://github.com/authorjapps/zerocode/wiki/Kafka-Testing-Introduction
- https://medium.com/test-kafka-based-applications/https-medium-com-testing-kafka-based-applications-85d8951cec43
- https://github.com/apache/kafka/tree/trunk/streams/src/test/java/org/apache/kafka/streams/integration/utils
- https://www.baeldung.com/spring-boot-kafka-testing
- https://www.testcontainers.org/modules/kafka

Lektion x – Advanced Kafka Development

Specifying Offsets

- Consumer property `auto.offset.reset` legt fest, was bei bei einem ungültigem Offset in Kafka für die Consumer Group des Consumers passiert
  - Wenn eine Consumer Group das erste mal startet
  - Wenn das Consumer Offset kleiner als das kleinste Offset ist
  - Wenn das Consumer Offset größer als das letzte Offset ist

Kann gesetzt werden auf:

- `earliest`: Automatischer Reset des Offsets zu kleinstem verfügbarem Wert
- `latest` (default): Automatischer Reset des Offsets zu letztem verfügbarem Wert
- `none`: Excepton, wenn kein früheres Offset für die Consumer Group gefunden werden kann

Offsets ändern - Consumer

- KafkaConsumer API unterstützt Ansehen der Offsets und dynamisches Ändern des Nächsten zu Lesenden Offsets

- View Offsets:
  - `position(TopicPartition)`: Offset der nächsten Nachricht
  - `offsetsForTimes(Map<TopicPartition, Long> timestampsToSearch)`: sucht die Offsets für die angegebenen Partitionen nach Zeitstempel

- Change Offsets:
  - `seekToBeginning(Collection<TopicPartition>)`: Sucht erstes Offset von jeder der angegebenen Partitionen
  - `seekToBeginning(Collection<TopicPartition>)`: Sucht letztes Offset von jeder der angegebenen Partitionen
  - `seek(TopicPartition, offset)`: Sucht angegebenes Offset in angegebener Partition

Offsets ändern - Consumer

- **Beispiel:** Suchen bis zum Anfang aller Partitions, die von einem Consumer zu einer bestimmten Topic gelesen werden

```
consumer.subscribe(Arrays.asList(„MyTopic“));
consumer.poll(0);
consumer.seekToBeginning(consumer.assignment());
```

Offsets manuell comitten

- Default: `enable.auto.commit` ist auf `true`, alle 5 Sekunden, während des `poll()` Aufrufes

- **Problem:**
  - Szenario: Zwei Sekunden nach dem letzten Commit wird ein Rebalance ausgelöst, nach Rabalance starten Consumer bei der letzten Offset Position, die comitted wurde

  → **In diesem Fall: Offset ist zwei Sekunden und alle Nachrichten aus diesen zwei Sekunden werden zweimal verarbeitet! (At least once)**

Offsets manuell comitten

- `enable.auto.commit` auf `false` setzen

- `commitSync()`
  - Blockiert bis success, versucht so lange zu comitten, bis fataler Error auftritt
  - Für „at most once“: Aufruf von `commitSync()` direkt nach `poll()`, danach Nachrichten verarbeiten
  - Consumer sollte sicher gehen, dass alle Nachrichten die von poll() zurückgegeben wurden verarbeitet werden, sonst könnten Nachrichten verloren gehen

- `commitAsync()`
  - Sofortige Rückgabe
  - Nimmt optional einen Callback, der ausgelöst wird, wenn der Broker antwortet
  - Hat einen höheren Durchsatz, da der Consumer den nächsten Nachrichtenstapel verarbeiten kann, bevor der Commit zurückkehrt
  - **Nachteil:** Consumer kann später feststellen, dass die Übertragung fehlgeschlagen ist

Offsets manuell comitten

- **Note:** Das Offset, was comitted wird (unabhängig ob manuell oder automatisch) ist das Offset der als nächstes zu lesenden Nachricht!

- **Default:** Kafka speichert Offsets in einer speziellen Topic: `__consumer_offsets`
- Ggf. möchte man Offsets außerhalb von Kafka speichern (z.B. in Datenbank-Tabelle)
  - Wert lesen, dann `seek()` verwenden, um beim Starten der Anwendung an die richtige Position zu gelangen

Partitioning Data

Default Partitioning Schema:

- Kafka hashed den Message Key und benutzt das Ergebnis, um die Nachricht einer Partition zuzuordnen

→ Alle Messages mit gleichem Key gehen in die gleiche Partition

- Wenn der Key null ist, wird die Nachricht einer zufälligen Partition zugeordnet (round-robin Algorithmus)
- Dieses Verhalten kann überschreiben werden und ein eigenes Partitioning Schema eingeführt werden

Erstellen eines Custom Partitioners:

- Implementieren des `Partitioner` Interfaces
  - Enthält die `configure`, `close` und `partition` Methoden
  - `partition` enthält topic, key, serialized key, value, serialized value und cluster metadata
  - Gibt die Nummer der Partition zurück zu der die Nachricht gesendet wird

Beispiel: Custom Partitioner

Alle Nachrichten mit bestimmten Key in eine bestimmte Partition, alle anderen Nachrichten über die übrigen Partitions verteilen

Alternative zu einem Custom Partitioner

- Es ist möglich die Partition, in die eine Nachricht gehen soll, zu bestimmen, wenn der `ProducerRecord` erstellt wird:

```
ProducerRecord<String, String> record = new ProducerRecord<String,
String>("my_topic", 0, key, value);
```

→ Ordnet die Nachricht der Partition 0 zu

Diskussion: Welche Methode ist bevorzugt?

Aufgabe 7 – Error handling

- Implementieren Sie error handling für den Consumer in Aufgabe 3
  - Wenn eine Nachricht nicht verarbeitet werden kann, dann wird sie in die dead letter queue geschrieben
  - Der Consumer loggt jeden erfolgreich manuell comitteten offset
  - Der Consumer erhält über einen Kommandozeilenparameter den letzten erfolgreich bearbeiteten Offset und beginnt mit der nächsten Nachricht – verwalten Sie nur eine partition.

- Implementieren sie error handling für den Producer in Aufgabe 3
  - Der Producer loggt den offset einer erfolgreich übermittelten Nachricht. Synchronisieren Sie den commit