

KAFKA

ABLAUF

Bei fast allen Punkten Mischung aus Theorie und Praxis.

Übungsmaterial + Presentation

 https://github.com/e-geist/kafka_workshop_material

- Vorstellung und Lernziele
- Kafka Messages Einstieg
- Was ist Kafka und welche Probleme löst es?
- Kafka Komponenten (Broker, Topics, Partitionen)
- Kafka Messages: Keys und Values
- Message Offsets und Consumer Groups
- Kafka Clients (Producer/Consumer, Streams, KSQL, Connect)
- Log Compaction, Retention und Replication
- Keys und Values Best Practices
- Message Delivery Semantics (at-least-, exactly-, at-most-once)
- Kafka Security
- Weiterführende Themen

ÜBER MICH



- Software und Data Engineering Freelancer
- mehr als 6 Jahre Praxiserfahrung mit Kafka
- Kontakt
 - eugengeist.com
 - mail@eugengeist.com
 - <https://linkedin.com/in/eugen-geist/>

ÜBER EUCH

- Position (Admin, Entwickler, ...)
- Bisherige Berührungspunkte mit Kafka
- Erwartungen

LERNZIELE

Ihr wisst

- welche Probleme Kafka löst
 - wie ihr Kafka Nachrichten in verschiedenen Formaten empfängt und sendet
 - über die einzelnen Kafka-Komponenten und ihre Zusammenhänge
- Bescheid
- über Details, wie Message Delivery Semantics und Security, Bescheid
 - wo ihr Nachschauen könnt, falls Ihr Fragen zu Kafka habt

KAFKA MESSAGES EINSTIEG

- Kafka Cluster mit 1 Broker starten
- Kafka Messages Empfang und direkte Ausgabe auf shell
- Kafka Messages senden via shell

 Material 01

WAS IST KAFKA UND WELCHE PROBLEME LÖST ES?

*Apache Kafka is an open-source distributed **event streaming platform** used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.*

Kafka Website, 2025-06-30, 16:00

Events: grundlegende Datenstrukturen, die ein Ereignis in einem System oder einer Umgebung aufzeichnen

ibm.com, 2025-06-30, 16:05

Beispiele:

- User loggt sich in Website ein
- eine Temperatur wird von einem Sensor erfasst
- ein System schickt eine Anfrage an ein anderes System

Event Streaming ist die Praxis, Echtzeitdaten von [...] zu erfassen und sie zur sofortigen Verarbeitung und Speicherung oder für Analysen und Analyseberichte in Echtzeit an verschiedene Ziele zu übertragen.

ibm.com, 2025-06-30, 16:05

Kafka als Event-Streaming Plattform erlaubt

- den Import- und Export von Events (Event Streams) aus/zu anderen Systemen
- die zuverlässige Persistierung von Events, solange es notwendig ist
- die Verarbeitung von Events live oder retrospektiv

HARD FACTS

- Ende 2010 erstmals von LinkedIn entwickelt
- seit 2011 OpenSource
- seit Oktober 2012 Teil der Apache Software Foundation
- nach Franz Kafka benannt, da es ein "für das Schreiben optimierte System" ist
- ständig weiterentwickelt, aktuell Version 4.0.0
- quasi Industriestandard, siehe [powered by auf Website](#) für Verbreitung
- kommerzielles Angebot
 - Confluent, von ursprünglichen Kafka Entwicklern gegründet, bietet Cloud-Angebot, kommerzielle Features und Beratung
 - Managed Kafka in AWS und Google Cloud als eigene Services

Typische Anwendungsfälle

- Analyse und Speicherung von fortlaufenden Datenströmen von IoT Geräten bspw. Sensoren
- Speicherung und Live-Verarbeitung von Finanztransaktionen bspw. an Börsen
- Kommunikation zwischen Microservices
- Speicherung und Live-Verarbeitung von Kundenbestellungen eines Online-Shops

Warum Kafka und nicht relationale Datenbank?

- **Skalierung:** Datenströme von mehreren Millionen Events pro Sekunde stellen kein Problem dar
- **Event-basierte Verarbeitung:** relationale Datenbanken müssten gepollt werden
- **Asynchrone Kommunikation:** Nachrichtensender und -empfänger agieren unabhängig voneinander
- **Fehlertoleranz:** Datenreplikation stellt Verfügbarkeit von Daten sicher
- **Unveränderlichkeit (Immutability):** Geschriebene Events können nicht verändert werden

Sollte Kafka relationale Datenbanken ersetzen?

NEIN!

- nicht ausgelegt auf effizienten key/value lookup von Daten
- existierende Events können nicht verändert werden
- benötigt Erweiterungen wie ksqlDB zum querien von Daten, wenn nicht anhand eines Offsets
- komplexes Setup für schnelle Verarbeitung von Daten

Exkurs: Event Sourcing

Jede Veränderung des Zustandes einer Applikation/Entität wird in einem Event gespeichert.

Diese Events werden in der Reihenfolge der Veränderungen für die Lebenszeit des Systems gespeichert.

Der Zustand der Applikation wird durch das Auswerten der Events in der gespeicherten Reihenfolge ermittelt.

martinfowler.com, 2025-07-13, 20:05

In Szenario ohne Event Sourcing würde eine Datenbank für das Abspeichern des Zustandes ermittelt aus Events benutzt werden.

Ist Kafka geeignet für Event Sourcing? Ja und Nein.

- **Vorteile**
 - Speicherung von Events in Reihenfolge
 - Schnell + skaliert
 - keine Datenbank neben Events notwendig
- **Nachteile**
 - ineffizient beim Laden des Zustandes einzelner Entitäten bspw. anhand einer ID
 - Backups schwieriger als bei Datenbanken mit gängigen Backup-Mechanismen
 - Storage auf Dauer teurer als bspw. blob-storage
 - jede Applikation interessiert am Endzustand muss "Abspielen" von Events implementieren
- Für zeitlich begrenzte Szenarien (bspw. Entitäten, die nach einem Tag verschwinden) gut umsetzbar.

OFFIZIELLE DOKUMENTATION

- <https://kafka.apache.org/>
 -  Get Started
 -  Docs
- <https://docs.confluent.io/kafka>

KAFKA KOMPONENTEN (BROKER, TOPICS, PARTITIONEN)

Kafka wird auf mehreren Servern (einem Cluster) betrieben. Die Server zuständig für die Speicherung der Nachrichten werden Broker genannt.

kafka.apache.org, 2025-07-13, 20:10

*Clients ermöglichen das Schreiben (**Producer**) und Lesen(**Consumer**) von Events auf/von einem Kafka-Cluster.*

kafka.apache.org, 2025-07-13, 20:10

Events werden in Topics auf Kafka-Brokern organisiert und gespeichert.

