Tutorial: Creating a Streaming Data Pipeline

This quick start provides you with a first hands-on look at the Kafka 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 Apache Kafka®.

This quick start only provides a high-level overview of the Streams API. More details are provided in the rest of the Kafka Streams documentation.

Purpose

This quick start shows how to run theWordCount demo application that is included in 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):

```
// 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));
```

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 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 Kafka called WordCount.
4. Inspect the output data of the application, using the so-called*console consumer* included in 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.

First, you must install **Oracle Java JRE or JDK 1.8** on your local machine.

Second, you must install Confluent Platform 5.3.0 usingZIP and TAR archives. Once installed, change into the installation directory:

```
# *** IMPORTANT STEP ****
# The subsequent paths and commands used throughout this quick start assume that
# your are in the following working directory:
  cd confluent-5.3.0/

# Note: If you want to uninstall the Confluent Platform at the end of this quick start,
# run the following commands.
#
#     rm -rf confluent-5.3.0/
#     rm -rf /tmp/kafka          # Data files of Kafka broker (server)
#     rm -rf /tmp/kafka-streams  # Data files of applications using Kafka's Streams API
#     rm -rf /tmp/zookeeper      # Data files of ZooKeeper
```

Tip

These instructions assume you are installing Confluent Platform by using ZIP or TAR archives. For more information, see On-Premises Deployments.

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

Tip

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 if you want to move data from other data systems into Kafka, or via 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`:

```
echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit" > /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.

Note

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

Now that we have generated some input data, we can run our first Kafka Streams based Java application.

We will run the WordCount demo application, which is included in 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.

```
# Run the WordCount demo application.
# The application writes its results to a Kafka output topic -- there won't be any STDOUT output in your console.
# You can safely ignore any WARN log messages.
  ./bin/kafka-run-class org.apache.kafka.streams.examples.wordcount.WordCountDemo
```

Note

No deployment magic here: The WordCount demo is a normal Java application that can be started and deployed just like any other Java application. The script kafka-run-class is nothing but a simple wrapper for `java -cp ...`.

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). To terminate the demo enter `control-c` from the keyboard.

Inspect the output data

Tip

In this section we will use built-in CLI tools to manually read data from Kafka. In practice, you would rather rely on other means to retrieve data from Kafka, for instance via Kafka Connect if you want to move data from Kafka to other data systems, or via Kafka Clients from within your own applications.

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

```
./bin/kafka-console-consumer --bootstrap-server localhost:9092 \
        --topic streams-wordcount-output \
        --from-beginning \
        --formatter kafka.tools.DefaultMessageFormatter \
        --property print.key=true \
        --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
        --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

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. You can stop the console consumer via `Ctrl-C`.

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. We will talk more about how a stream processing application works in the subsequent chapters of this documentation, where we notably explain the duality between streams and tables in fact, the output we have seen above is actually the changelog stream of a KTable, with the KTable being the result of the aggregation operation performed by the WordCount demo application.

Stop the Kafka cluster

Once you are done with the quick start you can shut down the Kafka cluster in the following order:

1. First, stop the **Kafka broker** by entering `Ctrl-C` in the terminal it is running in. Alternatively, you can `kill` the broker process.
2. Lastly, stop the **ZooKeeper instance** by entering `Ctrl-C` in its respective terminal. Alternatively, you can `kill` the ZooKeeper process.

Congratulations, you have now run your first Kafka Streams applications against data stored in a single-node Kafka cluster, yay!

Next steps

As next steps we would recommend you to:

- Read the Kafka Streams Architecture to understand its key concepts and design principles.
- Take a deep dive into the Kafka Streams Developer Guide, which includes many code examples to get you started, as well as the documentation of the Kafka Streams DSL. This will get you started on writing your own Kafka Streams applications.
- Run through the self-paced Kafka Streams tutorial for developers 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 for moving data between Kafka and other data systems such as Hadoop.
- Kafka Clients for reading and writing data from/to Kafka from within your own applications.

Please report any inaccuracies on this page or suggest an edit.

3 Votes

Last updated on Sep 10, 2019.