Kafka Streams (or **Streams API), a light weight stream-processing library written in Java which leverages Kafka as its internal messaging layer**. It was added in the Kafka 0.10.0.0 release.

The library allows for the development of stateful stream-processing applications that are scalable, elastic, and fully fault-tolerant. **The main API is a stream-processing domain-specific language (DSL) that offers high-level operators like filter, map, grouping, windowing, aggregation, joins, and the notion of tables.** Additionally, the Processor API can be used to implement custom operators for a more low-level development approach.

In every industry, businesses are adopting stream processing platforms as a key capability to deliver the promise of digital business transformation. **Stream processing helps in unlocking data from application silos by enabling data to flow as stream of events from systems of record to systems of engagement in near real time**.

**Tutorial: Creating a Streaming Data Pipeline**

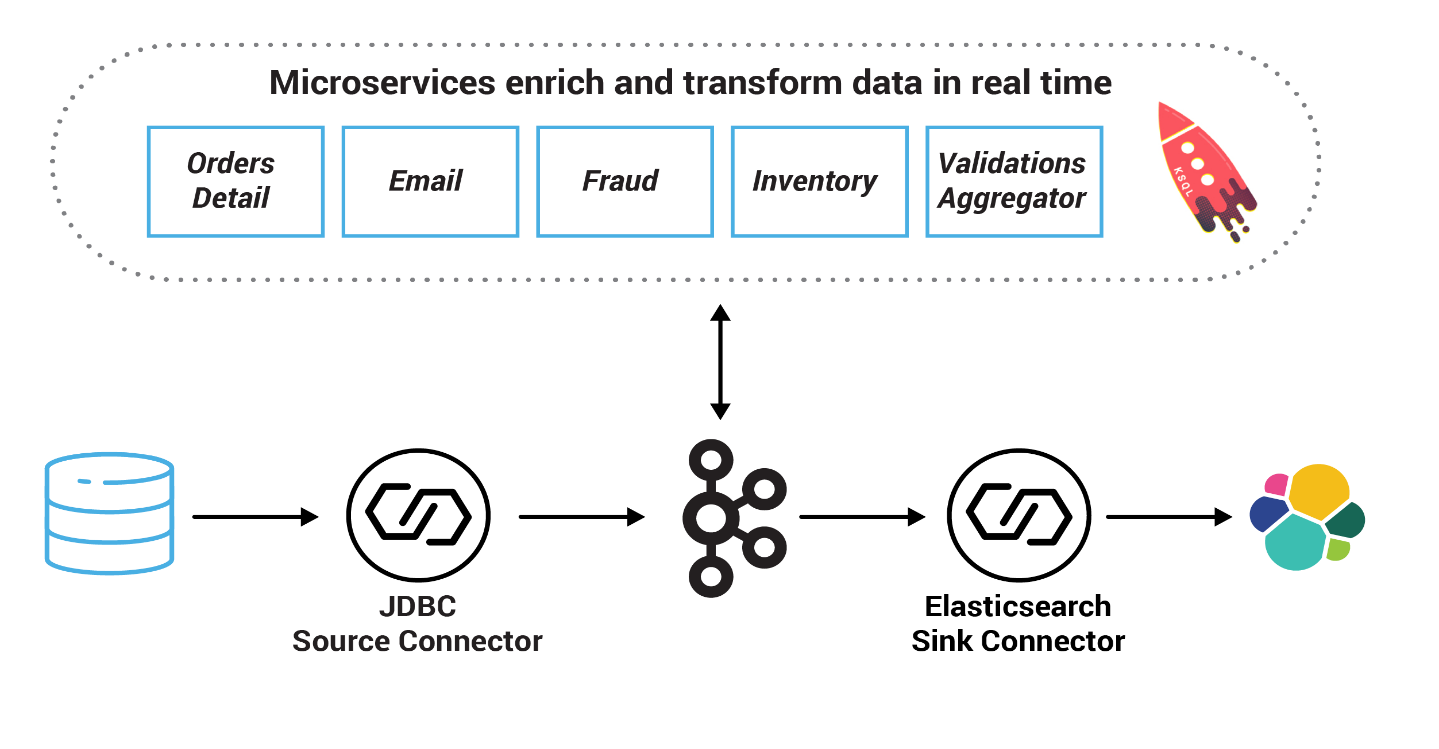
This quick start provides you with a first hands-on look at Kafka's Streams API. **It will demonstrate how to run your first Java application that uses the Kafka Streams library by showcasing a simple end-to-end data pipeline powered by Kafka.**

Before proceeding we need to go through Kafka Streams

Kafka Streams is a client library for building applications and microservices, **where the input and output data are stored in a Kafka cluster**. It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology.

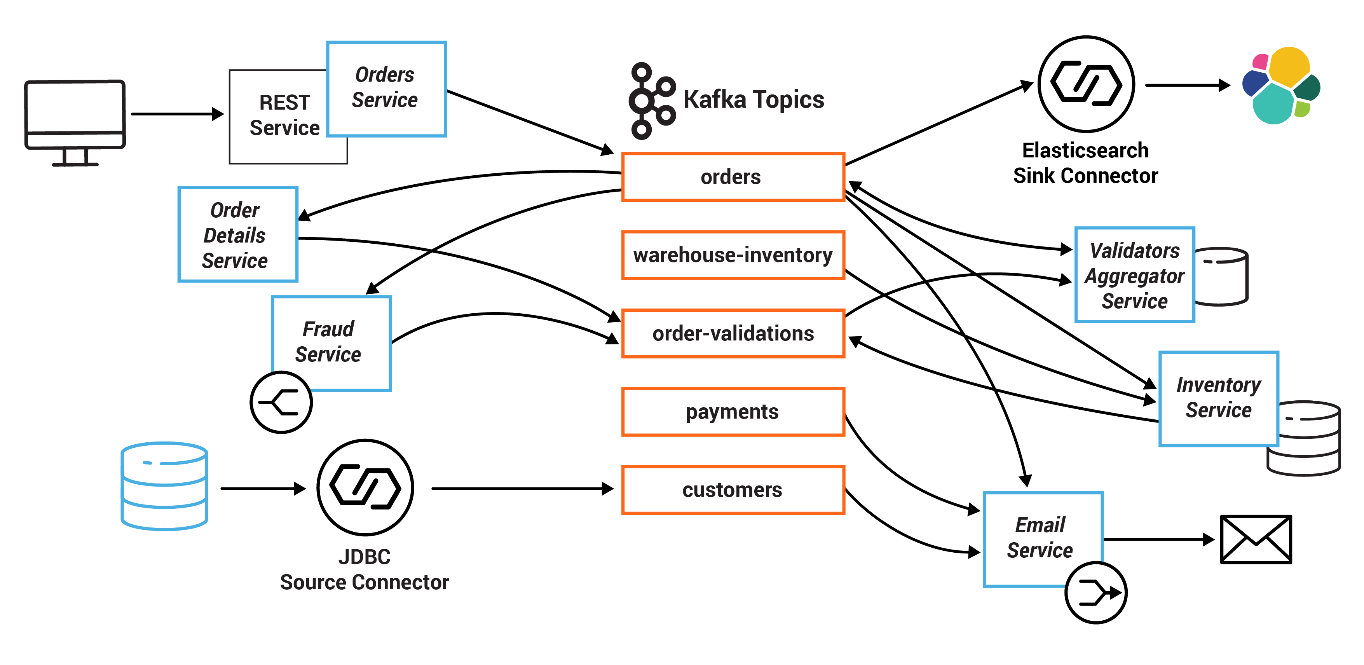
# Tutorial: Introduction to Streaming Application Development

The tutorial is based on a small microservices ecosystem, showcasing an order management workflow, such as one might find in retail and online shopping. It is built using Kafka Streams, whereby business events that describe the order management workflow propagate through this ecosystem.



### **Microservices**

Now let’s see a sample microservices example



In this example, the system centers on an Orders Service which exposes a REST interface to POST and GET Orders. Posting an Order creates an event in Kafka that is recorded in the topic orders. This is picked up by different validation engines (Fraud Service, Inventory Service and Order Details Service), which validate the order in parallel, emitting a PASS or FAIL based on whether each validation succeeds.

The result of each validation is pushed through a separate topic, Order Validations, so that we retain the \_single[writer\_](https://docs.confluent.io/current/tutorials/examples/microservices-orders/docs/index.html#id4) status of the Orders Service —> Orders Topic (Ben Stopford's [book](https://www.confluent.io/designing-event-driven-systems) discusses several options for managing consistency in event collaboration). The results of the various validation checks are aggregated in the Validation Aggregator Service, which then moves the order to a Validated or Failed state, based on the combined result.

To allow users to GET any order, the Orders Service creates a queryable materialized view (embedded inside the Orders Service), using a state store in each instance of the service, so that any Order can be requested historically. Note also that the Orders Service can be scaled out over a number of nodes, in which case GET requests must be routed to the correct node to get a certain key. This is handled automatically using the interactive queries functionality in Kafka Streams.

The Orders Service also includes a blocking HTTP GET so that clients can read their own writes. In this way, we bridge the synchronous, blocking paradigm of a RESTful interface with the asynchronous, non-blocking processing performed server-side.

There is a simple service that sends emails, and another that collates orders and makes them available in a search index using Elasticsearch.

Finally, KSQL is running with persistent queries to enrich streams and to also check for fraudulent behavior.

All the services are client applications written in Java, and they use the Kafka Streams API. The java source code for these microservices are in the [kafka-streams-examples repo](https://github.com/confluentinc/kafka-streams-examples/tree/5.1.2-post/src/main/java/io/confluent/examples/streams/microservices).

