Introduction to Kafka and Event-Driven

Inter-process communication- Previous Slides Recap

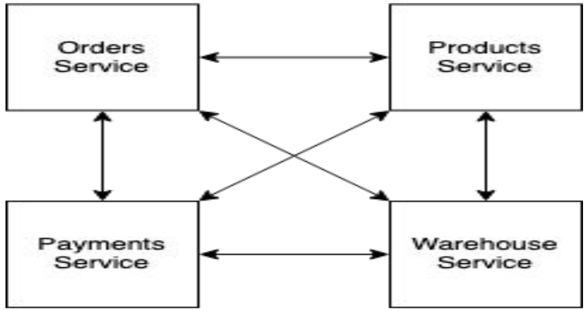
	one-to-one	one-to-many
Synchronous	Request / Response	-
Asynchronous	Asynchronous Request / Response One way notification	Publish / Subscribe Publish / Async responses

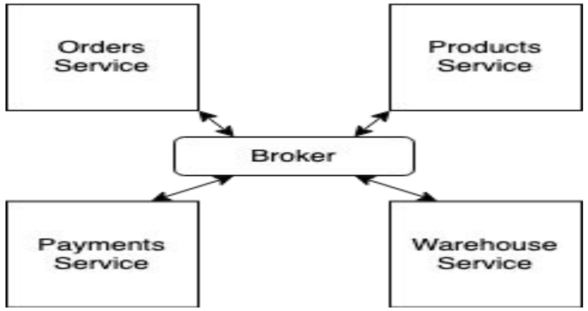
Asynchronous Messaging

Two **types** of async messaging:

- Brokerless
- Broker-based (**Kafka**, RabbitMQ, AWS kinesis...)

The message **represents** a **change** of the **state**, and is triggered when a user or service performs an action.

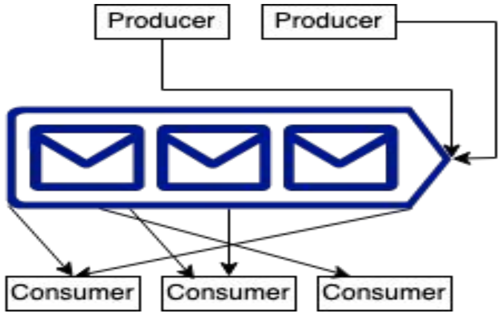


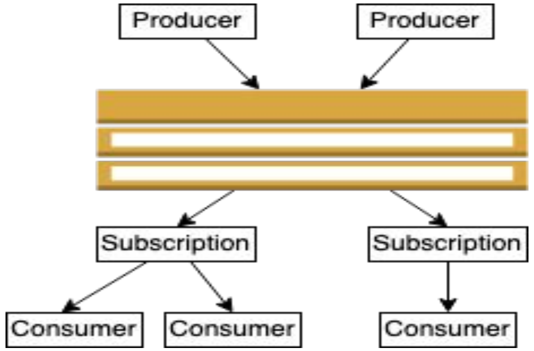


Brokers

- **Point to Point (1-to-1)**: message is sent from one producer to a consumer via a **queue (e.g. AWS SQS)**. Multiple consumers can listen on the queue but **only one** of them will get the message.

- **Publish/Subscribe (1-to-many)**: message is sent from one producer to multiple consumers/subscribers via a **topic (e.g. Kafka, AWS Kinesis)**. The subscribers may or may not acknowledge the published message





What is Kafka?

- An **event streaming platform** implemented as "**distributed commit log**".
- Uses an append-only **commit log** which provides a durable and replayable record.
- Designed to provide protection **against** failures and **scaling performance**.
- Emphasis on **high throughput** stream processing.
- Is a **database inside out**, a tool for storing data, processing it in real time, and creating views, optimized for continual computation rather than batch processing.
- Real-time processing first, long-term storage second.

Kafka Main Concepts

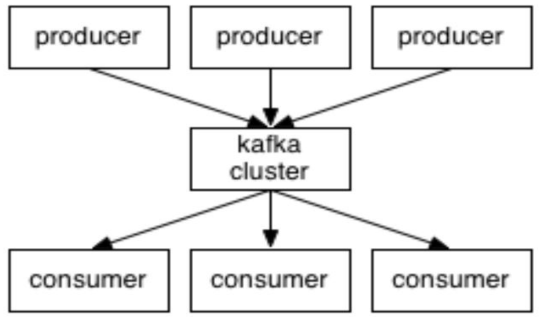
- **Events** have a **key** (optional), **value**, timestamp and metadata headers (optional).
- Events are **written** to **topics** for a configurable period of time called **retention period**.
- **Topics** are always multi-producer and multi-consumer and can be read as often as needed.
- Topics are **partitioned** and spread over a number of **partitions** (just a **bucket** located on different Kafka brokers).
- Events with the **same key** are **always** written to same the partition.
- For efficiency, messages are written into **batches** (collection of messages produced to same topic and partition)
- Kafka guarantees event **time-ordering** inside a **partition**.

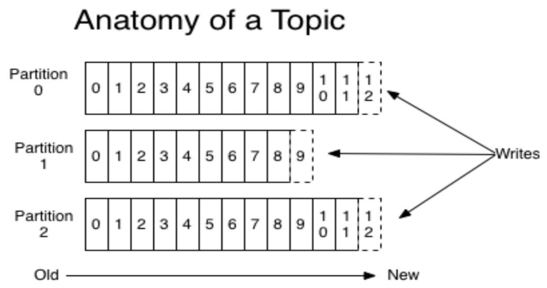
Kafka Main Concepts 2

- **Partitions** enable high throughput and scalability.
- Kafka enforces durability by **replicating partitions**.
- **Leader replica** where all producer / consumer requests go through.
- **Follower replicas** which replicate messages from the leader and aim to stay in-sync.
- Events written to the partition leader are not immediately readable by consumers.
- Only when all in-sync replicas have acknowledged the write the message is considered committed.
- This ensures that messages cannot be lost by a broker failure.
- Partition replication provides data fault tolerance and high-availability.