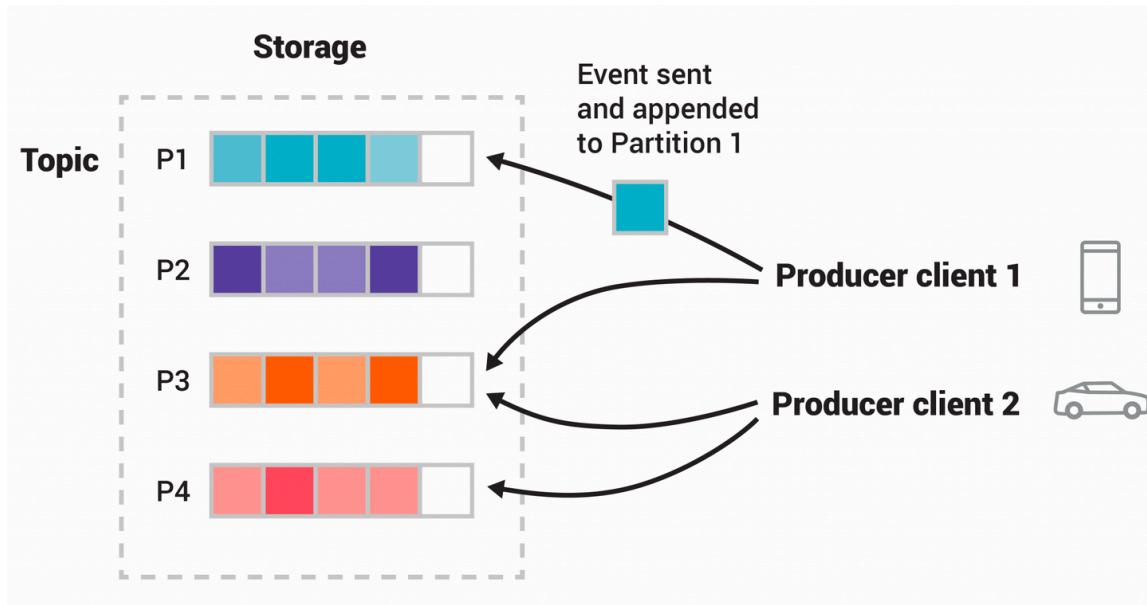
kafka.apache.org, 2025-07-13, 20:10

- wenn Events Dateien sind, sind Topics Dateiordner
- Topics können n verschiedene Consumer und m Producer haben
- Events in einem Topic können beliebig oft gelesen werden -> sie werden nicht nach einem Lesen gelöscht

*Topics werden in Teile unterteilt (**partitioniert**) und auf mehreren Brokern verteilt.*

kafka.apache.org, 2025-07-16, 19:10

- wenn ein neues Event in einem Topic veröffentlicht wird, wird es einer Partition (**hinten**) angehängt
 - die Reihenfolge der Events einer Partition
 - beim Lesen entsprechen der Reihenfolge beim Schreiben
 - ist nicht mehr veränderbar (Immutability)
- wenn Event Reihenfolge relevant → Zuordnung zu gleicher Partition notwendig



kafka.apache.org, 2025-07-16, 19:30

- mehrere Clients können auf derselben Partition schreiben und lesen
- Clients können auf verschiedene Partitionen zuschreiben
- Partitionen des selben Topics können auf verschiedenen Brokern liegen

PRAXIS

- Kafka Topics erstellen
- Kafka Topics löschen
- Kafka Topics anzeigen

 Material 02

KAFKA MESSAGES: KEYS UND VALUES

Kafka Nachrichten bestehen aus

- einem **Key** beliebigen Typs
- einem **Value** beliebigen Typs
- einem **Zeitstempel**, der entweder CreateTime oder LogAppendTime ist
- optionalen Metadaten (**Header**)

KAFKA MESSAGE KEY

- kann beliebigen Inhalt enthalten, da Byte-Array
- **wird meistens zur Zuordnung zu Partitionen benutzt**

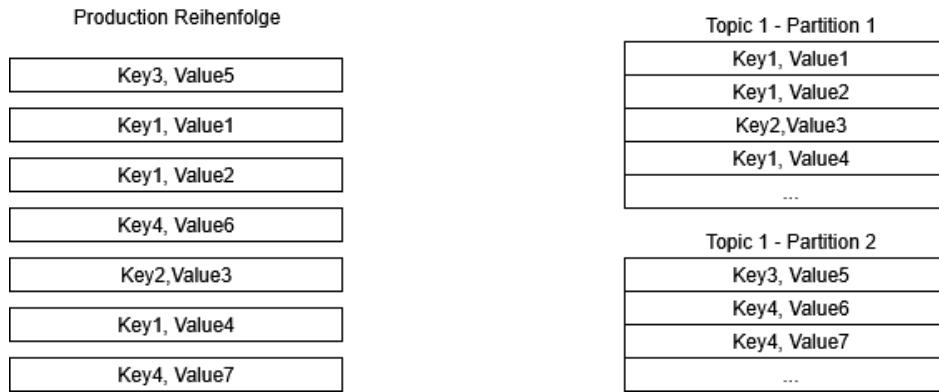
KAFKA KEY UND PARTITIONING

- auf welche Partition geschrieben wird, wird **clientseitig** im Producer entschieden
- Partition wird vorgegeben durch
 1. explizite Vorgabe während des Sendens der Nachricht
 2. DefaultPartitioner (Standardverhalten der meisten Producer): das Hashing des Keys
 3. CustomPartitioner: manuell geschriebener Code, der über Partitionzuordnung entscheidet
 4. Round-Robin: gleichmäßige Verteilung auf alle Partitionen unabhängig von Key und Value

DEFAULT PARTITIONER

- weist Partitions anhand von Key in der Message zu
- Key wird gehasht und anhand von modulo des Hashes einer der Partitionen zugeordnet
- solange Anzahl der Partitionen konstant:

Messages mit gleichem Key → gleiche Partition → Lese-Reihenfolge wie Schreib-Reihenfolge



Default Partitioner

- bei Änderung der Partitionen können neue Messages mit demselben Key auf anderen Partitionen landen
- Messages ohne Key werden Round-Robin, gleichmäßig auf alle Partitionen nacheinander, verteilt → **keine Garantie bzgl. Reihenfolge**
- Standardverhalten der meisten Producer, wenn nichts explizit anderes eingestellt wird

CUSTOM PARTITIONER

- manuelle Implementierung der Partitionszuweisung
- Zuweisung kann anhand von Key, Value oder komplett unabhängig implementiert werden
- Beispiel: Key hat Präfix "important" -> immer Zuweisung zu letzter Partition

BEISPIELE FÜR VORIMPLEMENTIERTE CUSTOMPARTITIONER

- abhängig von benutzter Kafka-Library
- **Round-Robin-Partitioner:** weist Nachrichten gleichmäßig nacheinander Partitionen zu (wie bei DefaultPartitioner ohne Key)
- **Uniform-Sticky-Partitioner:** weist Nachrichten solange einer Partition zu bis diese voll ist und wechselt dann auf die nächste

Production Reihenfolge

Key3, Value5
Key1, Value1
Key1, Value2
Key4, Value6
Key2, Value3
Key1, Value4
Key4, Value7

Topic 1 - Partition 1

Key3, Value5
Key1, Value2
Key2, Value3
Key4, Value7
...

Topic 1 - Partition 2

Key1, Value1
Key4, Value6
Key1, Value4
...