Summary of services and the topics they consume from and produce to:

| **Service** | **Consumes From** | **Produces To** |
| --- | --- | --- |
| InventoryService | orders, warehouse-inventory | order-validations |
| FraudService | orders | order-validations |
| OrderDetailsService | orders | order-validations |
| ValidationsAggregatorService | order-validations, orders | orders |
| EmailService | orders, payments, customers | platinum, gold, silver, bronze |
| OrdersService |  | orders |

This quick start shows how to run the [WordCount demo application](https://github.com/apache/kafka/blob/2.1/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java) that is included in Apache Kafka. Here's the gist of the code, converted to use Java 8 lambda expressions so that it is easier to read (taken from the variant [WordCountLambdaExample](https://github.com/confluentinc/kafka-streams-examples/tree/5.1.2-post/src/main/java/io/confluent/examples/streams/WordCountLambdaExample.java)):

**WordCountLambdaExample.java**

|  |
| --- |
| /\* |
|  | \* Copyright Confluent Inc. |
|  | \* |
|  | \* Licensed under the Apache License, Version 2.0 (the "License"); |
|  | \* you may not use this file except in compliance with the License. |
|  | \* You may obtain a copy of the License at |
|  | \* |
|  | \* http://www.apache.org/licenses/LICENSE-2.0 |
|  | \* |
|  | \* Unless required by applicable law or agreed to in writing, software |
|  | \* distributed under the License is distributed on an "AS IS" BASIS, |
|  | \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. |
|  | \* See the License for the specific language governing permissions and |
|  | \* limitations under the License. |
|  | \*/ |
|  | package io.confluent.examples.streams; |
|  |  |
|  | import org.apache.kafka.common.serialization.Serde; |
|  | import org.apache.kafka.common.serialization.Serdes; |
|  | import org.apache.kafka.streams.KafkaStreams; |
|  | import org.apache.kafka.streams.StreamsBuilder; |
|  | import org.apache.kafka.streams.StreamsConfig; |
|  | import org.apache.kafka.streams.kstream.KStream; |
|  | import org.apache.kafka.streams.kstream.KTable; |
|  | import org.apache.kafka.streams.kstream.Produced; |
|  |  |
|  | import java.util.Arrays; |
|  | import java.util.Properties; |
|  | import java.util.regex.Pattern; |
|  |  |
|  | /\*\* |
|  | \* Demonstrates, using the high-level KStream DSL, how to implement the WordCount program that |
|  | \* computes a simple word occurrence histogram from an input text. This example uses lambda |
|  | \* expressions and thus works with Java 8+ only. |
|  | \* <p> |
|  | \* In this example, the input stream reads from a topic named "streams-plaintext-input", where the values of |
|  | \* messages represent lines of text; and the histogram output is written to topic |
|  | \* "streams-wordcount-output", where each record is an updated count of a single word, i.e. {@code word (String) -> currentCount (Long)}. |
|  | \* <p> |
|  | \* Note: Before running this example you must 1) create the source topic (e.g. via {@code kafka-topics --create ...}), |
|  | \* then 2) start this example and 3) write some data to the source topic (e.g. via {@code kafka-console-producer}). |
|  | \* Otherwise you won't see any data arriving in the output topic. |
|  | \* <p> |
|  | \* <br> |
|  | \* HOW TO RUN THIS EXAMPLE |
|  | \* <p> |
|  | \* 1) Start Zookeeper and Kafka. Please refer to <a href='http://docs.confluent.io/current/quickstart.html#quickstart'>QuickStart</a>. |
|  | \* <p> |
|  | \* 2) Create the input and output topics used by this example. |
|  | \* <pre> |
|  | \* {@code |
|  | \* $ bin/kafka-topics --create --topic streams-plaintext-input \ |
|  | \* --zookeeper localhost:2181 --partitions 1 --replication-factor 1 |
|  | \* $ bin/kafka-topics --create --topic streams-wordcount-output \ |
|  | \* --zookeeper localhost:2181 --partitions 1 --replication-factor 1 |
|  | \* }</pre> |
|  | \* Note: The above commands are for the Confluent Platform. For Apache Kafka it should be {@code bin/kafka-topics.sh ...}. |
|  | \* <p> |
|  | \* 3) Start this example application either in your IDE or on the command line. |
|  | \* <p> |
|  | \* If via the command line please refer to <a href='https://github.com/confluentinc/kafka-streams-examples#packaging-and-running'>Packaging</a>. |
|  | \* Once packaged you can then run: |
|  | \* <pre> |
|  | \* {@code |
|  | \* $ java -cp target/kafka-streams-examples-5.0.0-SNAPSHOT-standalone.jar io.confluent.examples.streams.WordCountLambdaExample |
|  | \* } |
|  | \* </pre> |
|  | \* 4) Write some input data to the source topic "streams-plaintext-input" (e.g. via {@code kafka-console-producer}). |
|  | \* The already running example application (step 3) will automatically process this input data and write the |
|  | \* results to the output topic "streams-wordcount-output". |
|  | \* <pre> |
|  | \* {@code |
|  | \* # Start the console producer. You can then enter input data by writing some line of text, followed by ENTER: |
|  | \* # |
|  | \* # hello kafka streams<ENTER> |
|  | \* # all streams lead to kafka<ENTER> |
|  | \* # join kafka summit<ENTER> |
|  | \* # |
|  | \* # Every line you enter will become the value of a single Kafka message. |
|  | \* $ bin/kafka-console-producer --broker-list localhost:9092 --topic streams-plaintext-input |
|  | \* }</pre> |
|  | \* 5) Inspect the resulting data in the output topic, e.g. via {@code kafka-console-consumer}. |
|  | \* <pre> |
|  | \* {@code |
|  | \* $ bin/kafka-console-consumer --topic streams-wordcount-output --from-beginning \ |
|  | \* --bootstrap-server localhost:9092 \ |
|  | \* --property print.key=true \ |
|  | \* --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer |
|  | \* }</pre> |
|  | \* You should see output data similar to below. Please note that the exact output |
|  | \* sequence will depend on how fast you type the above sentences. If you type them |
|  | \* slowly, you are likely to get each count update, e.g., kafka 1, kafka 2, kafka 3. |
|  | \* If you type them quickly, you are likely to get fewer count updates, e.g., just kafka 3. |
|  | \* This is because the commit interval is set to 10 seconds. Anything typed within |
|  | \* that interval will be compacted in memory. |
|  | \* <pre> |
|  | \* {@code |
|  | \* hello 1 |
|  | \* kafka 1 |
|  | \* streams 1 |
|  | \* all 1 |
|  | \* streams 2 |
|  | \* lead 1 |
|  | \* to 1 |
|  | \* join 1 |
|  | \* kafka 3 |
|  | \* summit 1 |
|  | \* }</pre> |
|  | \* 6) Once you're done with your experiments, you can stop this example via {@code Ctrl-C}. If needed, |
|  | \* also stop the Kafka broker ({@code Ctrl-C}), and only then stop the ZooKeeper instance (`{@code Ctrl-C}). |
|  | \*/ |
|  | public class WordCountLambdaExample { |
|  |  |
|  | public static void main(final String[] args) throws Exception { |
|  | final String bootstrapServers = args.length > 0 ? args[0] : "localhost:9092"; |
|  | final Properties streamsConfiguration = new Properties(); |
|  | // Give the Streams application a unique name. The name must be unique in the Kafka cluster |
|  | // against which the application is run. |
|  | streamsConfiguration.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "wordcount-lambda-example"); |
|  | streamsConfiguration.put(StreamsConfig.CLIENT\_ID\_CONFIG, "wordcount-lambda-example-client"); |
|  | // Where to find Kafka broker(s). |
|  | streamsConfiguration.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, bootstrapServers); |
|  | // Specify default (de)serializers for record keys and for record values. |
|  | streamsConfiguration.put(StreamsConfig.DEFAULT\_KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass().getName()); |
|  | streamsConfiguration.put(StreamsConfig.DEFAULT\_VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass().getName()); |
|  | // Records should be flushed every 10 seconds. This is less than the default |
|  | // in order to keep this example interactive. |
|  | streamsConfiguration.put(StreamsConfig.COMMIT\_INTERVAL\_MS\_CONFIG, 10 \* 1000); |
|  | // For illustrative purposes we disable record caches |
|  | streamsConfiguration.put(StreamsConfig.CACHE\_MAX\_BYTES\_BUFFERING\_CONFIG, 0); |
|  |  |
|  | // Set up serializers and deserializers, which we will use for overriding the default serdes |
|  | // specified above. |
|  | final Serde<String> stringSerde = Serdes.String(); |
|  | final Serde<Long> longSerde = Serdes.Long(); |
|  |  |
|  | // In the subsequent lines we define the processing topology of the Streams application. |
|  | final StreamsBuilder builder = new StreamsBuilder(); |
|  |  |
|  | // Construct a `KStream` from the input topic "streams-plaintext-input", where message values |
|  | // represent lines of text (for the sake of this example, we ignore whatever may be stored |
|  | // in the message keys). |
|  | // |
|  | // Note: We could also just call `builder.stream("streams-plaintext-input")` if we wanted to leverage |
|  | // the default serdes specified in the Streams configuration above, because these defaults |
|  | // match what's in the actual topic. However we explicitly set the deserializers in the |
|  | // call to `stream()` below in order to show how that's done, too. |
|  | final KStream<String, String> textLines = builder.stream("streams-plaintext-input"); |
|  |  |
|  | final Pattern pattern = Pattern.compile("\\W+", Pattern.UNICODE\_CHARACTER\_CLASS); |
|  |  |
|  | final KTable<String, Long> wordCounts = textLines |
|  | // Split each text line, by whitespace, into words. The text lines are the record |
|  | // values, i.e. we can ignore whatever data is in the record keys and thus invoke |
|  | // `flatMapValues()` instead of the more generic `flatMap()`. |
|  | .flatMapValues(value -> Arrays.asList(pattern.split(value.toLowerCase()))) |
|  | // Count the occurrences of each word (record key). |
|  | // |
|  | // This will change the stream type from `KStream<String, String>` to `KTable<String, Long>` |
|  | // (word -> count). In the `count` operation we must provide a name for the resulting KTable, |
|  | // which will be used to name e.g. its associated state store and changelog topic. |
|  | // |
|  | // Note: no need to specify explicit serdes because the resulting key and value types match our default serde settings |
|  | .groupBy((key, word) -> word) |
|  | .count(); |
|  |  |
|  | // Write the `KTable<String, Long>` to the output topic. |
|  | wordCounts.toStream().to("streams-wordcount-output", Produced.with(stringSerde, longSerde)); |
|  |  |
|  | // Now that we have finished the definition of the processing topology we can actually run |
|  | // it via `start()`. The Streams application as a whole can be launched just like any |
|  | // normal Java application that has a `main()` method. |
|  | final KafkaStreams streams = new KafkaStreams(builder.build(), streamsConfiguration); |
|  | // Always (and unconditionally) clean local state prior to starting the processing topology. |
|  | // We opt for this unconditional call here because this will make it easier for you to play around with the example |
|  | // when resetting the application for doing a re-run (via the Application Reset Tool, |
|  | // http://docs.confluent.io/current/streams/developer-guide.html#application-reset-tool). |
|  | // |
|  | // The drawback of cleaning up local state prior is that your app must rebuilt its local state from scratch, which |
|  | // will take time and will require reading all the state-relevant data from the Kafka cluster over the network. |
|  | // Thus in a production scenario you typically do not want to clean up always as we do here but rather only when it |
|  | // is truly needed, i.e., only under certain conditions (e.g., the presence of a command line flag for your app). |
|  | // See `ApplicationResetExample.java` for a production-like example. |
|  | streams.cleanUp(); |
|  | streams.start(); |
|  |  |
|  | // Add shutdown hook to respond to SIGTERM and gracefully close Kafka Streams |
|  | Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); |
|  | } |
|  |  |
|  | } |