Kafka How It works - High Level

- Producers **send messages** to a Kafka broker.
- Messages are stored in a **topic**.
- Consumers **read** messages by subscribing to topics.
- Topics are splitted into **partitions** and are append-only.
- All messages inside a partition are **ordered** in the sequence they came in.
- Kafka follows the principle of a **dumb broker** and smart consumer, as doesn't keep track of what records are read by the consumer.
- Consumers use **offset** to keep track of the latest consumed message.

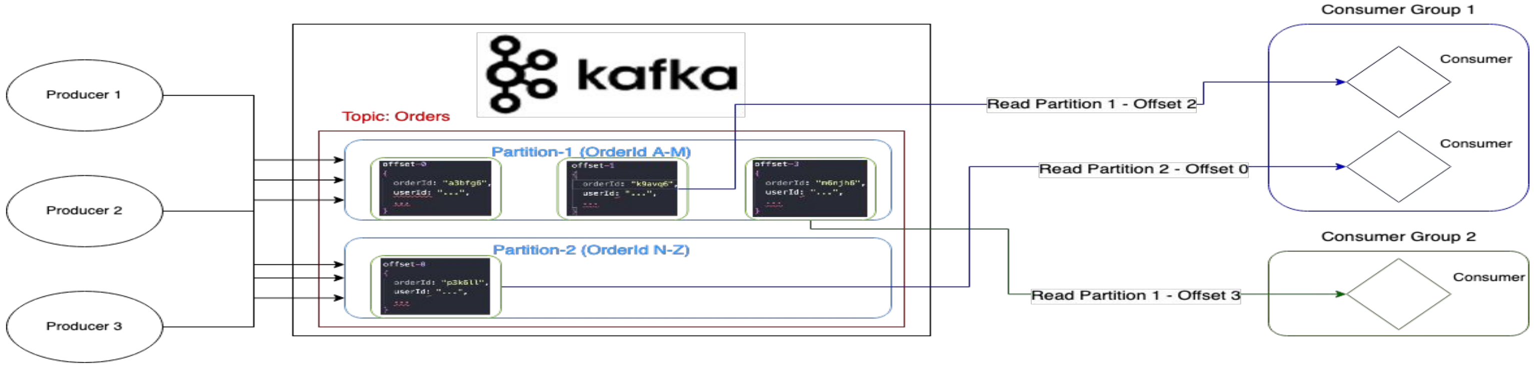




Kafka Broker Concepts (1)

- **Broker** is a single Kafka *server* allocated in a cluster (can contain multiple).

- The **goal** of broker is to:
  - receive messages from producers.
  - assign offsets to messages.
  - store the messages to disk.
  - respond to consumers requests for messages in specific partition.


Producer 1
Producer 2
Producer 3
kafka
Topic: Orders
Partition-1 (OrderId A-M)
Partition-2 (OrderId N-Z)
Read Partition 1 - Offset 2
Read Partition 2 - Offset 0
Read Partition 1 - Offset 3
Consumer Group 1
Consumer
Consumer
Consumer Group 2
Consumer

```
offset-0
{
  orderId: "a3bfg6",
  userId: "...",
  ...
}
```

```
offset-1
{
  orderId: "k9avq6",
  userId: "...",
  ...
}
```

```
offset-3
{
  orderId: "m6njh6",
  userId: "...",
  ...
}
```

```
offset-0
{
  orderId: "p3k6ll",
  userId: "...",
  ...
}
```

Kafka Broker Concepts (2)

- One broker is elected as **controller** and is responsible for
  - admin operations
  - assign partitions
  - electing partition leader
  - monitor for broker failures

- When controller notices that a **broker left** cluster:
  - goes over all partitions who need new leader
  - finds new leader
  - sends request to all broker with the list of new leaders and followers replicas.

Kafka Broker Configuration

Let’s explore some **core broker configurations**:

1. **broker.id** -> Every broker must have an unique integer identifier in a single cluster.
2. **port** -> Listener on TCP port default value 9092.
3. **logs.dirs** -> Directories used to persist data on disk.
4. **auto.create.topics.enable** -> Specifies if the broker can automatically create topics.

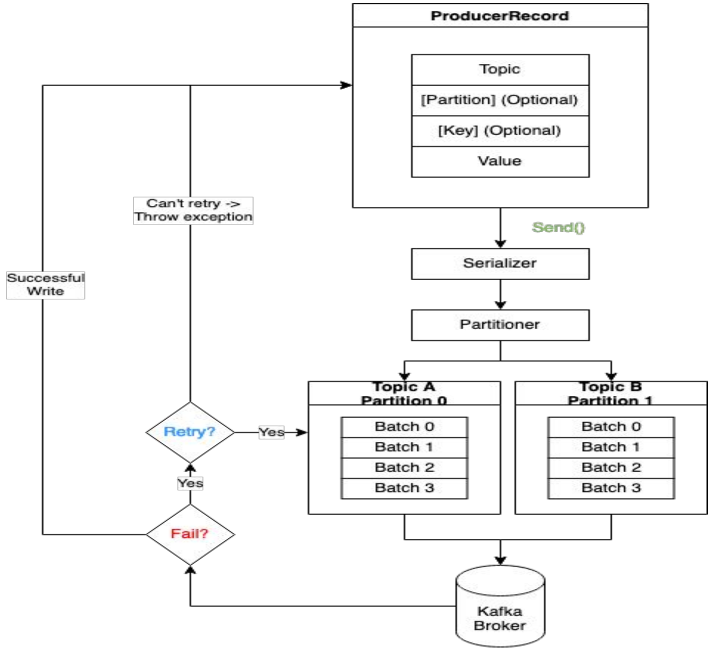
Kafka Topic Configuration

Let’s explore some **core topic configurations**.