Round-Robin-Partitioner

Production Reihenfolge

Key3, Value5
Key1, Value2
Key2, Value3
Key4, Value7

Topic 1 - Partition 1

Key3, Value5
Key1, Value2
Key2, Value3
Key4, Value7

Topic 1 - Partition 2

Key1, Value1
Key4, Value6
Key1, Value4

Uniform-Sticky-Partitioner

BLEIBT SOLANGE BEIM DEFAULTPARTITIONER, BIS EINEN WIRKLICH GUTEN GRUND GIBT ZU WECHSELN

- von allen (größeren) Kafka Client Libraries verwendet
 - keine Kompatibilitätsprobleme
 - Producing aus verschiedenen Programmiersprachen in dasselbe Topic resultiert in gleichem Partitioning
- weniger Konfigurationsaufwand

KAFKA MESSAGE VALUE

- kann beliebigen Inhalt enthalten, da Byte-Array
- wird benutzt, um Daten zu transportieren
- Format sollte abhängig von Applikation sein
- gängige Formate:
 - JSON
 - Avro
 - Protobuf
 - MessagePack

PRAXIS

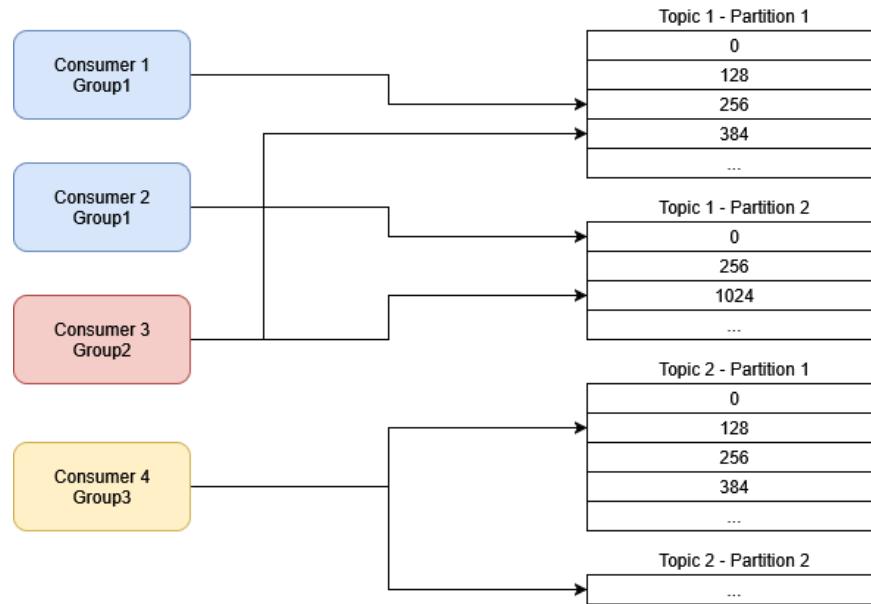
- Ausgabe von Keys und Partitionen beim Consumen
- Senden von Nachrichten ohne Key
- Senden von Nachrichten mit unterschiedlichen Keys



Material 03

MESSAGE OFFSETS UND CONSUMER GROUPS

- Kafka Messages werden per Topic und Partition mit eindeutigem Offset (64-bit Ganzzahl) identifiziert
- Kafka Consumer merken sich welche Nachrichten sie pro Topic und Partition verarbeitet haben anhand des Offsets
- um Messages mit mehreren Consumern gleichzeitig verarbeiten zu können, sowohl zur Skalierung, als auch mehrere Applikationen, wurden **Consumer Groups** eingeführt
 - Offsets werden für Topic und Partitionen pro Consumer Group separat getrackt
 - Kafka Consumer werden per Consumer Group Name einer Consumer Group zugewiesen
 - Offsets für Consumer Groups werden in eigenem Topic gespeichert, damit neue und wiederkehrende Consumer aktuellen Offset bekommen können



Consumer Groups Offsets in Partitionen

- bei mehreren Consumern
 - mit **verschiedenen** Consumer Groups, werden Messages aus allen Partitionen an alle Consumer Groups geliefert → Mehrfachverarbeitung möglich, bspw. für verschiedene Applikationen
 - mit **denselben** Consumer Groups, werden Messages aus verschiedenen Partitionen an verschiedene Consumer geliefert → Verteilung der Verarbeitung eines Topics auf mehrere Consumer möglich für Horizontalskalierung
- generell: eine Topic Partition wird maximal von einem Consumer per Consumer Group bearbeitet

- meisten Consumer starten *standardmäßig*
 - bei **existierender** Consumer Group mit Messages nach zuletzt verarbeitetem Offset
 - bei **nicht existierender** Consumer Group mit neu ankommenden Messages → bereits existierende Messages werden nicht verarbeitet
- Consumer Offset kann allerdings pro Topic Partition beliebig gesetzt werden
 - flexibel, wenn benötigt
 - mit explizitem Offset
 - an den Anfang

PRAXIS

- Ausgabe von Consumer Groups
- Mehrere Consumer mit unterschiedlichen Consumer Groups
- Mehrere Consumer mit gleicher Consumer Group
- Message Consumption nach Pause

 Material 04

- Offsets von abgerufenen Messages werden standardmäßig automatisch von Consumern an Broker zur Speicherung übermittelt (**committed**)
 - **Auto-Commit** kann in Consumern auch abgeschaltet werden, um zu kontrollieren wann eine Message als *verarbeitet* gilt
- Beispiele bei Clients und mehr Details zur Nutzung bei Message Delivery Semantics

KAFKA CLIENTS

PRODUCER/CONSUMER

STREAMS

KSQL

CONNECT

- **Producer/Consumer:** senden/empfangen von Batches von Message in Applikation via direkter Kommunikation mit Kafka Brokern
- **Streams:** Transformation von Datenströmen aus Input Topics nach Output Topics
- **ksqlDB:** SQL-Abstraktion zur Verarbeitung von Kafka Messages
- **Connect:** Integration von anderen Datensystemen mit Kafka via vordefinierten Konnektoren

PRODUCER/CONSUMER

- am häufigsten verwendete Client API
- in gängigsten Programmiersprachen verfügbar, manchmal verschiedene Implementierungen
- erlaubt volle Kontrolle über Serialisierung, Partitionierung und Verarbeitung
- **Producer:** sendet Nachrichten an Kafka Topics
- **Consumer:** liest Nachrichten aus Kafka Topics
- direkte Kommunikation mit Brokern bzw. Leader von Partitionen (dazu mehr bei Replikation) → dazu werden Metadaten von Brokern angefragt

PRODUCER

- sendet Nachrichten an spezifische Partition → Entscheidung über Partition findet im Producer statt
- integriertes Batching durch asynchrones Senden
 - Kafka Producer sammelt Nachrichten bevor sie an Broker gesendet werden
 - Nachrichten werden nach Erreichen bestimmter Anzahl oder Latenzgrenze gesendet
 - Anzahl von Nachrichten und Latenzgrenze konfigurierbar

CONSUMER

- **pull-basiert:** Client fragt neue Nachrichten regelmäßig bei Brokern ab
- falls Consumer Nachrichten langsamer verarbeitet, als sie ins Topic produziert werden, entstehen keine Probleme
 - nicht verarbeitete Nachrichten können nachgeholt werden
 - Consumer kontrolliert wie oft neue Nachrichten abgefragt werden
- um busy-waiting zu vermeiden, bieten Clients long-polling an → der Client wird erst benachrichtigt, wenn eine konfigurierbare Menge Bytes an neuen Nachrichten vorhanden ist

JAVA

- offizieller Client: kafka-clients von org.apache.kafka (aktuell Version 4.0.0)
 - [MVN Repository](#)
 - von Confluent mitentwickelt
 - [Offizielle Dokumentation](#)
- Spring Abstraktion für offiziellen Kafka Client
 - [MVN Repository](#)
 - nutzt kafka-clients, erlaubt allerdings Anwendung von Spring Prinzipien, wie Dependency Injection
 - [Offizielle Dokumentation](#)