**WordCountDemo.java**

|  |
| --- |
| /\* |
|  | \* Licensed to the Apache Software Foundation (ASF) under one or more |
|  | \* contributor license agreements. See the NOTICE file distributed with |
|  | \* this work for additional information regarding copyright ownership. |
|  | \* The ASF licenses this file to You under the Apache License, Version 2.0 |
|  | \* (the "License"); you may not use this file except in compliance with |
|  | \* the License. You may obtain a copy of the License at |
|  | \* |
|  | \* http://www.apache.org/licenses/LICENSE-2.0 |
|  | \* |
|  | \* Unless required by applicable law or agreed to in writing, software |
|  | \* distributed under the License is distributed on an "AS IS" BASIS, |
|  | \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. |
|  | \* See the License for the specific language governing permissions and |
|  | \* limitations under the License. |
|  | \*/ |
|  | package org.apache.kafka.streams.examples.wordcount; |
|  |  |
|  | import org.apache.kafka.clients.consumer.ConsumerConfig; |
|  | import org.apache.kafka.common.serialization.Serdes; |
|  | import org.apache.kafka.streams.KafkaStreams; |
|  | import org.apache.kafka.streams.StreamsBuilder; |
|  | import org.apache.kafka.streams.StreamsConfig; |
|  | import org.apache.kafka.streams.kstream.KStream; |
|  | import org.apache.kafka.streams.kstream.KTable; |
|  | import org.apache.kafka.streams.kstream.Produced; |
|  |  |
|  | import java.util.Arrays; |
|  | import java.util.Locale; |
|  | import java.util.Properties; |
|  | import java.util.concurrent.CountDownLatch; |
|  |  |
|  | /\*\* |
|  | \* Demonstrates, using the high-level KStream DSL, how to implement the WordCount program |
|  | \* that computes a simple word occurrence histogram from an input text. |
|  | \* <p> |
|  | \* In this example, the input stream reads from a topic named "streams-plaintext-input", where the values of messages |
|  | \* represent lines of text; and the histogram output is written to topic "streams-wordcount-output" where each record |
|  | \* is an updated count of a single word. |
|  | \* <p> |
|  | \* Before running this example you must create the input topic and the output topic (e.g. via |
|  | \* {@code bin/kafka-topics.sh --create ...}), and write some data to the input topic (e.g. via |
|  | \* {@code bin/kafka-console-producer.sh}). Otherwise you won't see any data arriving in the output topic. |
|  | \*/ |
|  | public final class WordCountDemo { |
|  |  |
|  | public static void main(final String[] args) { |
|  | final Properties props = new Properties(); |
|  | props.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "streams-wordcount"); |
|  | props.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092"); |
|  | props.put(StreamsConfig.CACHE\_MAX\_BYTES\_BUFFERING\_CONFIG, 0); |
|  | props.put(StreamsConfig.DEFAULT\_KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass().getName()); |
|  | props.put(StreamsConfig.DEFAULT\_VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass().getName()); |
|  |  |
|  | // setting offset reset to earliest so that we can re-run the demo code with the same pre-loaded data |
|  | // Note: To re-run the demo, you need to use the offset reset tool: |
|  | // https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool |
|  | props.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest"); |
|  |  |
|  | final StreamsBuilder builder = new StreamsBuilder(); |
|  |  |
|  | final KStream<String, String> source = builder.stream("streams-plaintext-input"); |
|  |  |
|  | final KTable<String, Long> counts = source |
|  | .flatMapValues(value -> Arrays.asList(value.toLowerCase(Locale.getDefault()).split(" "))) |
|  | .groupBy((key, value) -> value) |
|  | .count(); |
|  |  |
|  | // need to override value serde to Long type |
|  | counts.toStream().to("streams-wordcount-output", Produced.with(Serdes.String(), Serdes.Long())); |
|  |  |
|  | final KafkaStreams streams = new KafkaStreams(builder.build(), props); |
|  | final CountDownLatch latch = new CountDownLatch(1); |
|  |  |
|  | // attach shutdown handler to catch control-c |
|  | Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-shutdown-hook") { |
|  | @Override |
|  | public void run() { |
|  | streams.close(); |
|  | latch.countDown(); |
|  | } |
|  | }); |
|  |  |
|  | try { |
|  | streams.start(); |
|  | latch.await(); |
|  | } catch (final Throwable e) { |
|  | System.exit(1); |
|  | } |
|  | System.exit(0); |
|  | } |
|  | } |

[**WordCountLambdaExample**](https://github.com/confluentinc/kafka-streams-examples/tree/5.1.2-post/src/main/java/io/confluent/examples/streams/WordCountLambdaExample.java)

// Serializers/deserializers (serde) for String and Long types

final Serde<String> stringSerde = Serdes.String();

final Serde<Long> longSerde = Serdes.Long();

// Construct a `KStream` from the input topic "streams-plaintext-input", where message values

// represent lines of text (for the sake of this example, we ignore whatever may be stored

// in the message keys).

KStream<String, String> textLines = builder.stream("streams-plaintext-input", Consumed.with(stringSerde, stringSerde));

KTable<String, Long> wordCounts = textLines

// Split each text line, by whitespace, into words. The text lines are the message

// values, i.e. we can ignore whatever data is in the message keys and thus invoke

// `flatMapValues` instead of the more generic `flatMap`.

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

// We use `groupBy` to ensure the words are available as message keys

.groupBy((key, value) -> value)

// Count the occurrences of each word (message key).

.count();

// Convert the `KTable<String, Long>` into a `KStream<String, Long>` and write to the output topic.

wordCounts.toStream().to("streams-wordcount-output", Produced.with(stringSerde, longSerde));

# Kafka Connect

Kafka Connect, an open source component of Apache Kafka, is a framework for connecting Kafka with external systems such as databases, key-value stores, search indexes, and file systems.

Using Kafka Connect you can use existing connector implementations for common data sources and sinks to move data into and out of Kafka.

**Source Connector**

A source connector ingests entire databases and streams table updates to Kafka topics. It can also collect metrics from all of your application servers into Kafka topics, making the data available for stream processing with low latency.

**Sink Connector**

A sink connector delivers data from Kafka topics into secondary indexes such as Elasticsearch or batch systems such as Hadoop for offline analysis.

Kafka Connect is focused on streaming data to and from Kafka, making it simpler for you to write high quality, reliable, and high performance connector plugins. It also enables the framework to make guarantees that are difficult to achieve using other frameworks. Kafka Connect is an integral component of an ETL pipeline when combined with Kafka and a stream processing framework.

Kafka Connect can run either as a standalone process for running jobs on a single machine (e.g., log collection), or as a distributed, scalable, fault tolerant service supporting an entire organization. This allows it to scale down to development, testing, and small production deployments with a low barrier to entry and low operational overhead, and to scale up to support a large organization's data pipeline.

The main benefits of using Kafka Connect are:

* **Data Centric Pipeline** -- use meaningful data abstractions to pull or push data to Kafka.
* **Flexibility and Scalability** -- run with streaming and batch-oriented systems on a single node or scaled to an organization-wide service.
* **Reusability and Extensibility** -- leverage existing connectors or extend them to tailor to your needs and lower time to production.

# Kafka Clients

Confluent Platform includes client libraries for multiple languages that provide both low-level access to Kafka and higher level stream processing.

All JARs included in the packages are also available in the Confluent Maven repository. Here's a sample POM file showing how to add this repository:

<repositories>

<repository>

<id>confluent</id>

<url>https://packages.confluent.io/maven/</url>

</repository>

*<!-- further repository entries here -->*

</repositories>

The Confluent Maven repository includes compiled versions of Kafka.

To reference the Kafka version 2.1 that is included with Confluent Platform 5.1.2, use the following in your pom.xml:

<dependencies>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka\_2.11</artifactId>

<version>2.1.1-cp1</version>

</dependency>

*<!-- further dependency entries here -->*

</dependencies>

To use Confluent Platform serializers that integrate with the rest of the Confluent Platform you would include the following in your pom.xml:

<dependencies>

<dependency>

<groupId>io.confluent</groupId>

<artifactId>kafka-avro-serializer</artifactId>

*<!-- For Confluent Platform 5.1.2 -->*

<version>5.1.2</version>

</dependency>

*<!-- further dependency entries here -->*

</dependencies>

# Kafka Java Producer

Confluent Platform includes the Java producer shipped with Kafka.

This section gives a high-level overview of how the producer works, an introduction to the configuration settings for tuning, and some examples from each client library.

## Concepts

The Kafka producer is conceptually much simpler than the consumer since it has no need for group coordination. Its main function is to **map each message to a topic partition and send a produce request to the leader of that partition**. It does the first of these with a **partitioner**, **which typically selects a partition using a hash function**. The partitioners shipped with Kafka guarantee that all messages with the same non-empty key will be sent to the same partition. If no key is provided, then the partition is selected in a round-robin fashion to ensure an even distribution across the topic partitions.

**Each partition in the Kafka cluster has a leader and a set of replicas among the brokers.**

**All writes to the partition must go through the partition leader.**

**The replicas are kept in sync by fetching from the leader.**

When the leader shuts down or fails, the next leader is chosen from among the in-sync replicas. Depending on how the producer is configured, each produce request to the partition leader can be held until the replicas have successfully acknowledged the write. This gives the producer some control over message durability at some cost to overall throughput.

Messages written to the partition leader are not immediately readable by consumers regardless of the producer's acknowledgement settings.

**When all in-sync replicas have acknowledged the write, then the message is considered committed,** which makes it available for reading. This ensures that messages cannot be lost by a broker failure after they have already been read. Note that this implies that **messages which were acknowledged by the leader only** (i.e. **acks=1 {in java** config.put("acks", "all");**}**) **can be lost if the partition leader fails** before the replicas have copied the message. Nevertheless, this is often a reasonable compromise in practice to ensure durability in most cases while not impacting throughput too significantly.

## Configuration

**Core Configuration**:

You are required to set the **bootstrap.servers** property so that the producer can find the Kafka cluster. Although not required, you should always set a **client.id since this allows you to easily correlate requests on the broker with the client instance which made it**. These settings are the same for Java, C/C++, Python, Go and .NET clients.

**Message Durability**:

You can control the durability of messages written to Kafka through the **acks** setting.

**acks=1(default)** 🡪 The default value of "1" requires an explicit acknowledgement from the partition leader that the write succeeded.

**acks=all** 🡪  The strongest guarantee that Kafka provides is with "acks=all", which guarantees that **not only did the partition leader accept the write, but it was successfully replicated to all of the in-sync replicas**.

**acks=0** 🡪 You can also use a value of "0" to maximize throughput, but you will have no guarantee that the message was successfully written to the broker's log since the broker does not even send a response in this case. This also means that you will not be able to determine the offset of the message.

Note that for the C/C++, Python, Go and .NET clients, this is a per-topic configuration, but can be applied globally using the default\_topic\_conf sub-configuration in C/C++ and default.topic.config sub-configuration in Python, Go and .NET.

**Message Ordering**:

In general, messages are written to the broker in the same order that they are received by the producer client. However, if you enable message retries by setting **retries** to a value larger than 0 (which is the default), then message reordering may occur since the retry may occur after a following write succeeded. To enable retries without reordering, you can set **max.in.flight.requests.per.connection to 1** to ensure that only one request can be sent to the broker at a time. Without retries enabled, the broker will preserve the order of writes it receives, but there could be gaps due to individual send failures.