1. **num.partitions** -> Defines how many partitions per topic. Number of partitions can only be increased, never decreased.
2. **log.retention.ms** -> Defines how long Kafka will keep messages. Default value 1 week.
3. **log.retention.bytes** -> Defines the max number of bytes of messages per partition, before they are considered as expired.
4. **message.max.bytes** -> Defines the max size of a message. Default value is 1MB.

Kafka Producer - Concepts (1)

- Emits messages **(ProducerRecord)** to Kafka.
- ProducerRecord is delivered to **Partitioner**, which determines the message partition.
- If the key is not specified, Partitioner chooses **round-robin** algorithm to determine the partition, otherwise uses **key hash** to determine the partition.
- **Partitioner** adds ProducerRecord to a **batch,** which will be delivered to **same topic** and **partition**.
- Broker receives records and responses back with **success** or **failure**.
- In case of **failure**, if **retry** is possible, re-sends records otherwise throws an exception.



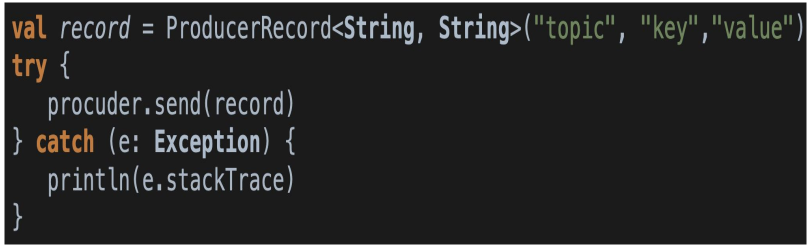
Kafka Producer - Concepts(2)

There are 3 ways to **send message** to Kafka via Producer API:

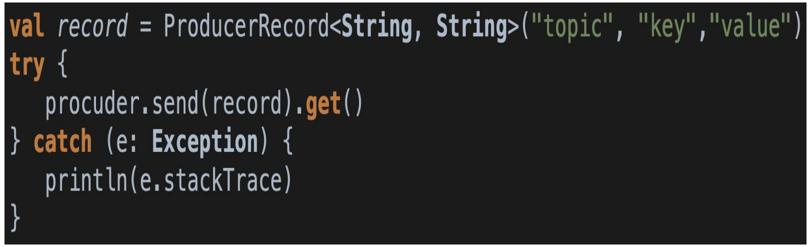
1. **Fire & Forget**: Using *send()* we emit a message, **without** taking waiting broker response - if message was sent successfully or not.

2. **Sync**: Using *send().get()* we are waiting (**blocking**) for a reply from the broker. If there are any non-retriable errors we will get them in the catch block.

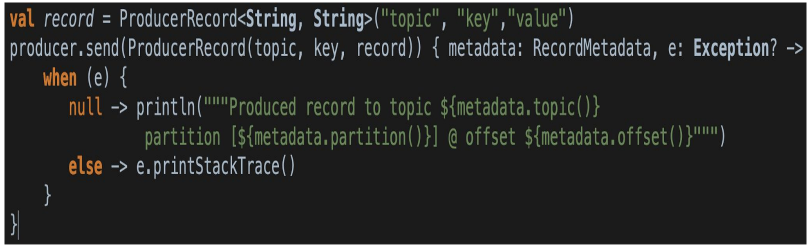
3. **Async:** Adding a *callback* in send() method, we receive an asynchronous response (**non-blocking**) from the broker, if the message was sent with success or not.



```
val record = ProducerRecord<String, String>("topic", "key","value")
try {
   procuder.send(record)
} catch (e: Exception) {
   println(e.stackTrace)
}
```



```
val record = ProducerRecord<String, String>("topic", "key","value")
try {
   procuder.send(record).get()
} catch (e: Exception) {
   println(e.stackTrace)
}
```



```
val record = ProducerRecord<String, String>("topic", "key","value")
producer.send(ProducerRecord(topic, key, record)) { metadata: RecordMetadata, e: Exception? ->
   when (e) {
      null -> println("""Produced record to topic ${metadata.topic()}
                   partition [${metadata.partition()}] @ offset ${metadata.offset()}""")
      else -> e.printStackTrace()
   }
}
```

Kafka Producer - Configuration (1)

Let’s explore some ***required producer configurations***.

1. **bootstrap.servers** -> List of brokers, so that the producer can find the Kafka cluster. Example host:port
2. **key.serializer** -> Serializer class for ProducerRecord’s key which implements the *org.apache.kafka.common.serialization.Serializer* interface.
3. **value.serializer** -> Serializer class for ProducerRecord’s value that implements the *org.apache.kafka.common.serialization.Serializer* interface.

Kafka Producer - Configuration (2)

Let’s explore some ***core producer configurations***.

4. **client.id** -> Allows to easily correlate requests on the broker with the client instance which made it
5. **acks** -> Controls the durability of messages written to Kafka
   a. *acks=0*: Producer will **not wait** for a reply from the broker if the message was sent successfully. No guarantee that the message was successfully written to the broker.
   b. acks=1 (default): Producer is **waiting** until receives a success response **from** the partition **leader** that the write succeeded. Message still can get lost if leader crashes and a follower replica which is promoted to leader does not have the message.
   c. *acks=all*: **Strongest** guarantee, as producer is **waiting** until the message is successfully written in the partition **leader** and replicated to all of the **in-sync replicas**. Increases latency.