JAVA CONSUMER

- Consumer-Klasse: KafkaConsumer
- Konfiguration des Consumers → Konstruktor erhält Instanz von Properties → Initialisierung via Code oder Datei
- KafkaConsumer.subscribe wird mit Collection von Topic-Namen aufgerufen, um Nachrichten zu empfangen
- KafkaConsumer.poll wird aufgerufen, um neue Nachrichten vom Broker zu empfangen
- Nachrichten werden als Container ConsumerRecords<K, V> zurückgegeben
→ Typ-Parameter abhängig von konfigurierten Deserializern
- Iteration über ConsumerRecords liefert ConsumerRecord<K, V> → Typ-Parameter abhängig von konfigurierten Deserializern
- Zugriff auf Message-Daten über ConsumerRecord-Instanzen
 - Key: .key()
 - Value: .value()
 - Zeitstempel: .timestamp()
- [JavaDoc](#)

JAVA PRODUCER

- Producer-Klasse: KafkaProducer
- Konfiguration des Producers → Konstruktor erhält Instanz von Properties
→ Initialisierung via Code oder Datei
- KafkaProducer.send wird mit ProducerRecord-Instanz aufgerufen, um Nachrichten zu versenden
- Producer sendet asynchron → nach erreichen einer konfigurierten Batch-Größe oder Latenz
 - KafkaProducer.flush kann aufgerufen werden, um aktuellen Batch an Messages sofort zu senden
- ProducerRecord erhält in Konstruktor Werte von Message, die gesendet werden soll
 - Topic
 - Value
 - Key (optional)
 - ...
- [JavaDoc](#)

PRAXIS

- Java Kafka Client Producer
- Java Kafka Client Consumer



Material 05 Part 1

JAVA CONSUMER OFFSETS

- Standardmäßig übermittelt Java Consumer automatisch abgerufene Offsets (**Auto-Commit**)
 - Kann ausgeschaltet werden über Einstellung `enable.auto.commit` bzw. `ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG`
 - Auto-Commit Interval kann über `auto.commit.interval.ms` bzw. `ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG` konfiguriert werden
 - über `KafkaConsumer.commitAsync` bzw. `KafkaConsumer.commitSync` können zuletzt abgerufene Offsets explizit committed werden
- Initiales Verhalten von Consumer Group kann über `auto.offset.reset` bzw. `ConsumerConfig.AUTO_OFFSET_RESET_CONFIG` gesteuert werden
 - `earliest`: neue Consumer Groups fangen bei erster existierender Message im Topic an
 - `latest`: neue Consumer Groups empfangen nur Messages, die nach ihrer Erstellung gesendet werden

- Offset kann manuell gesetzt werden
 - KafkaConsumer.seek setzt Offset explizit für eine Partition
 - KafkaConsumer.seekToBeginning setzt Offset auf erste verfügbare Message für Liste an Partitionen
 - KafkaConsumer.seekToEnd setzt Offset nach letzter verfügbarer Message für Liste an Partitionen

PRAXIS

- Java Consumer Offset
- Java Consumer Offset Commit



Material 05 Part 2

STREAMS

- Andere Art von Kafka Client: übernimmt Flow von Applikation
- Erlaubt Definition für Transformationen von Daten aus Kafka Topic und schreibt Ergebnis in anderes Kafka Topic
- abstrahiert Kafka Topics als Streams und Tabellen
- Definition von möglichen Transformationen als Domain Specific Language (DSL)
- API vor allem für Java vorhanden + kleinere Implementierungen in anderen Sprachen (bspw. Faust für Python)

STREAMS UND TABELLEN

- KStream abstrahiert Kafka Topic als Stream/Datenstrom von einzelnen Records
 - partitionierter Datenstrom -> bei mehreren Applikationsinstanzen bekommt jede Instanz nur einen Teil der Daten
- KTable abstrahiert Kafka Topic als Tabelle bzw. Changelog Stream einer Tabelle
 - Messages mit einem Value ($\neq \text{null}$) und dem selben Key werden als UPSERT (INSERT bzw. UPDATE) in der Tabelle gewertet
 - Messages ohne Value werden als DELETE gewertet
 - partitionierte Tabelle -> bei mehreren Applikationsinstanzen bekommt jede Instanz nur einen Teil der Daten
- GlobalKTable abstrahiert wie KTable, nur dass jede Applikationsinstanz alle Daten bekommt

TRANSFORMATIONEN

- Typische Stream + Tabellen Stateless- und Stateful-Transformationen möglich:
 - Map, Filter, Branch
 - GroupBy, Aggregate, Count
 - Reduce, Joins, Foreach, Peek
- Mögliche Transformationen abhängig von Art des Streams (KStream vs KTable)
- Verkettung von Transformationen möglich -> *Pipeline*
- Details über DSL

- `map` erlaubt Transformation von eingehendem Message-Key bzw. -Value
 - für Key,Value möglich auf KStream
 - für Value möglich auf KStream und KTable
 - bspw. `toUpperCase` auf String-Value, um alle Values in Großbuchstaben zu transformieren
- `filter` erlaubt das Filtern von Werten aus KStreams und KTables
 - leitet nur Messages weiter, deren Rückgabewert in übergebener Funktion `true` ergab
 - bspw. `value.startsWith("A")` leitet nur Messages weiter, deren Value mit einem A anfängt

- `foreach` erlaubt Operationen auf eingehenden Messages und terminiert den Stream/die Table danach
 - nach `foreach` sind keine weiteren Operationen auf dem KStream/der KTable möglich
 - nützlich um bspw. HTTP-Aufruf pro Message auszuführen oder Message Inhalte in Datei zu schreiben
 - falls weitere Operationen nach `foreach` notwendig, stattdessen `peek` nutzen

AUSGABE

- to auf KStream
 - Erfordert Angabe von Topic an das Ergebnis gesendet wird
 - Erfordert Angabe von Serialisierung für Key und Value
- KTable muss zunächst mit toStream in KStream umgewandelt werden

PRAXIS

- Java Kafka Streams Client
- Java Kafka Streams Filter
- Java Kafka Streams Mapping
- Java Kafka Streams Output

 Material 06

RETENTION

LOG COMPACTION

REPLICATION

Messages in Kafka Topics können auf zwei verschiedenen Wegen gelöscht werden

- **delete:** löscht Nachrichten nach Erreichen eines konfigurierten Zeit- oder Größenlimits
- **compact:** behält nur neuste Nachricht pro Key (*to compact -> verdichten*)
 - auch Kombination möglich
 - alte Segmente (Sammlung von Messages) werden nach Zeit-/Größenlimits gelöscht
 - beibehaltene Segmente werden compacted
- Konfigurierbar über **log.cleanup.policy** im Broker
 - allerdings auch einstellbar pro Topic bei Erstellung
 - Standardeinstellung: delete