**Batching and Compression**:

Kafka producers attempt to collect sent messages into batches to improve throughput. With the Java client, you can use **batch.size** to control the maximum size in bytes of each message batch. To give more time for batches to fill, you can use **linger.ms** to have the producer delay sending. Compression can be enabled with the **compression.type**setting. Compression covers full message batches, so larger batches will typically mean a higher compression ratio.

With the C/C++, Python, Go and .NET clients, you can use **batch.num.messages** to set a limit on the number of messages contained in each batch. To enable compression, use **compression.codec**.

**Queuing Limits**:

Use **buffer.memory** to limit the total memory that is available to the Java client for collecting unsent messages. When this limit is hit, the producer will block on additional sends for as long as **max.block.ms** before raising an exception. Additionally, to avoid keeping records queued indefinitely, you can set a timeout using **request.timeout.ms**. If this timeout expires before a message can be successfully sent, then it will be removed from the queue and an exception will be thrown.

The C/C++, Python, Go and .NET clients have similar settings. Use **queue.buffering.max.messages** to limit the total number of messages that can be queued (for transmission, retries, or delivery reports) at any given time **queue.buffering.max.ms**limits the amount of time the client waits to fill up a batch before sending it to the broker.

### **Initial Setup**

The Java producer is constructed with a standard Properties file.

Properties config = new Properties();

config.put("client.id", InetAddress.getLocalHost().getHostName());

config.put("bootstrap.servers", "host1:9092,host2:9092");

config.put("acks", "all");

new KafkProducer<K, V>(config);

Configuration errors will result in a raised KafkaException from the constructor of KafkaProducer. The main difference in librdkafka is that it handles the errors for each setting directly:

### **Asynchronous Writes**

All writes are asynchronous by default. The Java producer includes a **send()** **API which returns a future which can be polled to get the result of the send**.

final ProducerRecord<K, V> = new ProducerRecord<>(topic, key, value);

Future<RecordMetadata> future = producer.send(record);

If you want to invoke some code after the write has completed you can also provide a callback. In Java this is implemented as a Callback object:

final ProducerRecord<K, V> = new ProducerRecord<>(topic, key, value);

producer.send(record, new Callback() {

public void onCompletion(RecordMetadata metadata, Exception e) {

if (e != null)

log.debug("Send failed for record {}", record, e);

}

});

### **Synchronous Writes**

To make writes synchronous, just wait on the returned future. This would typically be a bad idea since it would kill throughput, but may be justified in some cases.

Future<RecordMetadata> future = producer.send(record);

RecordMetadata metadata = future.get();

# Kafka Java Consumer

## Concepts

A **consumer group** is a set of consumers which cooperate to consume data from some topics. The partitions of all the topics are divided among the consumers in the group. As new group members arrive and old members leave, the partitions are re-assigned so that each member receives a proportional share of the partitions. This is known as **rebalancing the group**.

The main difference between the older "high-level" consumer and the new consumer is that the former depended on ZooKeeper for group management, while the latter uses a group protocol built into Kafka itself. In this protocol, one of the brokers is designated as the group's **coordinator** and is responsible for managing the members of the group as well as their partition assignments.

The coordinator of each group is chosen from the leaders of the internal offsets topic **\_\_consumer\_offsets**, which is used to store committed offsets. Basically the group's ID is hashed to one of the partitions for this topic and the leader of that partition is selected as the coordinator. In this way, management of consumer groups is divided roughly equally across all the brokers in the cluster, which allows the number of groups to scale by increasing the number of brokers.

When the consumer starts up, it finds the coordinator for its group and sends a request to join the group. The coordinator then begins a group rebalance so that the new member is assigned its fair share of the group's partitions. Every rebalance results in a new **generation** of the group.

Each **member in the group must send heartbeats to the coordinator in order to remain a member of the group**.

If no hearbeat is received before expiration of the configured **session timeout**, then the coordinator will kick the member out of the group and reassign its partitions to another member.

**Offset Management**: After the consumer receives its assignment from the coordinator, it must determine the initial position for each assigned partition. When the group is first created, before any messages have been consumed, the position is set according to a configurable offset reset policy (auto.offset.reset). Typically, consumption starts either at the earliest offset or the latest offset.

As a consumer in the group reads messages from the partitions assigned by the coordinator, it must commit the offsets corresponding to the messages it has read. If the consumer crashes or is shutdown, its partitions will be re-assigned to another member, which will begin consumption from the last committed offset of each partition. If the consumer crashes before any offset has been committed, then the consumer which takes over its partitions will use the reset policy.

The offset commit policy is crucial to providing the message delivery guarantees needed by your application. By default, the consumer is configured to use an automatic commit policy, which triggers a commit on a periodic interval. The consumer also supports a commit API which can be used for manual offset management. In the [examples below](https://docs.confluent.io/current/clients/consumer.html#advanced-client-consumer), we show several detailed examples of the commit API and discuss the tradeoffs in terms of performance and reliability.

When writing to an external system, the consumer's position must be coordinated with what is stored as output. That is why the consumer stores its offset in the same place as its output. For example, a [Kafka Connect](https://docs.confluent.io/current/connect/index.html#kafka-connect) connector populates data in HDFS along with the offsets of the data it reads so that it is guaranteed that either data and offsets are both updated, or neither is. A similar pattern is followed for many other data systems that require these stronger semantics, and for which the messages do not have a primary key to allow for deduplication.

This is how Kafka supports [exactly-once processing](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-processing-guarantees) in Kafka Streams, and the transactional producer or consumer can be used generally to provide exactly-once delivery when transferring and processing data between Kafka topics. Otherwise, Kafka guarantees at-least-once delivery by default, and you can implement at-most-once delivery by disabling retries on the producer and committing offsets in the consumer prior to processing a batch of messages.

## Configuration

**Core Configuration:**

The only required setting is **bootstrap.servers**, but you should set a **client.id** since this allows you to easily correlate requests on the broker with the client instance which made it. Typically, all consumers within the same group will share the same client ID in order to enforce [client quotas](https://docs.confluent.io/current/kafka/post-deployment.html#quotas).

**Group Configuration**: You should always configure **group.id** unless you are using the simple assignment API and you don't need to store offsets in Kafka.

You can control the session timeout by overriding the **session.timeout.ms** value. **The default is 30 seconds, but you can safely increase it to avoid excessive rebalances** if you find that your application needs more time to process messages. This concern is mainly relevant if you are using the Java consumer and handling messages in the same thread. In that case, you may also want to adjust **max.poll.records** to tune the number of records that must be handled on every loop iteration. See [basic usage](https://docs.confluent.io/current/clients/consumer.html#consumer-basic-usage) below for more detail on this issue.

**The main drawback to using a larger session timeout is that it will take longer for the coordinator to detect when a consumer instance has crashed, which means it will also take longer for another consumer in the group to take over its partitions.** For normal shutdowns, however, the consumer sends an explicit request to the coordinator to leave the group which triggers an immediate rebalance.

The other setting which affects rebalance behavior is **heartbeat.interval.ms. This controls how often the consumer will send heartbeats to the coordinator.** It is also the way that the consumer detects when a rebalance is needed, so a lower heartbeat interval will generally mean faster rebalances. The default setting is three seconds. For larger groups, it may be wise to increase this setting.

**Offset Management**

The two main settings affecting offset management are whether auto-commit is enabled and the offset reset policy. First, if you set **enable.auto.commit (which is the default**), then the consumer will automatically commit offsets periodically at the interval set by **auto.commit.interval.ms. The default is 5 seconds.**

Second, use **auto.offset.reset** to define the behavior of the consumer when there is no committed position (which would be the case when the group is first initialized) or when an offset is out of range. You can choose either to reset the position to the "**earliest**" offset or the "**latest" offset (the default**). You can also select "none" if you would rather set the initial offset yourself and you are willing to handle out of range errors manually.

## Initialization

The Java consumer is constructed with a standard Properties file.

Properties config = new Properties();

config.put("client.id", InetAddress.getLocalHost().getHostName());

config.put("group.id", "foo");

config.put("bootstrap.servers", "host1:9092,host2:9092");

new KafkaConsumer<K, V>(config);

Configuration errors will result in a KafkaException raised from the constructor of KafkaConsumer.

### **Java Client**

The Java client is designed around an event loop which is driven by the **poll()** API. This design is motivated by the UNIX select and poll system calls. A basic consumption loop with the Java API usually takes the following form:

**while (running) {**

**ConsumerRecords<K, V> records = consumer.poll(Long.MAX\_VALUE);**

**process(records); // application-specific processing**

**consumer.commitSync();**

**}**

There is no background thread in the Java consumer. The API depends on calls to poll() to drive all of its IO including:

* Joining the consumer group and handling partition rebalances.
* Sending periodic heartbeats if part of an active generation.
* Sending periodic offset commits (if autocommit is enabled).
* Sending and receiving fetch requests for assigned partitions.

Due to this single-threaded model, no heartbeats can be sent while the application is handling the records returned from a call to **poll()**. This means that the consumer will fall out of the consumer group if either the event loop terminates or if a delay in record processing causes the session timeout to expire before the next iteration of the loop. This is actually by design. One of the problems that the Java client attempts to solve is ensuring the liveness of consumers in the group. **As long as the consumer is assigned partitions, no other members in the group can consume from the same partitions, so it is important to ensure that it is actually making progress**.

This feature protects your application from a large class of failures, but the downside is that it puts the burden on you to tune the session timeout so that the consumer does not exceed it in its normal record processing. The **max.poll.records** configuration option places an upper bound on the number of records returned from each call. **You should use both poll() and max.poll.records with a fairly high session timeout (e.g. 30 to 60 seconds), and keeping the number of records processed on each iteration bounded so that worst-case behavior is predictable**

**If you fail to tune these settings appropriately, the consequence is typically a CommitFailedException raised from the call to commit offsets for the processed records.** If you are using the automatic commit policy, then you might not even notice when this happens since the consumer silently ignores commit failures internally (unless it's occurring often enough to impact lag metrics). You can catch this exception and either ignore it or perform any needed rollback logic.

**while** **(**running**)** **{**

ConsumerRecords**<**K**,** V**>** records **=** consumer**.**poll**(**Long**.**MAX\_VALUE**);**

process**(**records**);** *// application-specific processing*

**try** **{**

consumer**.**commitSync**();**

**}** **catch** **(**CommitFailedException e**)** **{**

*// application-specific rollback of processed records*

**}**

**}**

## Advanced Examples

### **Basic Poll Loop**

The consumer API is centered around the **poll() method, which is used to retrieve records from the brokers**.

The subscribe()**method controls which topics will be fetched in poll.** Typically, consumer usage involves an initial call to **subscribe()** to setup the topics of interest and then a loop which calls poll() until the application is shutdown.

The consumer intentionally avoids a specific threading model. It is not safe for multi-threaded access and it has no background threads of its own. In particular, this means that all IO occurs in the thread calling **poll()**. In the example below, the poll loop is wrapped in a Runnable which makes it easy to use with an **ExecutorService**.