Kafka Producer - Configuration (3)

6. **retries** -> Controls how many times Producer will retry in case of error.
7. **enable.idempotence** -> When set to 'true', the producer will ensure that exactly one copy of each message is written in the stream.
8. **max.in.flight.requests.per.connection** -> Max number of unacknowledged requests the client will send on a single connection before blocking. Note that if this config is set to be greater than 1 and enable.idempotence is set to false, there is a risk of message re-ordering after a failed send due to retries (i.e., if retries are enabled).
9. **batch.size** -> Controls amount in bytes for each batch.

Kafka Consumer - Concepts (1)

- **Reads** messages from Kafka topics.
- An **offset** (increasing integer value) is used to track which messages are already consumed.
- Each message in a specific partition has a **unique offset**.
- **Consumer group** is a set of consumers. Each partition is assigned and consumed **only** by **one** consumer.
- Every consumer group has a **group coordinator** which is one of the brokers.
- **Group coordinator** is responsible for managing the members of the group and partition’s assignments.

Kafka Consumer - Concepts (2)

- Can have **multiple apps** / clients read same topics. Each app gets all topic's messages.
- When a new consumer is **added** to the group or a consumer **crashes**, then a **rebalance process** starts.
- During rebalance stage, **partition ownership** is moving from one consumer to another and **can not** consume messages.
- Sends **heartbeats** to Kafka broker (group's coordinator), consumers maintain **membership** in group and **partition ownership**.
- If consumer **stops** sending **heartbeats**, group coordinator consider it as dead and triggers **rebalance**.

Kafka Consumer - Poll Loop

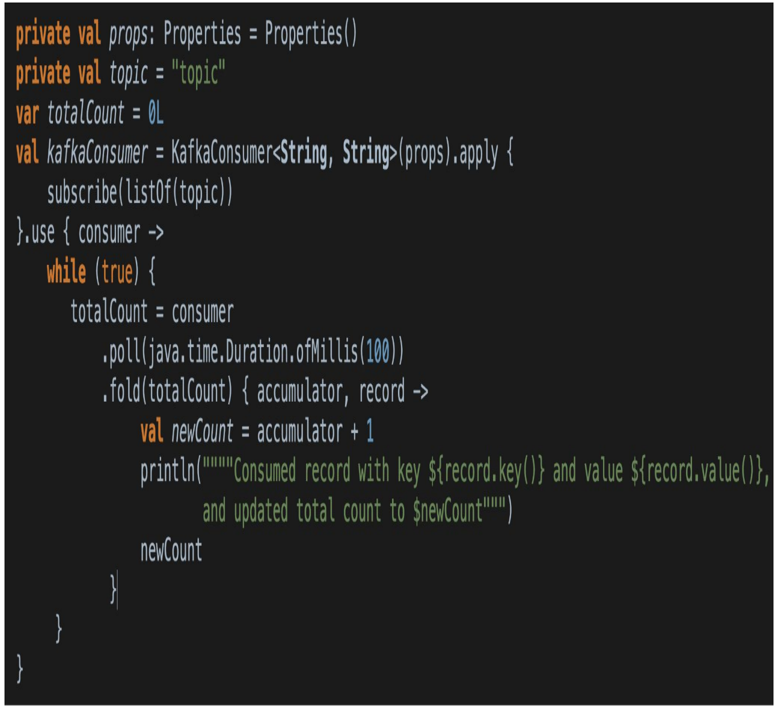
1. When the consumer starts up, it **finds** the group coordinator and requests to join the **group**.

2. After the consumer's assignment to its group by the coordinator, consumer must **find** the **initial position (offset)** for each assigned partition.

3. After consumer starts **reading messages** from the assigned partitions, it must **commit the offsets** for the messages it has read.

4. The **offset commit policy** is important to ensure the message delivery guarantee.

5. To maintain group membership and partition ownership, the consumer sends **heartbeats** to coordinator regularly.



```
private val props: Properties = Properties()
private val topic = "topic"
var totalCount = 0L
val kafkaConsumer = KafkaConsumer<String, String>(props).apply {
    subscribe(listOf(topic))
}.use { consumer ->
    while (true) {
        totalCount = consumer
            .poll(java.time.Duration.ofMillis(100))
            .fold(totalCount) { accumulator, record ->
                val newCount = accumulator + 1
                println("""Consumed record with key ${record.key()} and value ${record.value()},
                        and updated total count to $newCount""")
                newCount
            }
    }
}
```

Kafka Consumer - Configuration (1)

Let’s explore some ***required*** consumer *configurations*.

1. **bootstrap.servers** -> List of brokers, so that the consumer can find the Kafka cluster. Example host:port
2. **key.deserializer**-> Deserializer class for record’s key which implements the *org.apache.kafka.common.serialization.Deserializer* interface.
3. **value.deserializer** -> Deserializer class for record’s value that implements the *org.apache.kafka.common.serialization.Deserializer* interface.
4. **group.id** -> A unique string that identifies the consumer group this consumer belongs to.

Kafka Consumer - Configuration (2)

Let’s explore some ***core consumer*** *configurations*.

5. **enable.auto.commit** -> Controls if the consumer will commit offset automatically or not. Using auto commit gives us “at least once” delivery, as Kafka guarantees that no messages will be missed, but duplicates are possible.
6. **auto.commit.interval.ms** -> The frequency in milliseconds that the consumer offsets are auto-committed to Kafka if enable.auto.commit is set to true.
7. **fetch.min.bytes** -> Specifies the minimum amount of data received from the broker.
8. **heartbeat.interval.ms** -> The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities.
9. **auto.offset.reset** -> Specifies what to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server. E.g. *earliest*: automatically reset the offset to the earliest offset, while *latest:* automatically reset the offset to the latest offset.