EXKURS: SEGMENT

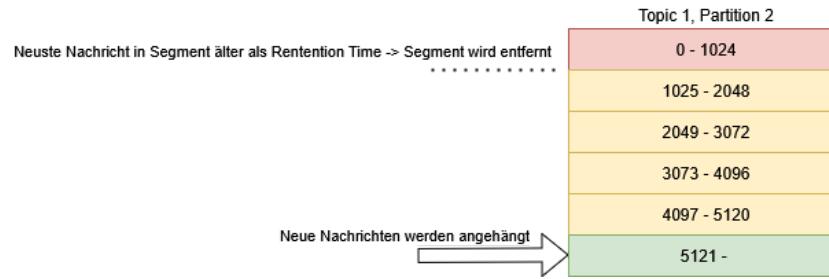
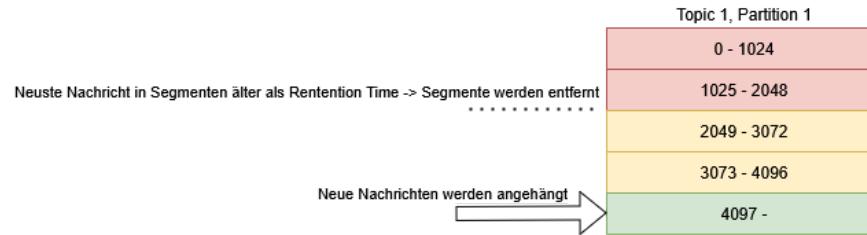
- Kafka Broker legen Messages als Dateien in Dateisystem des Brokers ab
- **Segment:** eine Datei auf der Festplatte eines Brokers, die eine Anzahl Messages eines Partition-Topics enthält
- neue Messages werden an neustes Segment angehängt
- sobald Segment konfigurierte Größe erreicht, wird ein neues Segment angelegt
- Größe eines Segmentes konfigurierbar über **segment.bytes**
- sobald ein Consumer Messages pollt, werden sie aus Segmenten gelesen und zurückgegeben

RETENTION

Es werden nur ganze Segmente, keine einzelnen Messages gelöscht

LÖSCHUNG NACH ABLAUF EINES ZEITLIMITS

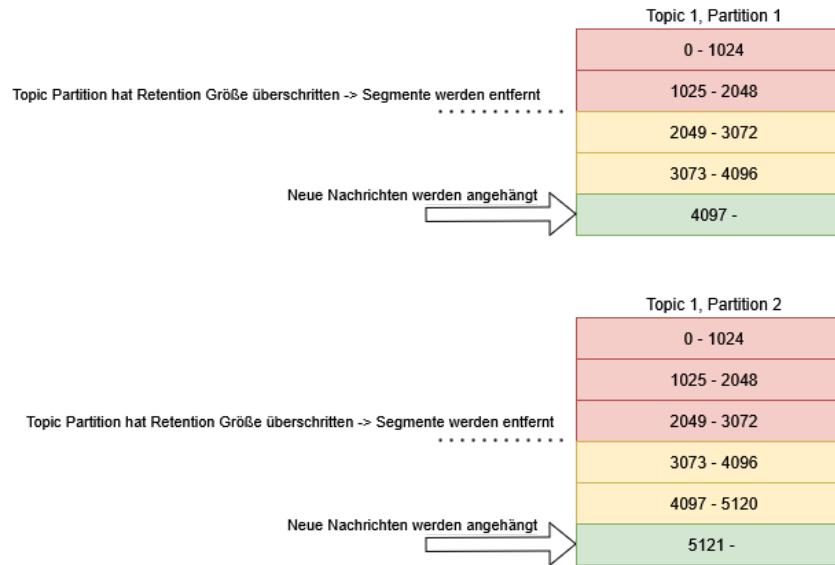
- konfigurierbar pro Topic
 - Standardwerte werden aus Broker-Konfiguration `log.retention.ms`, `log.retention.minutes` oder `log.retention.hours` bezogen
 - Überschreiben von Standardwerten bei Topic-Erstellung möglich
- Segment wird gelöscht, sobald Zeitstempel der **neusten Nachricht** das Zeitlimit erreicht hat



Retention Time Segment Löschung

LÖSCHUNG NACH ERREICHEN EINES GRÖSSENLIMITS

- konfigurierbar pro Topic
 - Standardwert wird aus Broker-Konfiguration `log.retention.bytes` bezogen
 - Überschreiben von Standardwert bei Topic-Erstellung möglich
- Segmente werden solange gelöscht bis Partition das konfigurierte Größenlimit unterschritten hat



Retention Size Segment Löschung

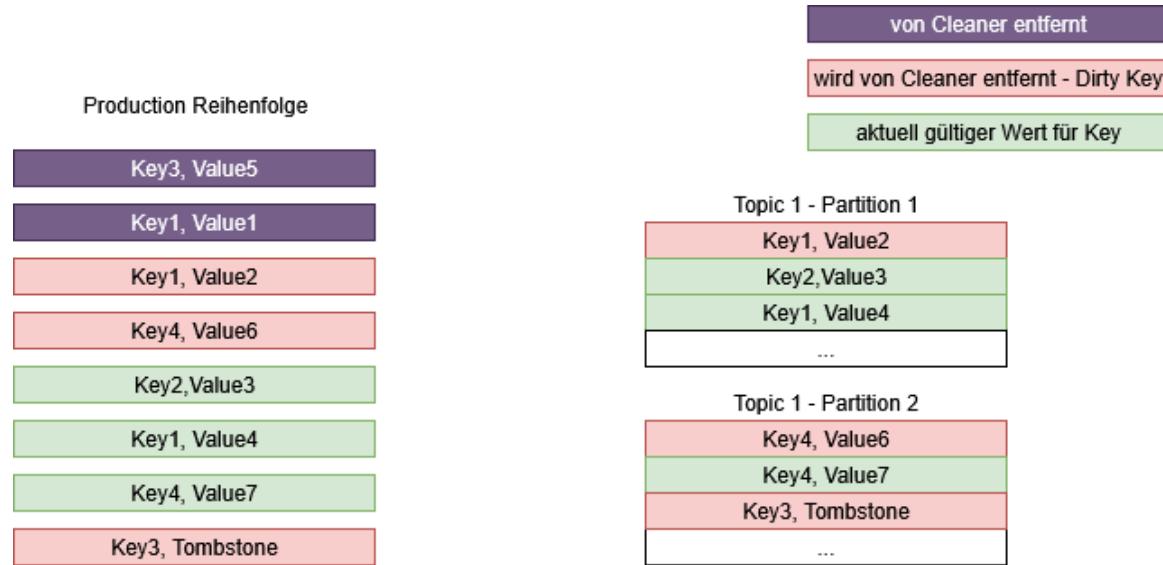
Zeit- und Größenbasierte Retention Policies können kombiniert werden

- Segmente werden gelöscht, sobald eines der Limits erreicht ist

LOG COMPACTION

Idee: Selektiv Einträge aus Kafka entfernen, für die eine aktuellere Version mit demselben Key existiert

- Messages mit *dirty* Keys werden aus Segmenten nach Erreichen einer Schwelle entfernt
 - neuere Messages mit gleichem Key müssen existieren
 - Cleaner läuft asynchron → Segmente werden niemals für Leseoperationen blockiert
- Keine Garantie, dass nurnoch ein Eintrag pro Key zu jeder Zeit, da Cleaner asynchron läuft → jedoch Garantie, dass immer mindestens neuster Eintrag pro Key erhalten bleibt
- explizite Löschung von Keys möglich durch Senden von Nachrichten mit leerem Value sogenannte *Tombstones*
 - Tombstones werden nach konfigurierbarer Zeit auch durch Cleaner automatisch entfernt
 - falls in der Zwischenzeit Consumer Tombstone liest, wird Message mit Key und leerem Value geliefert