**public** **abstract** **class** **BasicConsumeLoop** **implements** Runnable **{**

**private** **final** KafkaConsumer**<**K**,** V**>** consumer**;**

**private** **final** List**<**String**>** topics**;**

**private** **final** AtomicBoolean shutdown**;**

**private** **final** CountDownLatch shutdownLatch**;**

**public** **BasicConsumeLoop(**Properties config**,** List**<**String**>** topics**)** **{**

**this.**consumer **=** **new** KafkaConsumer**<>(**config**);**

**this.**topics **=** topics**;**

**this.**shutdown **=** **new** AtomicBoolean**(false);**

**this.**shutdownLatch **=** **new** CountDownLatch**(**1**);**

**}**

**public** **abstract** **void** **process(**ConsumerRecord**<**K**,** V**>** record**);**

**public** **void** **run()** **{**

**try** **{**

consumer**.**subscribe**(**topics**);**

**while** **(!**shutdown**.**get**())** **{**

ConsumerRecords**<**K**,** V**>** records **=** consumer**.**poll**(**500**);**

records**.**forEach**(**record **->** process**(**record**));**

**}**

**}** **finally** **{**

consumer**.**close**();**

shutdownLatch**.**countDown**();**

**}**

**}**

**public** **void** **shutdown()** **throws** InterruptedException **{**

shutdown**.**set**(true);**

shutdownLatch**.**await**();**

**}**

**}**

**We've hard-coded the poll timeout to 500 milliseconds. If no records are received before this timeout expires, then poll() will return an empty record set.** It's not a bad idea to add a shortcut check for this case if your message processing involves any setup overhead.

**To shutdown the consumer, a flag is added which is checked on each loop iteration.**

**After shutdown is triggered, the consumer will wait at most 500 milliseconds (plus the message processing time) before shutting down since it might be triggered while it is in poll(). A better approach is provided in the next example.**

**Note that you should always call close() after you are finished using the consumer.** Doing so will ensure that active sockets are closed and internal state is cleaned up. It will also trigger a group rebalance immediately which ensures that any partitions owned by the consumer are re-assigned to another member in the group. If not closed properly, the broker will trigger the rebalance only after the session timeout has expired. We've added a latch to this example to ensure that the consumer has time to finish closing before finishing shutdown.

### **Shutdown with Wakeup**

An alternative pattern for the **poll loop in the Java consumer is to use Long.MAX\_VALUE** for the timeout. **To break from the loop, you can use the consumer's wakeup() method from a separate thread. This will raise a WakeupException from the thread blocking inpoll(). If the thread is not currently blocking, then this will wakeup the next poll invocation**.

**public** **abstract** **class** **ConsumeLoop** **implements** Runnable **{**

**private** **final** KafkaConsumer**<**K**,** V**>** consumer**;**

**private** **final** List**<**String**>** topics**;**

**private** **final** CountDownLatch shutdownLatch**;**

**public** **BasicConsumeLoop(**KafkaConsumer**<**K**,** V**>** consumer**,** List**<**String**>** topics**)** **{**

**this.**consumer **=** consumer**;**

**this.**topics **=** topics**;**

**this.**shutdownLatch **=** **new** CountDownLatch**(**1**);**

**}**

**public** **abstract** **void** **process(**ConsumerRecord**<**K**,** V**>** record**);**

**public** **void** **run()** **{**

**try** **{**

consumer**.**subscribe**(**topics**);**

**while** **(true)** **{**

ConsumerRecords**<**K**,** V**>** records **=** consumer**.**poll**(**Long**.**MAX\_VALUE**);**

records**.**forEach**(**record **->** process**(**record**));**

**}**

**}** **catch** **(**WakeupException e**)** **{**

*// ignore, we're closing*

**}** **catch** **(**Exception e**)** **{**

log**.**error**(**"Unexpected error"**,** e**);**

**}** **finally** **{**

consumer**.**close**();**

shutdownLatch**.**countDown**();**

**}**

**}**

**public** **void** **shutdown()** **throws** InterruptedException **{**

consumer**.**wakeup**();**

shutdownLatch**.**await**();**

**}**

**}**

### **Synchronous Commits**

The previous examples assumed that the consumer is configured to auto-commit offsets (this is the default). Auto-commit basically works as a cron with a period set through the auto.commit.interval.ms configuration property. If the consumer crashes, then after a restart or a rebalance, the position of all partitions owned by the crashed consumer will be reset to the last committed offset. When this happens, the last committed position may be as old as the auto-commit interval itself. Any messages which have arrived since the last commit will have to be read again.

Clearly if you want to reduce the window for duplicates, you can reduce the auto-commit interval, but some users may want even finer control over offsets. The consumer therefore supports a commit API which gives you full control over offsets. The simplest and most reliable way to manually commit offsets is using a synchronous commit with commitSync(). As its name suggests, this method blocks until the commit has completed successfully.

**private** **void** **doCommitSync()** **{**

**try** **{**

consumer**.**commitSync**();**

**}** **catch** **(**WakeupException e**)** **{**

*// we're shutting down, but finish the commit first and then*

*// rethrow the exception so that the main loop can exit*

doCommitSync**();**

**throw** e**;**

**}** **catch** **(**CommitFailedException e**)** **{**

*// the commit failed with an unrecoverable error. if there is any*

*// internal state which depended on the commit, you can clean it*

*// up here. otherwise it's reasonable to ignore the error and go on*

log**.**debug**(**"Commit failed"**,** e**);**

**}**

**}**

**public** **void** **run()** **{**

**try** **{**

consumer**.**subscribe**(**topics**);**

**while** **(true)** **{**

ConsumerRecords**<**K**,** V**>** records **=** consumer**.**poll**(**Long**.**MAX\_VALUE**);**

records**.**forEach**(**record **->** process**(**record**));**

doCommitSync**();**

**}**

**}** **catch** **(**WakeupException e**)** **{**

*// ignore, we're closing*

**}** **catch** **(**Exception e**)** **{**

log**.**error**(**"Unexpected error"**,** e**);**

**}** **finally** **{**

consumer**.**close**();**

shutdownLatch**.**countDown**();**

**}**

**}**

Copy

In this example, a try/catch block is added around the call to commitSync. The CommitFailedException is thrown when the commit cannot be completed because the group has been rebalanced. This is the main thing to be careful of when using the Java client. Since all network IO (including heartbeating) and message processing is done in the foreground, it is possible for the session timeout to expire while a batch of messages is being processed. To handle this, you have two choices.

### **Asynchronous Commits**

Each call to the commit API results in an offset commit request being sent to the broker. Using the synchronous API, the consumer is blocked until that request returns successfully. This may reduce overall throughput since the consumer might otherwise be able to process records while that commit is pending. One way to deal with this is to increase the amount of data that is returned in each poll(). The consumer has a configuration setting fetch.min.bytes which controls how much data is returned in each fetch. The broker will hold onto the fetch until enough data is available (or fetch.max.wait.ms expires). The tradeoff, however, is that this also increases the amount of duplicates that have to be dealt with in a worst-case failure.

A second option is to use asynchronous commits. Instead of waiting for the request to complete, the consumer can send the request and return immediately.

**public** **void** **run()** **{**

**try** **{**

consumer**.**subscribe**(**topics**);**

**while** **(true)** **{**

ConsumerRecords**<**K**,** V**>** records **=** consumer**.**poll**(**Long**.**MAX\_VALUE**);**

records**.**forEach**(**record **->** process**(**record**));**

consumer**.**commitAsync**();**

**}**

**}** **catch** **(**WakeupException e**)** **{**

*// ignore, we're closing*

**}** **catch** **(**Exception e**)** **{**

log**.**error**(**"Unexpected error"**,** e**);**

**}** **finally** **{**

consumer**.**close**();**

shutdownLatch**.**countDown**();**

**}**

**}**

This quick start follows these steps:

1. Start a Kafka cluster on a single machine.
2. Write example input data to a Kafka topic, using the so-called console producer included in Apache Kafka.
3. Process the input data with a Java application that uses the Kafka Streams library. Here, we will leverage a demo application included in Apache Kafka called [WordCount](https://github.com/apache/kafka/blob/2.1/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java).
4. Inspect the output data of the application, using the so-called console consumer included in Apache Kafka.
5. Stop the Kafka cluster.

## Start the Kafka cluster

In this section we install and start a Kafka cluster on your local machine. This cluster consists of a single-node Kafka cluster (= only one broker) alongside a single-node ZooKeeper ensemble. Later on, we will run the WordCount demo application locally against that cluster. Note that, in production, you'd typically run your Kafka Streams applications on client machines at the perimeter of the Kafka cluster -- they do not run "inside" the Kafka cluster or its brokers.

We begin by starting the ZooKeeper instance, which will listen on localhost:2181. Since this is a long-running service, you should run it in its own terminal.

# Start ZooKeeper. Run this command in its own terminal.

./bin/zookeeper-server-start ./etc/kafka/zookeeper.properties

Next we launch the Kafka broker, which will listen on localhost:9092 and connect to the ZooKeeper instance we just started. Since this is a long-running service, too, you should run it in its own terminal.

# Start Kafka. Run this command in its own terminal

./bin/kafka-server-start ./etc/kafka/server.properties

Now that our single-node Kafka cluster is fully up and running, we can proceed to preparing the input data for our first Kafka Streams experiments.

## Prepare the topics and the input data

In this section we will use built-in CLI tools to manually write some example data to Kafka. In practice, you would rather rely on other means to feed your data into Kafka, for instance via [**Kafka Connect**](https://docs.confluent.io/current/connect/index.html#kafka-connect) if you want to move data from other data systems into Kafka, or via[**Kafka Clients**](https://docs.confluent.io/current/clients/index.html#kafka-clients) from within your own applications.

We will now send some input data to a Kafka topic, which will be subsequently processed by a Kafka Streams application.

First, we need to create the input topic, named **streams-plaintext-input**, and the output topic, named **streams-wordcount-output**

*# Create the input topic*

./bin/kafka-topics --create \

--zookeeper localhost:2181 \

--replication-factor 1 \

--partitions 1 \

--topic streams-plaintext-input

*# Create the output topic*

./bin/kafka-topics --create \

--zookeeper localhost:2181 \

--replication-factor 1 \

--partitions 1 \

--topic streams-wordcount-output

Next, we generate some input data and store it in a local file at /tmp/file-input.txt

The resulting file will have the following contents:

all streams lead to kafka

hello kafka streams

join kafka summit

Lastly, we send this input data to the input topic:

cat /tmp/file-input.txt | ./bin/kafka-console-producer --broker-list localhost:9092 --topic streams-plaintext-input

The Kafka console producer reads the data from STDIN line-by-line, and publishes each line as a separate Kafka message to the topic streams-plaintext-input, where the message key is null and the message value is the respective line such as all streams lead to kafka, encoded as a string.

**This Quick start vs. Stream Data Reality(tm):** You might wonder how this step-by-step quick start compares to a "real" stream data platform, where data is always on the move, at large scale and in realtime. Keep in mind that the purpose of this quick start is to demonstrate, in simple terms, the various facets of an end-to-end data pipeline powered by Kafka and Kafka Streams. For didactic reasons we intentionally split the quick start into clearly separated, sequential steps.