Kafka Consumer - Commits & Offsets (1)

- To commit an offset, the consumer emits a message to Kafka on a **special topic** called **__consumer_offsets**, for the read messages for each partition.

- A **rebalance** will be triggered if a consumer joins or crashes, and the new one will read the last committed offset for its assigned partition.

- Rebalance has two phases: **partition revocation** and **partition assignment**.
  - The *revocation* is always called before a rebalance and is our last chance to commit offsets before partitions reassignment.
  - The *assignment* method can be used to set the initial position of the assigned partitions.

- Messages can be processed twice (**duplication**), if the committed offset is smaller than the last message offset.

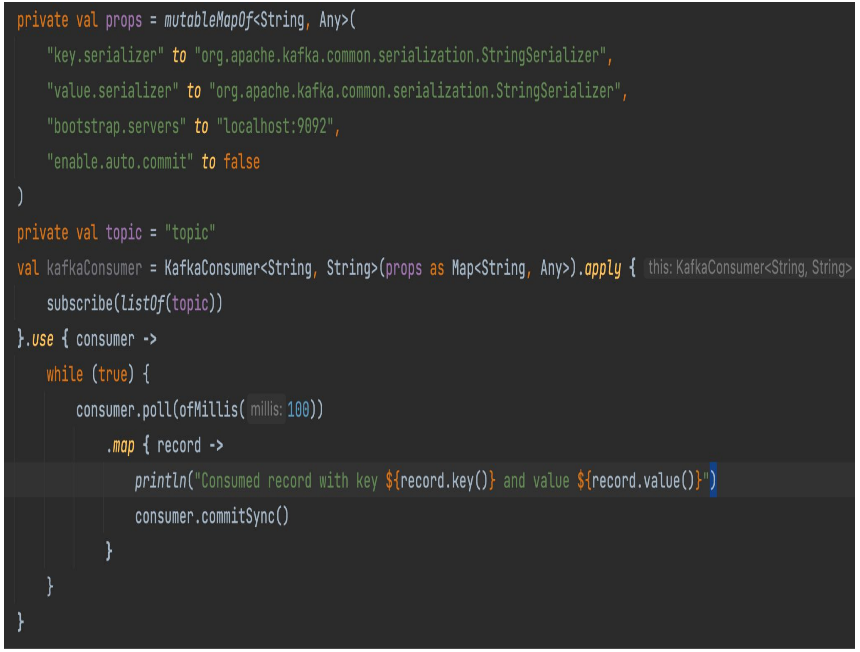
Kafka Consumer - Commits & Offsets (2)

A consumer can commit offset using **3 different ways**:

1. **Automatic Commit**: (at least once delivery)
   a. Set *enable.auto.commit* = true
   b. By default, every 5 secs, the largest offset received from poll() is **automatically** committed.
   c. We can select the **time-window** for auto committing offset using *auto.commit.interval.ms*.
   d. Auto-commit works as a **cron job** and the *auto.commit.interval.ms* config sets the time-window for triggering.
   e. If a consumer crashes before *auto.commit.interval.ms* period, a rebalance will be triggered, and new consumer's offset will be older than actual message offset, which leads to **duplications**, as it will re-read same messages.
   f. Automatic commits are convenient, but give **no** enough control on **reducing duplications**.

Kafka Consumer - Commits & Offsets (3)

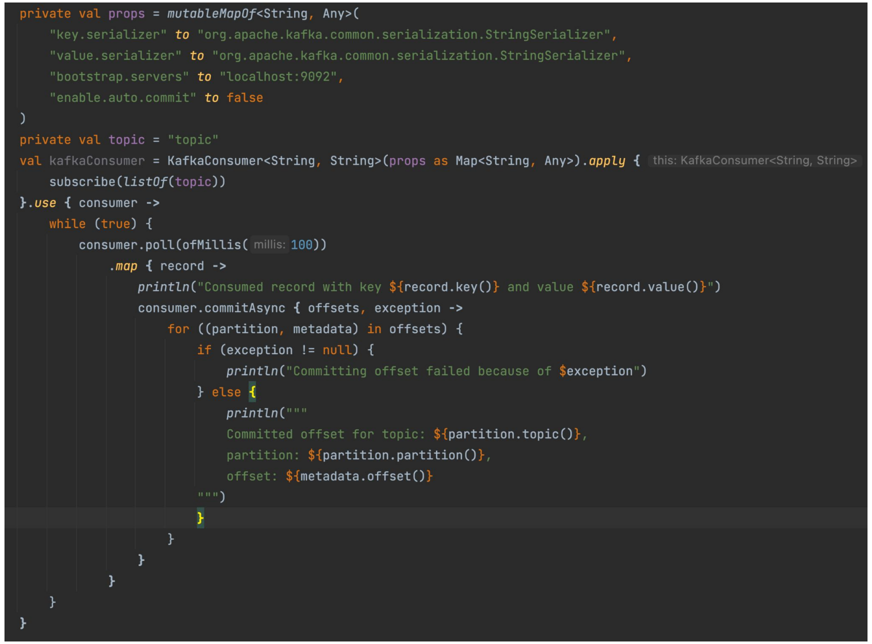
2. **Sync Commit**:
   a. Set *enable.auto.commit* = false
   b. *commitSync()* commits latest offset returned by poll() **synchronously**.
   c. This method should be called **after** processing all records, to **avoid** missing messages.
   d. The consumer is **blocked** until receives response from broker.
   e. If an error occurs, consumer will **retry** to commit until either succeeds or fails with non-retriable failure.



```kotlin
private val props = mutableMapOf<String, Any>(
    "key.serializer" to "org.apache.kafka.common.serialization.StringSerializer",
    "value.serializer" to "org.apache.kafka.common.serialization.StringSerializer",
    "bootstrap.servers" to "localhost:9092",
    "enable.auto.commit" to false
)
private val topic = "topic"
val kafkaConsumer = KafkaConsumer<String, String>(props as Map<String, Any>).apply {  this: KafkaConsumer<String, String>
    subscribe(listOf(topic))
}.use { consumer ->
    while (true) {
        consumer.poll(ofMillis( millis: 100))
            .map { record ->
                println("Consumed record with key ${record.key()} and value ${record.value()}")
                consumer.commitSync()
            }
    }
}
```