Log Compaction Cleanup

Cleaner Verhalten für Compaction **cluster-weit** konfigurierbar

- **log.cleaner.delete.retention.ms:** Dauer nach der ein Tombstone entfernt wird
- **log.cleaner.min.compaction.lag.ms:** Minimale Zeit, die eine Nachricht erhalten bleibt, selbst wenn eine neuere Nachricht mit gleichem Key existiert
- **log.cleaner.max.compaction.lag.ms:** Maximale Zeit, die eine Nachricht erhalten bleibt, sobald eine neuere Nachricht mit gleichem Key existiert
- **log.cleaner.min.cleanable.ratio:** Minimaler Anteil des gesamten Logs, der Dirty ist, bevor etwas compacted wird

Für mehr Kontrolle können die **log.cleaner.min.*** und **log.cleaner.max.compaction.lag.ms** Einstellungen kombiniert werden

REPLICATION

- Kafka repliziert jede Topic-Partition auf eine konfigurierbare Anzahl von Brokern
- bei Ausfall eines Brokers kann automatisch auf die noch verfügbaren Broker umgeschaltet werden, sodass Messages verfügbar bleiben

- jede Topic-Partition in Kafka hat
 - einen **Leader**
 - 0..n **Follower**
- Leader + Anzahl Follower = Replikationsfaktor - konfigurierbar pro Topic
- Typischerweise mehr Partitionen als Broker → Partitionen gleichmäßig auf Broker verteilt
- Messages
 - werden an den Leader einer Partition produced
 - können vom Leader und allen Followern einer Partition consumed werden

- Follower consumen Messages vom Leader und hängen diese an ihre eigenen Topics
- Partitionen auf Followern
 - sind identisch zu Leader (Messages, Offsets, Reihenfolge)
 - können allerdings zu jedem gegebenen Zeitpunkt weniger Messages enthalten aufgrund von Replikationsverzögerung

EXKURS: CONTROLLER

- Spezieller Kafka-Server in Kafka-Clustern mit Replikation
 - für das Speichern von Metadaten der einzelnen Broker
 - für das Auswählen neuer Leader, falls aktiver Leader ausfällt
- bis Kafka 3.9 war Kafka Controller
 - Apache Zookeeper: eine separate Software ODER
 - Kafka Raft: Kafka Server in speziellem Modus
- ab Kafka 4.0 wurde Kompatibilität zu Apache Zookeeper entfernt

- Rolle eines Kafka Servers wird über Einstellung `process.roles` bestimmt
 - **broker**: Server agiert als Broker zur Message Speicherung und Verwaltung
 - **controller**: Server agiert als Controller zur Metadaten-Verwaltung und Leader-Election
 - **broker,controller**: Server agiert als Controller und Broker gleichzeitig → nicht empfohlen in kritischen Szenarien, da Austauschen einzelner Server schwieriger

Zwei Bedingungen für Broker, damit er als lebendig/nicht ausgefallen wirkt

1. Der Broker muss eine aktive Session mit dem Controller haben, um regelmäßig die neusten Metadaten zu bekommen → bei KRaft regelmäßige Heartbeats
2. Broker, die Follower sind, dürfen in der Replikation von Messages nicht zu weit zurück fallen

Server gelten als **in sync**, wenn sie diese Bedingungen erfüllen

IN SYNC REPLICAS

- Leader einer Partition verwaltet die Menge der in sync Broker, im Folgenden **in sync Replicas (ISR)** genannt
- Leader ist auch Teil der ISR
- wenn ein Follower die Consumer-Session verliert oder in der Replikation zu weit zurück fällt, entfernt der Leader den Follower aus der Liste der ISR

- eine Message gilt als **committed**, wenn alle ISR für die entsprechende Partition, sie ihrem Log hinzugefügt haben
 - Eine Message wird nur an Consumer herausgegeben, wenn
 - sie auf alle ISR repliziert wurde
 - die Anzahl der ISR mindestens der Einstellung **min.insync.replicas** (Standardwert: 1) entspricht
 - Producer-Clients können je nach Bedarf auf
 - eine Bestätigung des **committed** Status warten
 - eine bestimmte Anzahl an Bestätigungen der ISRs warten
 - keine Bestätigung warten
- bei Ausfällen einzelner Broker kann Kafka ohne Probleme weiterarbeiten und Messages gehen nicht verloren
- bei kompletten Netzwerk Partitions kann es zu Ausfällen kommen, bspw. wenn ein Broker komplett separiert wird

LEADER AUSFALL

- die Menge an ISR pro Partition wird auch in den Controller Servern persistiert
- beim Ausfall eines Brokers, wird aus der Menge der ISR für jede Partition des ausgefallenen Brokers, von den Controllern ein neuer Leader bestimmt
- bei $n+1$ Replicas, können n Broker ausfallen, ohne dass Kafka in Probleme gerät
- ausgefallene Broker können, nachdem sie wieder intakt gesetzt wurden, Replikation aufholen und Teil der ISR werden

- alle Replica einer Partition sterben → Kafka wartet bis ein Broker aus den ISR wieder verfügbar ist
 - Daten der betroffenen Partitionen in der Zwischenzeit nicht verfügbar
 - dadurch sichergestellt, dass keine Daten verloren gehen
- über **unclean.leader.election.enable** konfigurierbar
 - falls eingeschaltet: erste wieder verfügbare Replica, selbst wenn nicht in ISR, wird Leader
 - da Replica ggf. nicht in sync → Datenverlust möglich, da Messages fehlen

→ Tradeoff

falls Verfügbarkeit wichtiger und Datenverlust tolerierbar
unclean.leader.election.enable aktivieren

PRAXIS

- Replication
- ISRs
- Ausfall von Brokern

 Material 07

MESSAGE KEYS UND VALUES

BEST PRACTICES

MESSAGE KEYS

Keys sollten **simpel** und **unveränderlich** sein, da sich bei Änderungen das Hashing und die Partition ändert.

- Dos
 - Strings/Ganzzahlen
 - Konstante Ids für Entitäten bspw. Customer_Id, Hardware_Id, Standort + Hardware_Id (bei Hardware_Ids pro Standort)
 - Messages, die aufeinander aufbauen, sollten gleichen Key haben
- Don'ts
 - JSON, Protobuf -> Formatierung von JSON oder Encoding kann Partitionierung brechen
 - Sich immer ändernde Ids, die nichts mit der Entität zu tun haben und Reihenfolge brechen

Null Keys sollten nur **mit Bedacht** eingesetzt werden, da dann Log Compaction anhand vom Key nicht funktioniert und Reihenfolge zufällig.

- Dos
 - Null Keys für komplett nicht zusammenhängende Events -> gleichmäßige Verteilung von Messages
- Don'ts
 - Null Keys für Messages, die aufeinander aufbauen
 - Null Keys für Messages, die immer vom gleichen Consumer verarbeitet werden sollten (→ gleiche Partition)

Keys sollten eine **hohe Kardinalität besitzen** und **Verteilung** der Messages pro Key sollte möglichst **gleich** sein.

- Dos
 - Zusammensetzen von verschiedenen Ids, um
 - höhere Kardinalität zu erreichen, bspw. Standort_Id+Hardware_Id
 - bessere Verteilung von Messages pro Key zu erreichen
 - CustomPartitioner implementieren, um bessere Verteilzung zu etablieren
- Don'ts
 - Viele verschiedene Keys, jedoch haben einige wenige viele Messages
 - Einige wenige Keys → kann im Nachgang zu Großteil der Messages auf einem Key führen

Falls Daten des Key relevant für Message Nutzung → auch in Value integrieren.