In practice though, these steps will typically look a bit different and noticeably happen in parallel. For example, input data might not be sourced originally from a local file but sent directly from distributed devices, and the data would be flowing continuously into Kafka. Similarly, the stream processing application (see next section) might already be up and running before the first input data is being sent, and so on.

## Process the input data with Kafka Streams

We will run the [WordCount demo application](https://github.com/apache/kafka/blob/2.1/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java), which is included in Apache Kafka. It implements the WordCount algorithm, which computes a word occurrence histogram from an input text. However, unlike other WordCount examples you might have seen before that operate on *finite, bounded data*, the WordCount demo application behaves slightly differently because it is designed to operate on an **infinite, unbounded stream** of input data. Similar to the bounded variant, it is a stateful algorithm that tracks and updates the counts of words. However, since it must assume potentially unbounded input data, it will periodically output its current state and results while continuing to process more data because it cannot know when it has processed "all" the input data. This is a typical difference between the class of algorithms that operate on unbounded streams of data and, say, batch processing algorithms such as Hadoop MapReduce. It will be easier to understand this difference once we inspect the actual output data later on.

Kafka's WordCount demo application is bundled with Confluent Platform, which means we can run it without further ado, i.e. we do not need to compile any Java sources and so on.

The WordCount demo application will read from the input topic streams-plaintext-input, perform the computations of the WordCount algorithm on the input data, and continuously write its current results to the output topic streams-wordcount-output (the names of its input and output topics are hardcoded).

## Inspect the output data

We can now inspect the output of the WordCount demo application by reading from its output topic streams-wordcount-output:

with the following output data being printed to the console:

all 1

streams 1

lead 1

to 1

kafka 1

hello 1

kafka 2

streams 2

join 1

kafka 3

summit 1

Here, the first column is the Kafka message key in **java.lang.String** format, and the second column is the message value in **java.lang.Long** format.

As we discussed above, a streaming word count algorithm continuously computes the latest word counts from the input data, and, in this specific demo application, continuously writes the latest counts of words as its output.

## Stop the Kafka cluster

First, stop the **Kafka broker**

Lastly, stop the **ZooKeeper instance**

## Next steps

* Read the [Kafka Streams Architecture](https://docs.confluent.io/current/streams/architecture.html#streams-architecture) to understand its key concepts and design principles.
* Take a deep dive into the [Kafka Streams Developer Guide](https://docs.confluent.io/current/streams/developer-guide/index.html#streams-developer-guide), which includes many code examples to get you started, as well as the documentation of the [Kafka Streams DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl). This will get you started on writing your own Kafka Streams applications.
* Run through the self-paced [Kafka Streams tutorial for developers](https://docs.confluent.io/current/tutorials/examples/microservices-orders/docs/index.html#tutorial-microservices-orders) to apply the basic principles of streaming applications in an event-drive architecture.

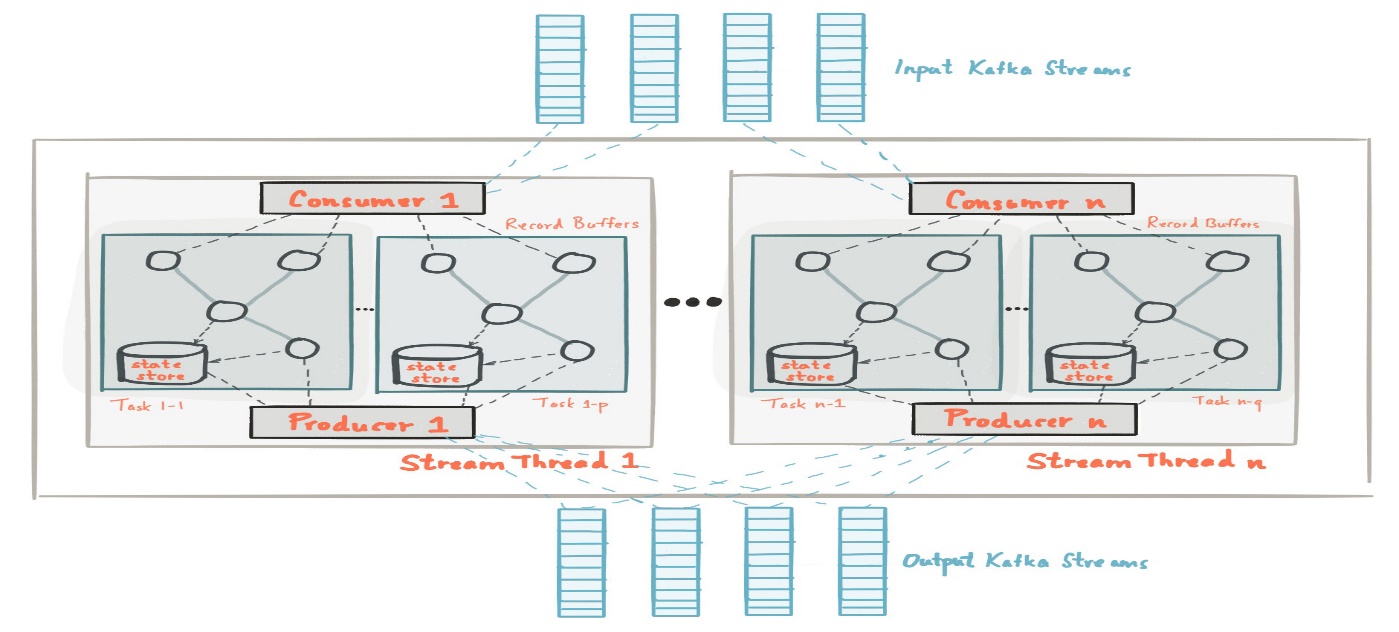
Beyond Kafka Streams, you might be interested in learning more about:

* [Kafka Connect](https://docs.confluent.io/current/connect/index.html#kafka-connect) for moving data between Kafka and other data systems such as Hadoop.
* [Kafka Clients](https://docs.confluent.io/current/clients/index.html#kafka-clients) for reading and writing data from/to Kafka from within your own applications.

# Streams Architecture

Kafka Streams simplifies application development by building on the Kafka producer and consumer APIs, and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity.

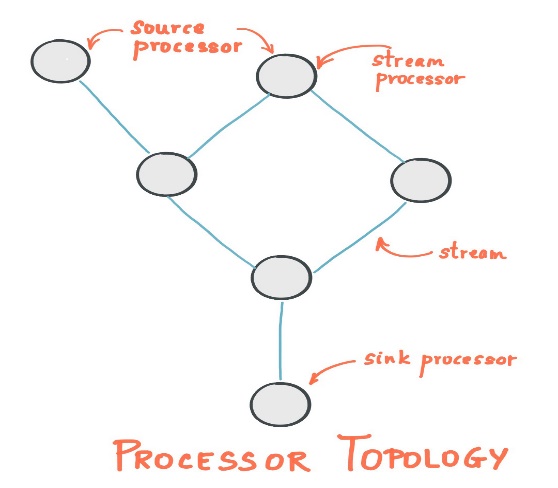
Here is the anatomy of an application that uses the Kafka Streams API. It provides a logical view of a Kafka Streams application that contains multiple stream threads, that each contain multiple stream tasks.



## Processor Topology

A **processor topology** or simply **topology** defines the stream processing computational logic for your application, i.e., how input data is transformed into output data. A topology is a graph of [stream processors](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-processor) (nodes) that are connected by [streams](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-stream) (edges). There are two special processors in the topology:

* **Source Processor**: A source processor is a special type of [stream processor](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-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 forward them to its down-stream processors.
* **Sink Processor**: A 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.

[](https://docs.confluent.io/current/_images/streams-architecture-topology.jpg)

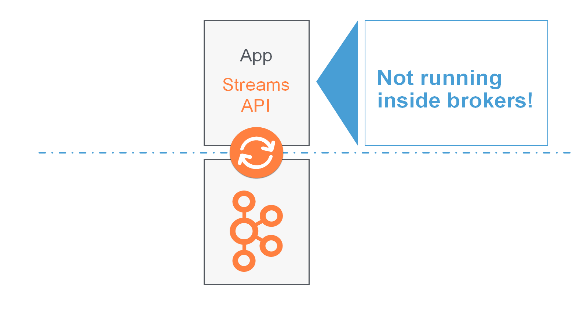
# Streams Concepts

* **The who's who:** Kafka distinguishes **producers**, **consumers**, and **brokers**. In short, producers publish data to Kafka brokers, and consumers read published data from Kafka brokers. Producers and consumers are totally decoupled, and both run outside the Kafka brokers in the perimeter of a Kafka cluster. A Kafka **cluster** consists of one or more brokers. An application that uses the Kafka Streams API acts as both a producer and a consumer.
* **The data:** Data is stored in **topics**. The topic is the most important abstraction provided by Kafka: it is a category or feed name to which data is published by producers. Every topic in Kafka is split into one or more **partitions**. Kafka partitions data for storing, transporting, and replicating it. Kafka Streams partitions data for processing it. In both cases, this partitioning enables elasticity, scalability, high performance, and fault tolerance.
* **Parallelism:** Partitions of Kafka topics, and especially their number for a given topic, are also the main factor that determines the parallelism of Kafka with regards to reading and writing data. Because of the tight integration with Kafka, the parallelism of an application that uses the Kafka Streams API is primarily depending on Kafka's parallelism.

## Stream Processing Application

A stream processing application is any program that makes use of the Kafka Streams library. In practice, this means it is probably "your" application. It may define its computational logic through one or more [processor topologies](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-processor-topology).

Your stream processing application doesn't run inside a broker. Instead, it runs in a separate JVM instance, or in a separate cluster entirely.



## Processor Topology

A **processor topology** or simply **topology** defines the computational logic of the data processing that needs to be performed by a stream processing application. A topology is a graph of stream processors (nodes) that are connected by streams (edges). Developers can define topologies either via the [low-level Processor API](https://docs.confluent.io/current/streams/developer-guide/processor-api.html#streams-developer-guide-processor-api) or via the [Kafka Streams DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl), which builds on top of the former.

## Stream Processor

A **stream processor** is a node in the processor topology as shown in the diagram of section [Processor Topology](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-processor-topology). It represents a processing step in a topology, i.e. it is used to transform data. Standard operations such as [map or filter](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-transformations-stateless), [joins](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-joins), and [aggregations](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-aggregating) are examples of stream processors that are available in Kafka Streams out of the box. A stream processor receives one input record at a time from its upstream processors in the topology, applies its operation to it, and may subsequently produce one or more output records to its downstream processors.

Kafka Streams provides two APIs to define stream processors:

1. The [declarative, functional DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl) is the recommended API for most users -- and notably for starters -- because most data processing use cases can be expressed in just a few lines of DSL code. Here, you typically use built-in operations such as mapand filter.
2. The [imperative, lower-level Processor API](https://docs.confluent.io/current/streams/developer-guide/processor-api.html#streams-developer-guide-processor-api) provides you with even more flexibility than the DSL but at the expense of requiring more manual coding work. Here, you can define and connect custom processors as well as directly interact with [state stores](https://docs.confluent.io/current/streams/architecture.html#streams-architecture-state).