Kafka Consumer - Commits & Offsets (4)

3. **Async Commit**:
   a. Set *enable.auto.commit* = false
   b. *commitAsync()* commits latest offset returned by poll() **asynchronously**.
   c. This method should be called after **processing** all records, to avoid missing messages.
   d. Consumer sends the request and return immediately **without** blocking.
   e. Using commitAsync(), we are **not** able to **retry** if case of retryable error.



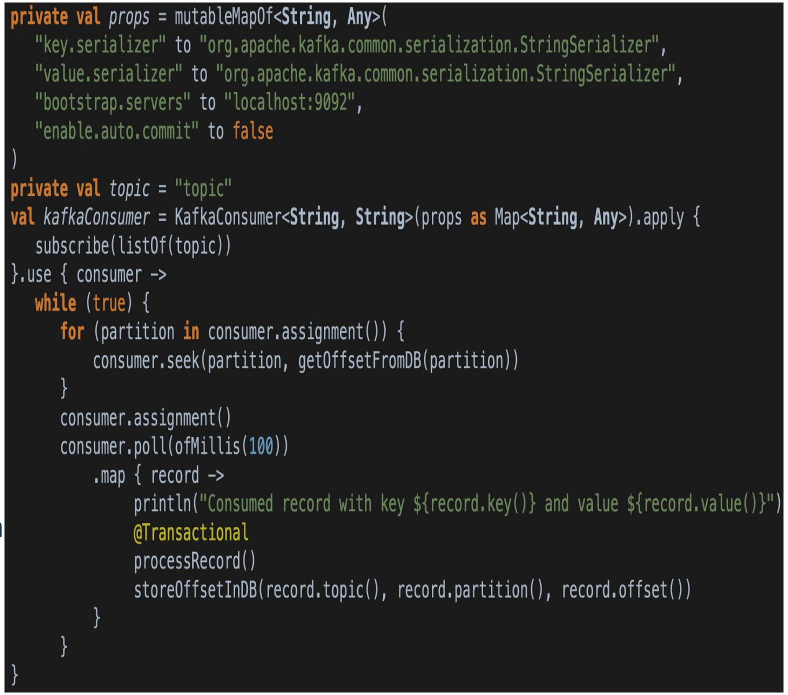
```
private val props = mutableMapOf<String, Any>(
    "key.serializer" to "org.apache.kafka.common.serialization.StringSerializer",
    "value.serializer" to "org.apache.kafka.common.serialization.StringSerializer",
    "bootstrap.servers" to "localhost:9092",
    "enable.auto.commit" to false
)
private val topic = "topic"
val kafkaConsumer = KafkaConsumer<String, String>(props as Map<String, Any>).apply {  this: KafkaConsumer<String, String>
    subscribe(listOf(topic))
}.use { consumer ->
    while (true) {
        consumer.poll(ofMillis( millis: 100))
            .map { record ->
                println("Consumed record with key ${record.key()} and value ${record.value()}")
                consumer.commitAsync { offsets, exception ->
                    for ((partition, metadata) in offsets) {
                        if (exception != null) {
                            println("Committing offset failed because of $exception")
                        } else {
                            println("""
                            Committed offset for topic: ${partition.topic()},
                            partition: ${partition.partition()},
                            offset: ${metadata.offset()}
                        """)
                        }
                    }
                }
            }
    }
}
```

Kafka Consumer - Commits & Offsets (5)

- There is still the chance of **duplication** when using commitAsync() or commitSync(). E.g. if app crashes after record's process but before commit offset. In this case the same messages will be re-processed.

- A way to remove duplicates, is to use a **database table** which keeps track of events that have already been **consumed** and avoid reprocessing them again.

- Also to reduce duplicates, we can **store offset** in a database's transaction with **processed events**. This way is an atomic operation and both or neither can happen. We can use **consumer.seek()** method to fetch specific offset from db and set it on consumer.



```kotlin
private val props = mutableMapOf<String, Any>(
   "key.serializer" to "org.apache.kafka.common.serialization.StringSerializer",
   "value.serializer" to "org.apache.kafka.common.serialization.StringSerializer",
   "bootstrap.servers" to "localhost:9092",
   "enable.auto.commit" to false
)
private val topic = "topic"
val kafkaConsumer = KafkaConsumer<String, String>(props as Map<String, Any>).apply {
   subscribe(listOf(topic))
}.use { consumer ->
   while (true) {
      for (partition in consumer.assignment()) {
         consumer.seek(partition, getOffsetFromDB(partition))
      }
      consumer.assignment()
      consumer.poll(ofMillis(100))
         .map { record ->
            println("Consumed record with key ${record.key()} and value ${record.value()}")
            @Transactional
            processRecord()
            storeOffsetInDB(record.topic(), record.partition(), record.offset())
         }
      }
}
```

Kafka Streams Concepts (1)