→ Key und Value werden voneinander entkoppelt und Key hat nur technischen Nutzen

Ansonsten: bei ggf. notwendigen Key Änderungen auch Änderungen auf Consumer Seite notwendig

MESSAGE VALUES

Values sollten strukturierte Daten in einem Format sein, dass Abwärts- und Aufwärtskompatibilität unterstützt.

- Dos
 - Standardisierte Serialisierungsoformate wie Avro, msgpack, Protobuf oder JSON
 - Nutzung einer Schema Registry evaluieren
- Don'ts
 - nicht standardisiert Formate wie CSV, TSV → bspw. Escape-Zeichen und wann " benutzt werden problematisch
 - Formate mit fixierter Breite → problematisch hinsichtlich Kompatibilität

Größe von Events beachten

- Dos
 - viele kleine Events
 - kleines Event zum Triggern + Verweis auf Speicher wo Payload gefunden werden kann
 - Kompression von Messages aktivieren
- Don'ts
 - Transport von großen binären Daten, wie Bildern

EXKURS: SCHEMA REGISTRY

- separater Service, der für die Verwaltung der Schemata/Daten-Kontrakte von Kafka Messages zuständig ist
- Zuständig für
 - Verwaltung von Daten Schemata
 - Bereitstellung von Daten Schemata
 - Kompatibilitätsregeln: kann Ab- und Aufwärtskompatibilität forcieren (konfigurierbar)
- von vielen Kafka Clients unterstützt

- Unterstützt Protobuf, JSON Schema und Avro
- Funktionsweise
 1. Producer generiert und registriert Schema
 2. Producer nutzt Schema für Message-Serialisierung und fügt Schema ID zur Message hinzu
 3. Consumer lädt anhand von ID Schema herunter und deserialisiert Message

VORTEILE

- Starke Datenkontrakte, die beim (De-)Serialisieren forciert werden → kein Senden/Empfangen von invaliden Nachrichten
- Zentrale Verwaltung von Datenschemata, die Reviews und Audits ermöglicht
- Zentrale Integration in Java Client Libraries

NACHTEILE

- weiterer Service, der betreut getestet und verwaltet werden muss
- falls Schema Registry nicht verfügbar, können keine Kafka Messages produced/consumed werden
- höhere Latenz, da Datenschemata über Netzwerk geholt werden müssen
- geregelte, aber komplexere und damit längere Prozesse zur Änderung bzw. Entwicklung von Daten-Schemata



SCHEMA REGISTRY NUTZEN WENN

- Consumer in anderen Teams oder öffentlich → Zentrale Bereitstellung von Schemata ohne Zusatzkommunikation
- (gesetzlich) regulierte Umgebung, die Nachverfolgbarkeit von Daten-Schemata erfordert

EXKURS: KOMPRESSION

Warum?

- Messages werden kleiner: können schneller übermittelt werden + verbrauchen weniger Speicher auf Broker

Kompression

- Funktionsweise
 - Producer komprimiert Message Batches pro Partition und sendet diese an Partition Leader
 - je nach Einstellung des Topics: Broker behält Kompression bei oder wandelt um
- Einstellungen
 - Producer: über **compression.type** einstellbar (Standard: keine)
 - Topic: über **compression.type** einstellbar (Standard: **producer** -> Kompression wird beibehalten)
- Kompatibilität beachten: ältere Clients unterstützen ggf. gewählte Kompressionsart nicht

→ **compression.type** von Topic sollte Kompression von Producer beibehalten, da sonst Umwandlung im Broker stattfindet. Nur in spezifischen Fällen bspw. Bottlenecks oder Kompatibilitätsproblemen von Producern/Consumern empfehlenswert.

MESSAGE DELIVERY SEMANTICS

Wie oft wird eine Message produziert und/oder gelesen?

- **at-most-once**: Messages werden **einmalig** ausgeliefert - bei System-Problemen sind sie verloren und es findet keine Wiederauslieferung statt
- **at-least-once**: Messages werden **1-n Mal** ausgeliefert - bei System-Problemen findet eine Wiederauslieferung statt, jedoch kann die gleiche Message mehrfach auftauchen
- **exactly-once**: Messages werden genau einmal ausgeliefert - bei System-Problemen gehen sie nicht verloren und werden nicht mehrfach ausgeliefert

PRODUCER

- **at-most-once:** Producer können konfiguriert werden, sodass sie auf keine Bestätigung der Broker warten
 - bei verlorenen Messages finden keine Retries statt
 - niedrigere Latenz
- **at-least-once:** Producer wartet auf Bestätigung vom Server, sendet Messages jedoch erneut, falls keine Bestätigung kommt
 - keine Messages gehen verloren
 - Latenz höher
 - Kafka-Default
- **exactly-once:** Producer können konfiguriert werden, sodass
 - bereits gesendete Messages bei Retries dedupliziert werden → **enable.idempotence** einschalten
 - Latenz höher
 - Topic-Übergreifend transaktionell produced wird → entweder komplette Transaktion wird produziert oder keine Nachricht

CONSUMER

- **at-most-once:** Consumer liest Messages aus Topic, committed Offset und verarbeitet Messages nach Commit
 - keine doppelte Verarbeitung
 - Messages gehen verloren, falls Consumer nach Commit vor Verarbeitung Probleme hat
- **at-least-once:** Consumer liest Messages aus Topic, verarbeitet sie und committed Offset
 - keine Messages gehen verloren
 - Messages könnten mehrfach verarbeitet werden, falls Consumer nach Verarbeitung vor Commit Probleme hat
- **exactly-once:** Consumer verarbeitet jede Message nur einmalig
 - mit Kafka Streams und Transaktionen möglich
 - bei Verarbeitung in externe Systeme schwierig zu realisieren

- Workarounds für *exactly-once* Consumer
 - *at-least-once* mit Idempotenten Messages: mehrfache Verarbeitung führt zu gleichem Endergebnis
 - *at-least-once* mit sequenziellen Ids, die als two-staged-commit benutzt werden
- in den meisten Fällen Kombination aus *at-least-once* Consumer und *at-least-once* Producer praktisch sehr gut nutzbar → bspw. bei INSERTs in relationale Datenbanken durch Deduplikation von Messages mit demselbem Primary Key

KAFKA SECURITY

Kafka erlaubt

- Authentifizierung via SSL(-Keyfiles) oder Simple Authentication and Security Layer
- Verschlüsselung von Daten während Übertragung → führt zu Verschlechterung der Performance abhängig von CPU und JVM
- Authorisierung von Operationen auf Ressourcen bzw. Clients dürfen nach Topic produzieren/von Topic konsumieren

Security ist in verschiedenen Ausbaustufen konfigurierbar, Kafka funktioniert allerdings auch ohne

Disclaimer: Kafka Security kann sehr komplex werden, daher hier nur
Ausschnitte

AUTHENTIFIZIERUNG UND VERSCHLÜSSELUNG

- Authentifizierungs- und Verschlüsselungsmethode wird pro Listener-Endpunkt auf Kafka-Broker definiert
- pro Kafka-Broker sind mehrere verschiedene Listener-Endpunkte möglich
- für Kommunikation zwischen Brokern und Broker und Client können verschiedene Protokolle definiert werden
- Beispiel
 - Authentifizierung + Kommunikation zwischen Brokern via SSL
 - Authentifizierung zwischen Brokern und Clients via SASL