## Stateful Stream Processing

Some stream processing applications don't require state -- they are **stateless** -- which means the processing of a message is independent from the processing of other messages. Examples are when you only need to transform one message at a time, or filter out messages based on some condition.

In practice, however, most applications require state -- they are **stateful** -- in order to work correctly, and this state must be managed in a [fault-tolerant manner](https://docs.confluent.io/current/streams/architecture.html#streams-architecture-fault-tolerance). Your application is stateful whenever, for example, it needs to [join](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-joins), [aggregate](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-aggregations), or [window](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-windowing) its input data. Kafka Streams provides your application with powerful, elastic, highly scalable, and fault-tolerant stateful processing capabilities.

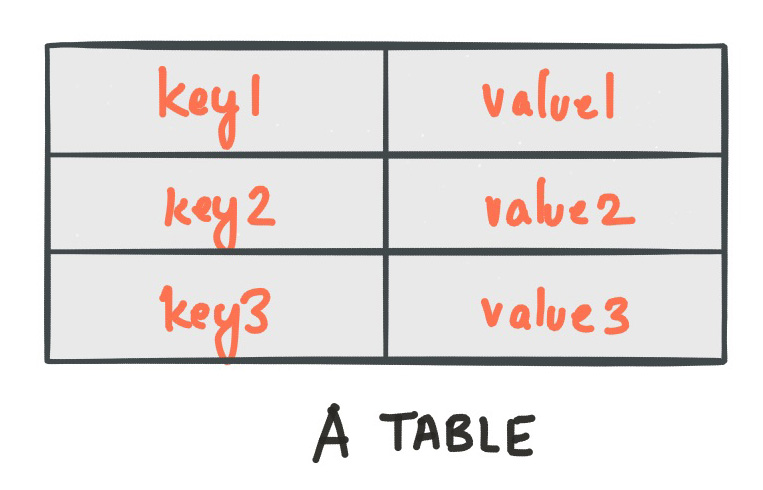
## Duality of Streams and Tables

When implementing stream processing use cases in practice, you typically need both **streams** and also **databases**. An example use case that is very common in practice is an e-commerce application that enriches an incoming stream of customer transactions with the latest customer information from a database table. In other words, streams are everywhere, but databases are everywhere, too.

Any stream processing technology must therefore provide **first-class support for streams and tables**. Kafka's Streams API provides such functionality through its core abstractions for [streams](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-kstream) and [tables](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-ktable), which we will talk about in a minute. Now, an interesting observation is that there is actually a **close relationship between streams and tables**, the so-called [stream-table duality](https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/). And Kafka exploits this duality in many ways: for example, to make your applications [elastic](https://docs.confluent.io/current/streams/developer-guide/running-app.html#streams-developer-guide-execution-scaling), to support [fault-tolerant stateful processing](https://docs.confluent.io/current/streams/developer-guide/processor-api.html#streams-developer-guide-state-store-fault-tolerance), or to run [interactive queries](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-interactive-queries) against your application's latest processing results. And, beyond its internal usage, the Kafka Streams API also allows developers to exploit this duality in their own applications.

Before we discuss concepts such as [aggregations](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-aggregations) in Kafka Streams we must first introduce **tables** in more detail, and talk about the aforementioned stream-table duality. Essentially, this duality means that a stream can be viewed as a table, and a table can be viewed as a stream.

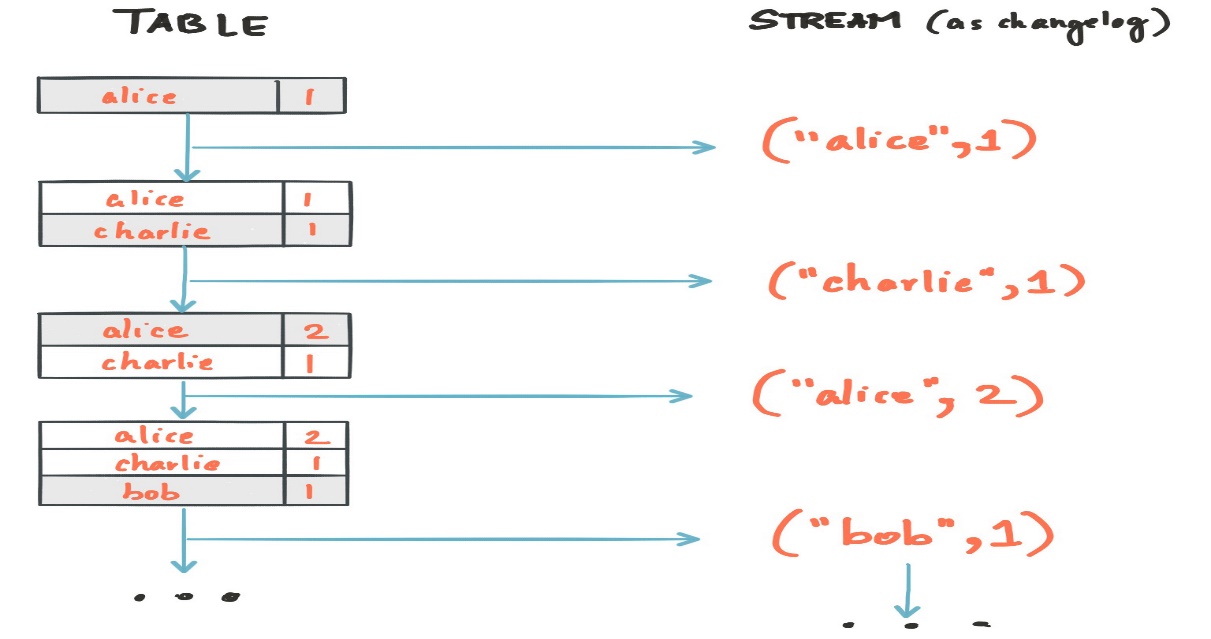
A simple form of a table is a collection of key-value pairs, also called a map or associative array. Such a table may look as follows:

[](https://docs.confluent.io/current/_images/streams-table-duality-01.jpg)

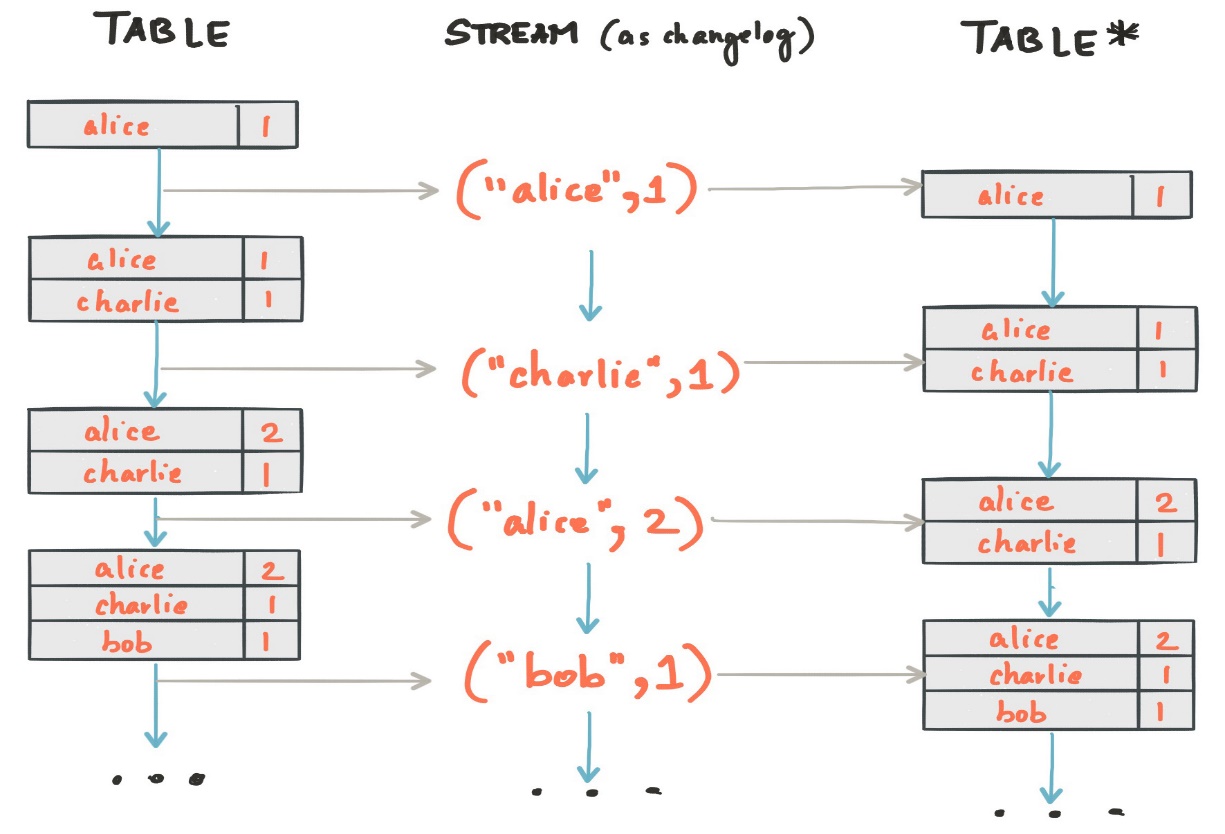
The **stream-table duality** describes the close relationship between streams and tables.

* **Stream as Table:** A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table. A stream is thus a table in disguise, and it can be easily turned into a "real" table by replaying the changelog from beginning to end to reconstruct the table. Similarly, aggregating data records in a stream will return a table. For example, we could compute the total number of pageviews by user from an input stream of pageview events, and the result would be a table, with the table key being the user and the value being the corresponding pageview count.
* **Table as Stream:** A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream (a stream's data records are key-value pairs). A table is thus a stream in disguise, and it can be easily turned into a "real" stream by iterating over each key-value entry in the table.

Let’s illustrate this with an example. Imagine a table that tracks the total number of pageviews by user (first column of diagram below). Over time, whenever a new pageview event is processed, the state of the table is updated accordingly. Here, the state changes between different points in time -- and different revisions of the table -- can be represented as a changelog stream (second column).

[](https://docs.confluent.io/current/_images/streams-table-duality-02.jpg)

Because of the stream-table duality, the same stream can be used to reconstruct the original table (third column):

[](https://docs.confluent.io/current/_images/streams-table-duality-03.jpg)

The same mechanism is used, for example, to replicate databases via change data capture (CDC) and, within Kafka Streams, to replicate its so-called [state stores](https://docs.confluent.io/current/streams/architecture.html#streams-architecture-state) across machines for [fault tolerance](https://docs.confluent.io/current/streams/architecture.html#streams-architecture-fault-tolerance). The stream-table duality is such an important concept for stream processing applications in practice that Kafka Streams models it explicitly via the [KStream](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-kstream) and [KTable](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-ktable) abstractions, which we describe in the next sections.