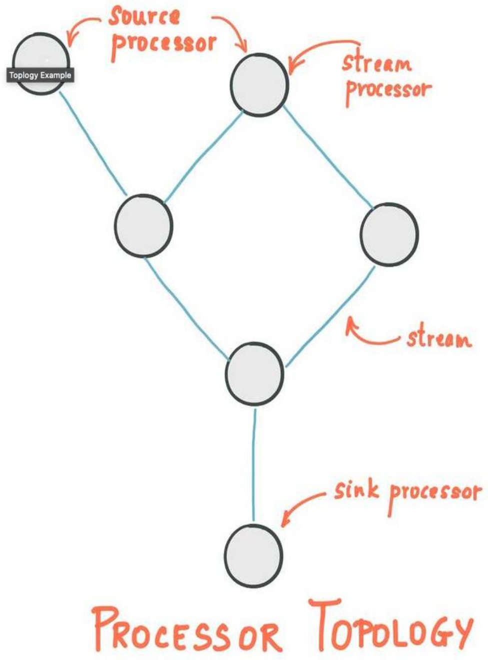
- A client library for stream processing applications built on top of Apache Kafka.
- Processes records from Kafka topics.
- Stream processing is the ongoing, and record-by-record real-time processing of data.
- Wrapper around Kafka Producer and Consumer.
- All Kafka topics are stored as a stream.
- It supports as primitives **KStreams** and **KTables** types and high level Streams DSL (**KSQL**).
  - KStreams: provides immutable data. It supports only inserting (appending) new events and It's a never-ending flow of data in a stream
  - KTables: is just an abstraction of the stream, where only the latest value is kept and allows for fast key lookups. It always *automatically updates* latest state if new value arrives on steam
  - A KStream can be interpreted as a KTable, and a KTable can be interpreted as a KStream.
- **Topology** describes processing flow defined a graph of stream processors (nodes) that are connected by streams (edges)

Kafka Streams Concepts (2)

- Can be easily embedded in any Java / Kotlin / Scala application.
- Fully integrated with Kafka security.
- No external dependencies on systems other than Apache Kafka itself.
- Supports **fault-tolerant local state** for stateful operations (joins, window aggregations).
- One-record-at-a-time processing to achieve millisecond processing latency.
- Supports event-time based windowing operations with late arrival of records.
- Supports **exactly-once processing**: Each record will be processed once and only once even when there is a failure on either Streams clients or Kafka brokers in the middle of processing. We have to set **processing.guarantee=exactly_once**

Kafka Streams Concepts (3)

- **Stream**: An ordered, replayable, and fault-tolerant sequence of immutable data records (key-value pair).
- **Stream Processor**: Is a node in the processor topology; it represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in the topology, applying its operation to it, and may subsequently produce one or more output records to its downstream processors.
- **Source Processor**: Is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its down-stream processors.
- **Sink Processor**: Is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.



Kafka Streams Example (1)

```kotlin
@ApplicationScoped
class StreamBuilder {
    fun startStreaming() {
        val topology = createTopology()
        val streams = KafkaStreams(topology, getStreamConfig())
        streams.start()
    }

    fun getStreamConfig(): Properties {
        val props = Properties()
        props[StreamsConfig.APPLICATION_ID_CONFIG] = APPLICATION_ID
        props[StreamsConfig.BOOTSTRAP_SERVERS_CONFIG] = BROKERS
        props[StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG] = Serdes.String().javaClass
        props[StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG] = Serdes.String().javaClass
        props[StreamsConfig.PROCESSING_GUARANTEE_CONFIG] = EXACTLY_ONCE_PROCESSING
        props[ConsumerConfig.AUTO_OFFSET_RESET_CONFIG] = LATEST_OFFSET_RESET
        return props
    }

    fun createTopology(): Topology {
        val builder = StreamsBuilder()
        builder.stream<String, String>(INPUT_TOPIC)
            .flatMapValues { value ->
                value.lowercase(Locale.getDefault())
                    .split(" ".toRegex())
                    .dropLastWhile { it.isEmpty() }
                    .toList()
            }.groupBy { _, value -> value }
            .count()
            .toStream()
            .to(OUTPUT_TOPIC, Produced.with(Serdes.String(), Serdes.Long()))

        return builder.build()
    }

    companion object {
        private const val APPLICATION_ID = "word-count-application"
        const val INPUT_TOPIC = "TextLines"
        const val OUTPUT_TOPIC = "WordsCount"
        private const val BROKERS = "kafka-broker:9092"
        private const val EXACTLY_ONCE_PROCESSING = "exactly_once"
        private const val LATEST_OFFSET_RESET = "latest"
    }
}
```

```kotlin
@QuarkusTest
class StreamBuilderTest {
    @Inject
    lateinit var streamBuilder: StreamBuilder
    private var testDriver: TopologyTestDriver? = null
    private var inputTopic: TestInputTopic<String, String>? = null
    private var outputTopic: TestOutputTopic<String, Long>? = null

    @BeforeEach
    fun setup() {
        testDriver = TopologyTestDriver(streamBuilder.createTopology(), streamBuilder.getStreamConfig())
        inputTopic = testDriver?.createInputTopic(
            StreamBuilder.INPUT_TOPIC,
            Serdes.String().serializer(),
            Serdes.String().serializer(),
            Instant.parse("2021-05-12T18:00:00Z"),
            Duration.ofMinutes(1)
        )
        outputTopic = testDriver?.createOutputTopic(
            StreamBuilder.OUTPUT_TOPIC,
            Serdes.String().deserializer(),
            Serdes.Long().deserializer()
        )
    }
    @AfterEach
    fun tearDown() {
        testDriver?.close()
    }

    @Test
    fun test() {
        val inputValues = listOf("A", "A", "B", "C", "B", "A", "A", "D", "C")
        val expectedList = listOf(
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "a", 1L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "a", 2L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "b", 1L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "c", 1L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "b", 2L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "a", 3L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "a", 4L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "d", 1L),
            ProducerRecord(StreamBuilder.OUTPUT_TOPIC, "c", 2L)
        )
        // Input
        for (inputValue in inputValues) {
            inputTopic?.pipeInput(inputValue)
        }
        // Output
        for (expected in expectedList) {
            val output = outputTopic?.readKeyValue()
            assertEquals(output?.key, expected.key())
            assertEquals(output?.value, expected.value())
        }
    }
}
```