- Konfiguration von Kafka Client abhängig von Endpunkt, der benutzt werden soll
- Property **security.protocol** entscheidet über Authentifizierungs-Verschlüsselungsmethode bei Verbindung von Client
 - muss selben Wert wie verwendeter Listener besitzen
 - mögliche Werte
 - **PLAINTEXT**: keine Authentifizierung und Verschlüsselung
 - **SSL**: SSL Authentifizierung und Verschlüsselung
 - **SASL_SSL**: SASL Authentifizierung, SSL Verschlüsselung
 - **SASL_PLAINTEXT**: SASL Authentifizierung, keine Verschlüsselung
- weitere Einstellungen abhängig von expliziter Verschlüsselungs- und Authentifizierungsmethode

Verfügbare Authentifizierungsmethoden via SASL

- GSSAPI/Kerberos: Authentifizierung via Kerberos Protokoll bspw. Active Directory
- PLAIN: Simple Username + Passwort Authentifizierung
- SCRAM-SHA-256/SCRAM-SHA-512: Sicherere Username + Passwort Authentifizierung
- OAUTHBEARER: Authentifizierung via OAuth2 Web-Token

Beispiel für **SASL_SSL** - Authentifizierung via SCRAM-SHA-512 +
Verschlüsselung von Datenstrom

Broker-Konfiguration-Auszug

```
# server.properties

# 1) Expose a SASL_SSL listener
listeners=EXTERNAL://0.0.0.0:9093
advertised.listeners=EXTERNAL://broker1.example.com:9093
listener.security.protocol.map=EXTERNAL:SASL_SSL
inter.broker.listener.name=EXTERNAL

# 2) Enable SCRAM-SHA-512 on that listener
sasl.enabled.mechanisms=SCRAM-SHA-512
# (optional, per-listener override)
# listener.name.external.sasl.enabled.mechanisms=SCRAM-SHA-512
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512

# 3) TLS cert for the broker (required for SASL_SSL)
ssl.keystore.type=PKCS12          # or JKS
ssl.keystore.location=/etc/kafka/ssl/broker.p12
ssl.keystore.password=changeit
ssl.key.password=changeit
```

Client-Konfiguration-Auszug

```
# --- security (SASL over TLS) ---
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-512
sasl.jaas.config=org.apache.kafka.common.security.scram.ScramLoginModule required \
    username="alice" \
    password="superSecretPassword";

# --- TLS trust (so the client trusts the broker's cert) ---
# If your broker uses a private/enterprise CA, point to your truststore:
ssl.truststore.location=/path/to/client-truststore.p12
ssl.truststore.password=changeit
ssl.truststore.type=PKCS12  # or JKS if you use JKS

# Hostname verification (HTTPS is the secure default; leave it)
ssl.endpoint.identification.algorithm=HTTPS
```

- Username und Passwort müssen auf Broker erstellt werden
- Truststore muss vom Broker benutztes Zertifikat enthalten → sichergestellt, dass Zertifikat zur Verschlüsselung vertrauenswürdig ist

AUTHORISIERUNG

- Kafka ACLs (Access Control Lists) erlauben Definition / Verbot von Operationen für User auf Kafka Ressourcen
- kafka-acls.sh erlaubt Verwaltung von Kafka ACLs
- Authorisierung muss für Broker + KRaft Controller definiert werden:

```
authorizer.class.name= \
org.apache.kafka.metadata.authorizer.StandardAuthorizer
```

- falls kein Authorizer definiert ist, darf jeder User alles
- falls Authorizer definiert ist und keine User-ACLs für eine Ressource existieren, dürfen nur Superuser Ressource verwenden
 - falls alle User Ressource verwenden können sollen bei nichtexistenten ACLs, in Broker-Konfiguration hinzufügen:

```
allow.everyone.if.no.acl.found=true
```

- Superuser können in Broker-Konfiguration definiert werden:

```
super.users=User:Bob;User:Alice
```

Beispiele für ACL-Definition

- Erlaube User Bob und Alice Consume + Produce von Topic *Test-Topic* von IPs **198.51.100.0** und **198.51.100.1**

```
./kafka-acls.sh --bootstrap-server localhost:9092 --add \
--allow-principal User:Bob --allow-principal User:Alice \
--allow-host 198.51.100.0 --allow-host 198.51.100.1 \
--operation Read --operation Write --topic Test-topic
```

- Erlaube allen Consume + Produce von Topic *Test-Topic* , außer User *BadBob* von IP **198.51.100.3**

```
./kafka-acls.sh --bootstrap-server localhost:9092 --add \
--allow-principal User:'*' --allow-host '*' \
--deny-principal User:BadBob --deny-host 198.51.100.3 \
--operation Read --topic Test-topic
```

WEITERFÜHRENDE THEMEN

QUOTAS

- Erlaubt Definition von Zugriffs-Kontingenten, um Überlastung von Ressourcen durch einzelne Clients zu vermeiden
- Definition von
 - erlaubter Netzwerk-Bandbreite pro Client(-Gruppe)
 - erlaubten Operationen pro Client(-Gruppe) anhand von Netzwerk und I/O CPU-Thread Nutzung
- Quota Design
- Quota Konfiguration

KAFKA DEPLOYMENTS ÜBER MEHRERE STANDORTE + REPLIKATION

- manche Applikationen sind über mehrere Standorte verteilt
 - Empfehlung
 - in jedem Standort lokales Kafka-Cluster
 - bei Bedarf: Replikation von Daten zwischen Clustern in verschiedenen Standorten
-  bei Netzwerk-Problemen zwischen Standorten funktionieren
Applikationen weiterhin

- Replikation von Daten über verschiedene Cluster möglich
- Beispielhafte Anwendungszwecke
 - Disaster Recovery
 - Sammlung von Daten in Standort-Clustern, Aggregation in zentralem Cluster
 - Cloud Migration
 - Compliance-Anforderungen
 - Isolation von Clustern (Produktiv <-> Entwicklungsumgebung)
- Tool zur Replikation zwischen verschiedenen Clustern: **MirrorMaker**
- **Inter-Cluster-Replikation + MirrorMaker Konfiguration**

KAFKA CONNECT

- erlaubt Streaming von Daten zwischen Kafka und anderen Systemen über vordefinierte Konnektoren
 - Kafka → S3
 - Postgres → Kafka
 - ...
- benötigt erweiterte Konfiguration von Kafka Cluster
- [Kafka Connect Dokumentation](#)
- [Übersicht verfügbarer Konnektoren](#)

KSQLDB

- SQL-Abstraktion für Kafka Messages
- von Confluent für kommerzielle Confluent Kafka Plattform entwickelt
- [ksqlDB Übersicht](#)

INTERNES FORMAT VON RECORDS UND LOG

- Implementierungs-Details, die helfen Kafka zu verstehen
- [Records | Messages](#)
- [Kafka Log | Format in Dateisystem](#)

UND VIELES MEHR

- Multi-Tenancy
- Monitoring
- Transaktionen
- Java, Hardware & OS



Offizielle Kafka Dokumentation inkl. weiterführender Links

VIELEN DANK!

OPTIMIERUNG ODER HILFE BEIM EINSTIEG IN KAFKA?

KONTAKTIERT MICH

- eugengeist.com
- mail@eugengeist.com
- <https://linkedin.com/in/eugen-geist/>