## KStream

A **KStream** is an abstraction of a **record stream**, where each data record represents a self-contained datum in the unbounded data set. Using the table analogy, data records in a record stream are always interpreted as an "INSERT" -- think: adding more entries to an append-only ledger -- because no record replaces an existing row with the same key. Examples are a credit card transaction, a page view event, or a server log entry.

To illustrate, let's imagine the following two data records are being sent to the stream:

**(**"alice", 1**)** --> **(**"alice", 3**)**

Copy

If your stream processing application were to sum the values per user, it would return 4 for alice. Why? Because the second data record would not be considered an update of the previous record. Compare this behavior of KStream to [KTable](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-ktable) below, which would return 3 for alice.

## KTable

A **KTable** is an abstraction of a **changelog stream**, where each data record represents an update. More precisely, the value in a data record is interpreted as an "UPDATE" of the last value for the same record key, if any (if a corresponding key doesn't exist yet, the update will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT aka INSERT/UPDATE because any existing row with the same key is overwritten. Also, null values are interpreted in a special way: a record with a null value represents a "DELETE" or tombstone for the record's key.

To illustrate, let's imagine the following two data records are being sent to the stream:

**(**"alice", 1**)** --> **(**"alice", 3**)**

Copy

If your stream processing application were to sum the values per user, it would return 3 for alice. Why? Because the second data record would be considered an update of the previous record. Compare this behavior of KTable with the illustration for [KStream](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-kstream) above, which would return 4 for alice.

## GlobalKTable

Like a [KTable](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-ktable), a **GlobalKTable** is an abstraction of a **changelog stream**, where each data record represents an update.

A GlobalKTable differs from a KTable in the data that they are being populated with, i.e. which data from the underlying Kafka topic is being read into the respective table. Slightly simplified, imagine you have an input topic with 5 partitions. In your application, you want to read this topic into a table. Also, you want to run your application across 5 application instances for [maximum parallelism](https://docs.confluent.io/current/streams/architecture.html#streams-architecture-parallelism-model).

* If you read the input topic into a **KTable**, then the "local" KTable instance of each application instance will be populated with data **from only 1 partition** of the topic's 5 partitions.
* If you read the input topic into a **GlobalKTable**, then the local GlobalKTable instance of each application instance will be populated with data **from all partitions of the topic**.

## Aggregations

An **aggregation** operation takes one input stream or table, and yields a new table by combining multiple input records into a single output record. Examples of aggregations are computing counts or sum.

**In the**[**Kafka Streams DSL**](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl)**, an input stream of an**[**aggregation operation**](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-aggregating)**can be a KStream or a KTable, but the output stream will always be a KTable.** This allows Kafka Streams to update an aggregate value upon the late arrival of further records after the value was produced and emitted. When such late arrival happens, the aggregating KStream or KTable emits a new aggregate value. Because the output is a KTable, the new value is considered to overwrite the old value with the same key in subsequent processing steps.

## Joins

A **join** operation merges two input streams and/or tables based on the keys of their data records, and yields a new stream/table.

The [join operations](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-joins) available in the [Kafka Streams DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl) differ based on which kinds of streams and tables are being joined; for example, **KStream-KStream joins versus KStream-KTable joins**.

## Windowing

Windowing lets you control how to group records that have the same key for stateful operations such as [aggregations](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-aggregating) or [joins](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-joins) into so-called windows. Windows are tracked per record key.

[Windowing operations](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-windowing) are available in the [Kafka Streams DSL](https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl). When working with windows, you can specify a **retention period** for the window. This retention period controls how long Kafka Streams will wait for **out-of-order** or **late-arriving** data records for a given window. If a record arrives after the retention period of a window has passed, the record is discarded and will not be processed in that window.

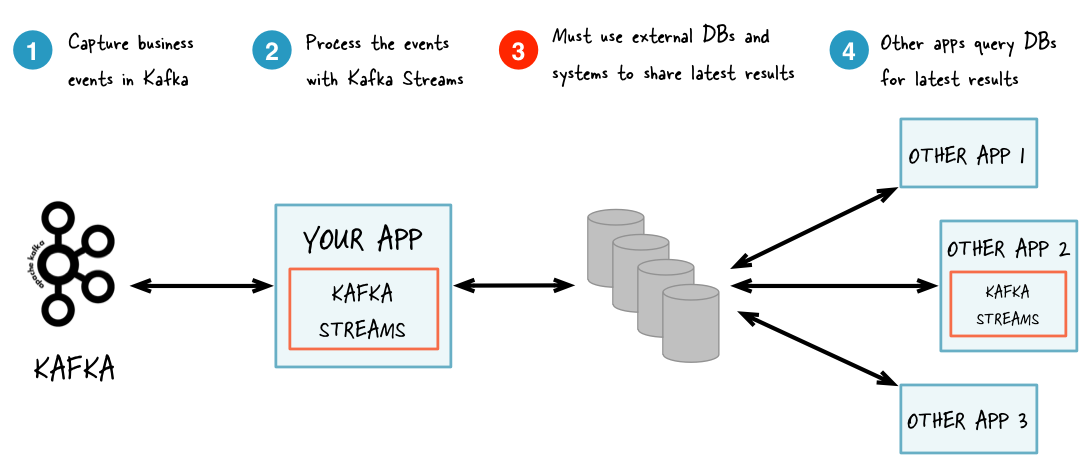
Late-arriving records are always possible in the real world and should be properly accounted for in your applications. It depends on the effective [time semantics](https://docs.confluent.io/current/streams/concepts.html#streams-concepts-time) how late records are handled. In the case of processing-time, the semantics are "when the record is being processed", which means that the notion of late records is not applicable as, by definition, no record can be late. Hence, late-arriving records can only be considered as such (i.e. as arriving "late") for event-time or ingestion-time semantics. In both cases, Kafka Streams is able to properly handle late-arriving records.

## Interactive Queries

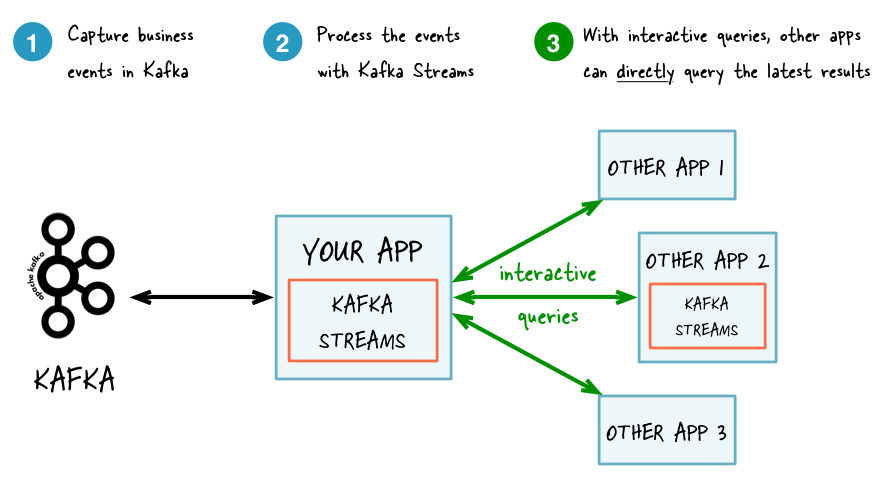
Interactive queries allow you to treat the stream processing layer as a lightweight embedded database, and to directly query the latest state of your stream processing application. You can do this without having to first materialize that state to external databases or external storage.

Interactive queries simplify the architecture and lead to more application-centric architectures.

The following diagram juxtapose two architectures: the first does not use interactive queries whereas the second architecture does. It depends on the concrete use case to determine which of these architectures is a better fit -- the important takeaway is that Kafka Streams and interactive queries give you the flexibility to pick and to compose the right one, rather than limiting you to just a single way.

[](https://docs.confluent.io/current/_images/streams-interactive-queries-01.png)

*Without interactive queries: increased complexity and heavier footprint of architecture.*

[](https://docs.confluent.io/current/_images/streams-interactive-queries-02.png)

*With interactive queries: simplified, more application-centric architecture.*

Here are some use case examples for applications that benefit from interactive queries:

* Real-time monitoring: A front-end dashboard that provides threat intelligence (e.g., web servers currently under attack by cyber criminals) can directly query a Kafka Streams application that continuously generates the relevant information by processing network telemetry data in real-time.
* Video gaming: A Kafka Streams application continuously tracks location updates from players in the gaming universe. A mobile companion app can then directly query the Kafka Streams application to show the current location of a player to friends and family, and invite them to come along. Similarly, the game vendor can use the data to identify unusual hotspots of players, which may indicate a bug or an operational issue.
* **Risk and fraud: A Kafka Streams application continuously analyzes user transactions for anomalies and suspicious behavior. An online banking application can directly query the Kafka Streams application when a user logs in to deny access to those users that have been flagged as suspicious.**
* Trend detection: A Kafka Streams application continuously computes the latest top charts across music genres based on user listening behavior that is collected in real-time. Mobile or desktop applications of a music store can then interactively query for the latest charts while users are browsing the store.

## Processing Guarantees

Kafka Streams supports at-least-once and exactly-once processing guarantees.

**At-least-once semantics**

Records are never lost but may be redelivered. If your stream processing application fails, no data records are lost and fail to be processed, but some data records may be re-read and therefore re-processed. At-least-once semantics is enabled by default (processing.guarantee="at\_least\_once") in your [Streams configuration](https://docs.confluent.io/current/streams/developer-guide/config-streams.html#streams-developer-guide-optional-configs).

**Exactly-once semantics**

Records are processed once. Even if a producer sends a duplicate record, it is written to the broker exactly once. Exactly-once stream processing is the ability to execute a read-process-write operation exactly one time. All of the processing happens exactly once, including the processing and the materialized state created by the processing job that is written back to Kafka. To enable exactly-once semantics, set processing.guarantee="exactly\_once" in your [Streams configuration](https://docs.confluent.io/current/streams/developer-guide/config-streams.html#streams-developer-guide-optional-configs).

## Out-of-Order Handling

Besides the guarantee that each record will be processed exactly-once, another challenge issue that many stream processing application will face is how to handle **out-of-order data** that may impact their business logic. In Kafka Streams, there are two causes that could potentially result in out-of-order data arrivals with respect to their timestamps:

* Within a topic partition, a record's timestamp may not be monotonically increasing along with their offsets. Since Kafka Streams will always try to process records following the offset order, it can cause records with larger timestamps (but smaller offsets) to be processed earlier than records with smaller timestamps (but larger offsets) in the same topic-partition.
* A [stream task](https://docs.confluent.io/current/streams/architecture.html#streams-architecture-tasks) may be processing multiple topic partitions, and if users configure the application to not wait for all partitions to contain some buffered data and pick from the partition with the smallest timestamp to process the next record, then later on when some records are fetched for other topic partitions, their timestamps may be smaller than those processed records, effectively causing older records to be processed after the newer records.