Kafka Streams Example (2)

In previous code examples we are trying to find character counts:

- Via *createTopology()* method, we create stream's topology, which accepts strings and finds the counter of its char.
- Via *getStreamConfig()* method, we add all the required KafkaStream configurations.
- Via *startStreaming()* method, we initialize KStream and start streaming.
- As we can see on the tests, our topology pipeline works as we get the expected output.

Kafka Reliable Data Delivery

- **Reliability** is referring to a system in which behavior is guaranteed.
  - Kafka provides order guarantee in each partition
  - Produced messages are considered committed when they are written to all in-sync replicas.
  - Messages which are committed will not be lost if one at least replica is alive.
  - Consumers can only read committed messages
- **Durability** ensures that once the operation has been carried out, it will persist and cannot be undone even if the system fails.
  - Kafka with multiple replicas provides durability.

Kafka configurations for Reliable Data Delivery (1)

On **broker level** we can set some **configuration** to provide reliable message storage and delivery:

1. **Replication Factor**
   default.replication.factor specifies each *partition's replicas*. A replication factor of N **allows loss** N-1 brokers, but the system will remain **reliable**. Higher replication factor means higher availability and reliability, but needs more disc capacity. (tradeoff among availability / capacity)

2. **Unclean Leader Election**
   unclean.leader.election.enable specifies if a **replica** which is **out-of-sync** can be selected as **leader**. Setting it to **false** we ensure that the electing replica is always in-sync with the leader, preventing losing messages but we wait until the original leader is online again, resulting in lower availability. (tradeoff among consistency / availability)

3. **Minimum In-Sync Replicas**
   Using *min.insync.replicas* we specify the minimum number of replicas that must *acknowledge* a write to be considered successful. Preferred value: **replication.factor - 1**.

Kafka configurations for Reliable Data Delivery (2)

On **producer level** we can set some configuration to provide reliable message storage and delivery:

1. **Send Acknowledgments**
   Best approach is setting *acks=all* to wait until leader and all in-sync replicas get the message, before is considered as “committed”. It *reduces* messages *lost risk* but is the slowest option, which can be improved by combining it with **async send** method and **batches**.

2. **Configuring Producer Retries**
   It is essential, the producer handles **retryable** errors. If we want to avoid losing messages, then our producer should always retry to send them.

Kafka configurations for Reliable Data Delivery (3)

On **consumer level** we can set some configuration to provide reliable message storage and delivery:

- **Manually** commit offsets, instead of auto-commit them, to have more **control** on **offsets** and we can **reduce duplications**.
- Reducing missing messages on consumer side via **committing offset** after **events** were **processed** and not after polling.
- Design **rebalances** properly (e.g. commit offset before a partition is revoked).
- **Enforce EoS** (exactly-once delivery). We can achieve it via database table to store already processed messages with unique id and storing offset in database in a transaction. This way manipulate records and commit offset becomes a atomic operation. This way we avoid re-process same messages.
- **Retry** on errors. We can write retryable errors in a separate topic and continue. These messages can be processed later by another service.

Kafka Use Cases

Kafka allows us to have a huge amount of messages. As a result, it's ideal to be used as the heart of our system’s architecture, which connects different applications. Below we can see some **use cases**:

- Messaging
- Website Activity Tracking
- Metrics
- Real-time data processing
- Event Sourcing
- Log Aggregation

Kafka Summary

- Apache Kafka is a distributed streaming platform.
- Kafka stores messages in topics.
- Topics are partitioned and replicated across multiple brokers in a cluster.
- Producers send messages to topics from which consumers read.
- Kafka is able to process streams of events real-time.
- Kafka is capable of handling enormous amount of events.
- Kafka provides low-latency, high-throughput, fault-tolerance and great scalability.

Let’s see some code snippets!

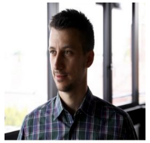
(https://github.com/arconsis/Eshop-EDA)

References

- Kafka The Definitive Guide by Neha Narkhede, Gwen Shapira, Todd Palino: https://www.amazon.com/Kafka-Definitive-Real-Time-Stream-Processing/dp/1491936169
- Designing event driven systems by Ben Stopford (https://www.confluent.io/designing-event-driven-systems/)
- https://docs.confluent.io/platform/current/installation/configuration
- https://docs.confluent.io/platform/current/clients/
- https://kafka.apache.org/
- https://kafka.apache.org/26/documentation/
- https://betterprogramming.pub/thorough-introduction-to-apache-kafka-6fbf2989bbc1
- https://jaceklaskowski.gitbooks.io/apache-kafka/content/
- https://preparingforcodinginterview.wordpress.com/2020/07/18/kafka-consumer-group-coordinator-consumer-group-leader/

Contact Info

- **Arconsis**:
  - Website: https://www.arconsis.com
  - Github: https://github.com/arconsis
- **Dimos Botsaris**:
  - Website: https://www.eldimious.com/
  - Github: https://github.com/eldimious
  - Email: botsaris.d@gmail.com
- **Alexandros Koufatzis**:
  - Github: https://github.com/akoufa
  - Email: akoufa@gmail.com



Thank you

Alexandros and Dimosthenis

arconsis