**Self-Learning Kafka Streams With Scala (Part 1)**

A few days ago, I came across a situation where I wanted to do a stateful operation on the streaming data. So, I started finding possible solutions for it. I came across many solutions that used many different technologies like Spark structured streaming, Apache Flink, Kafka Streams, etc.

All the solutions solved my problem, but I selected Kafka Streams because it met the majority of my requirements. After that, I started reading its documentation and trying to run its examples. But as soon as I started learning it, I hit a major roadblock in that Kafka Streams does not provide a Scala API! I was shocked to know that.

The reason I was expecting Kafka Streams to have a Scala API is that I was using Scala to build my application, and if Kafka Streams provided an API for it, then it would have been easy for me to include it in my application. But that didn’t turn out to be the case. And on top of that, when I searched for its Scala examples, I was only able to find a handful of them.

So I decided to learn it on my own. My first step was to build a “Hello World!” program using Kafka Streams and Scala, like this:

package com.knoldus.kafka.examples

import java.util.Properties

import org.apache.kafka.common.serialization.\_

import org.apache.kafka.streams.\_

import org.apache.kafka.streams.kstream.KStreamBuilder

/\*\*

\* Copyright Knoldus Software LLP, 2017. All rights reserved.

\*/

object StreamApplication {

def main(args: Array[String]): Unit = {

val config = {

val properties = new Properties()

properties.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "stream-application")

properties.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092")

properties.put(StreamsConfig.DEFAULT\_KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass)

properties.put(StreamsConfig.DEFAULT\_VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass)

properties

}

val builder = new KStreamBuilder()

val sourceStream = builder.stream("SourceTopic")

sourceStream.to("SinkTopic")

val streams = new KafkaStreams(builder, config)

streams.start()

}

}

Before running this example, we need to start the Kafka server. To do that, you can read their [quick start guide](https://kafka.apache.org/quickstart). After that, send some messages to Kafka topic SourceTopic and start a Kafka Consumer for Kafka topic SinkTopic.

$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic SourceTopic

hello world!

Now, run the example and you will see that Kafka consumer topic SinkTopic will receive the message.

$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic SinkTopic

hello world!

This means that now we are able to send messages from one Kafka topic to another via Kafka Streams.

This was my first step in learning Kafka Streams with Scala. I know it's not much, but I still need to explore more in Kafka Streams like transformations, joins, aggregations, etc., about which I will be writing in my future posts. Stay tuned!

The complete code can be downloaded from [GitHub](https://github.com/knoldus/kafka-streams-scala-examples).

# Self-Learning Kafka Streams With Scala (Part 2)

In our previous blog — [Self-Learning Kafka Streams With Scala (Part 1)](https://dzone.com/articles/self-learning-kafka-streams-with-scala-part-1) — we saw how to create a simple KStream in Scala. In this blog, we will see how to transform a KStream and create a new Stream from it.

But before we get into the details of the KStream transformations, let’s take a look at the code:

package com.knoldus.kafka.examples

import java.util.Properties

import org.apache.kafka.common.serialization.\_

import org.apache.kafka.streams.\_

import org.apache.kafka.streams.kstream.{KStreamBuilder, KeyValueMapper}

/\*\*

\* Copyright Knoldus Software LLP, 2017. All rights reserved.

\*/

object MapExample {

def main(args: Array[String]): Unit = {

val config = {

val properties = new Properties()

properties.put(StreamsConfig.APPLICATION\_ID\_CONFIG, "stream-application")

properties.put(StreamsConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092")

properties.put(StreamsConfig.DEFAULT\_KEY\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass)

properties.put(StreamsConfig.DEFAULT\_VALUE\_SERDE\_CLASS\_CONFIG, Serdes.String().getClass)

properties

}

val stringSerde = Serdes.String()

val integerSerde = Serdes.Integer()

val builder = new KStreamBuilder()

val originalStream = builder.stream("SourceTopic")

val mappedStream =

originalStream.map[String, Integer] {

new KeyValueMapper[String, String, KeyValue[String, Integer]] {

override def apply(key: String, value: String): KeyValue[String, Integer] = {

new KeyValue(key, new Integer(value.length))

}

}

}

mappedStream.to(stringSerde, integerSerde, "SinkTopic")

val streams = new KafkaStreams(builder, config)

streams.start()

}

}

Now, there are two major points to be noted down here:

1. **Why are we using anonymous functions of Java in the Scala map function?** The answer lies in [Part 1](https://dzone.com/articles/self-learning-kafka-streams-with-scala-part-1), where we mentioned that Kafka streams do not provide a Scala API, which leaves us with no choice but to use Java 8 anonymous functions.
2. **Here, we are providing serializer/deserializer (SerDe) for SinkTopic explicitly**. Why do we need that? In our [previous blog’s example](https://dzone.com/articles/self-learning-kafka-streams-with-scala-part-1), we didn’t do that. The reason is that we have given the String SerDe in KafkaStreams properties. This leaves Kafka Streams with only one type of SerDe to work with (String), whereas we need an integer SerDe for SinkTopic.

At last, let’s start the Kafka server, run the example, and send some messages:

$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic SourceTopic

hello world!

The result that we will receive is as follows:

$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --property value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer --topic SinkTopic

12

For the consumer, we have to specify the value.deserializer property. Otherwise, we will receive the result in binary format.

So, this is how transformations are written in Kafka Streams with Scala. I hope you liked this explanation and want to learn more about other operations in Kafka Streams like joins, aggregations, etc.

The complete code can be downloaded from [GitHub](https://github.com/knoldus/kafka-streams-scala-examples).

Please feel free to make suggestions or leave a comment!