# Simple Apache Kafka Producer and Consumer using Spring Boot

Sunny Srinidhi Follow
Nov 22, 2018 · 9 min read

> *Also published here: https://blog.contactsunny.com/tech/simple-apache-kafka-producer-and-consumer-using-spring-boot*

Before I even start talking about Apache Kafka here, let me answer your question after you read the topic—aren't there enough posts and guides about this topic already? Yes, there are plenty of reference documents and how-to posts about how to create Kafka producers and consumers in a Spring Boot application.

Then why am I writing another post about this? Well, in the future, I'll be talking about some advanced stuff, in the data science space. Apache Kafka is one of the most used technologies and tools in this space. It kind of becomes important to know how to work with Apache Kafka in a real-world application. So this is an introductory post to the technology, which we'll be referring to in the future. If you're familiar with this already, you can probably skip this and check out my other posts, which I've published after this one.

. . .

Now that we got that out of the way, let's start looking at the fun part. I'm not going to tell you what's Apache Kafka or what's Spring Boot. Because if you're reading this, I guess you already know what these are. If not, and if you want me to write introductory posts for these technologies, let me know, and I shall.

In this post, we'll see how to create a Kafka producer and a Kafka consumer in a Spring Boot application using a very simple method. Now, I agree that there's an even easier method to create a producer and a consumer in Spring Boot (using annotations), but you'll soon realise that it'll not work well for most cases. The code I'm presenting here is being used by us in various production systems, and it works without any issues (for the most part).

## The Dependencies

In this example, I'm working with a Spring Boot application which is created as a Maven project. So we declare all our dependencies in the *pom.xml* file. Copy and paste the following inside the <*dependencies*> block in your *pom.xml* file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>0.11.0.0</version>
</dependency>


<dependency>
    <groupId>org.codehaus.jettison</groupId>
    <artifactId>jettison</artifactId>
    <version>1.2</version>
</dependency>
```

## The Properties File

Once you we have dependencies in place, let's move on to the properties file. In the *resources* directory, create a file named *application.properties*. You can even create a *.yml* file, but for this post, we'll keep it simple. Once you have created the file, add the following into it. Make sure you change the host names and the Kafka topic to match your setup.

```
zookeeper.host=localhost:2181
zookeeper.groupId=thetechcheck
```

```
kafka.topic.thetechcheck=thetechcheck

kafka.bootstrap.servers=localhost:9092
```

## The Kafka Producer

Before we create a Kafka producer object, we have to set a few configuration items, which we'll have to pass to the producer object. As you might have guessed by now, we'll need to read some of the configuration from the *application.properties* file. We can easily do this in our Java class using the *@Value* annotation.

For the producer configuration, we only need the bootstrap servers configuration. To read this from the properties file, you can use the code snippet below. Put that inside the class which has the main() method. Or wherever you plan to create your producer object.

```
@Value("${kafka.bootstrap.servers}")
private String kafkaBootstrapServers;
```

> *There are many ways and places in the code where you can create these producer and consumer objects. For now, I'm keeping it very simple. But you can be assured that this code is production-ready.*

Once we have our bootstrap servers configuration in a String variable, we can create the producer object configuration using an instance of the *java.util.Properties* class, using the code snippet given below.

```
/*
 * Defining producer properties.
 */
Properties producerProperties = new Properties();
producerProperties.put("bootstrap.servers",
kafkaBootstrapServers);
producerProperties.put("acks", "all");
producerProperties.put("retries", 0);
producerProperties.put("batch.size", 16384);
producerProperties.put("linger.ms", 1);
producerProperties.put("buffer.memory", 33554432);
producerProperties.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
producerProperties.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
```

As you can see from the code snippet, we have hard-coded a few configuration items. We can use those values for now, without any worries.

Now that we have the configuration setup, let's create a producer object. You can use the code snippet below to do that.

```
/*
Creating a Kafka Producer object with the configuration
above.
 */
KafkaProducer<String, String> producer = new KafkaProducer<>
(producerProperties);
```

Next step is to write a function which will send our messages to the Kafka topic. The code for this is very simple. The KafkaProducer class gives us a method which takes care of all the heavy lifting. We just have to pass the topic to which we want to send the message along with the message itself. You can use the function below to send messages to the topic.

```
/**
 * Function to send a message to Kafka
 * @param payload The String message that we wish to send to
the Kafka topic
 * @param producer The KafkaProducer object
 * @param topic The topic to which we want to send the
message
 */
private static void sendKafkaMessage(String payload,
         KafkaProducer<String, String> producer,
         String topic)
{
    logger.info("Sending Kafka message: " + payload);
    producer.send(new ProducerRecord<>(topic, payload));
}
```

As you can see from the code snippet above, we are logging the message that we're sending. For this, we're using a *logger* object, which is an instance of the *org.apache.log4j.Logger* class. You can create this logger object as follows:

```
private static final Logger logger =
Logger.getLogger(SimpleKafkaProducerApplication.class);
```

## Sending A Message To The Kafka Topic

We have everything ready to start sending messages to Kafka. All we need is to call the the method we wrote above. To test it out, we'll send a few simple String messages in loop. A simple for loop will do, like what we have below.

```
/*
Creating a loop which iterates 10 times, from 0 to 9, and
sending a
simple message to Kafka.
 */
for (int index = 0; index < 10; index++) {
    sendKafkaMessage("The index is now: " + index, producer,
theTechCheckTopicName);
}
```

The *theTechCheckTopicName* variable is created using the configuration we made in the properties file. We can create this variable similar to how we created the bootstrap servers variable earlier:

```
@Value("${kafka.topic.thetechcheck}")
private String theTechCheckTopicName;
```

That's pretty much it, we now have successfully sent messages to an Apache Kafka topic using a Spring Boot application. You can open up a console consumer and check if you have got those messages into Kafka.

But in most real-word applications, you won't be exchanging simple Strings between Kafka producers and consumers. You will mostly be exchanging JSON objects, in a serialized fashion of course (Kafka only supports String messages). So how do you send JSON objects to Kafka? I have an example for that as well.

In the code snippet below, we are using the same for loop that we used earlier, but instead of sending Strings, we are creating JSON objects, adding some fields into those objects (even nested JSON objects), serializing those objects, and then sending them to Kafka. You can just copy and paste the code below and it should work.

```
/*
Creating a loop which iterates 10 times, from 0 to 9, and
creates an instance of JSONObject in each iteration. We'll
use this simple JSON object to illustrate how we can send a
JSON object as a message in Kafka.
 */
for (int index = 0; index < 10; index++) {

    /*
    We'll create a JSON object which will have a bunch of
fields, and another JSON object, which will be nested inside
the first JSON object. This is just to demonstrate how
complex objects could be serialized and sent to topics in
Kafka.
     */
    JSONObject jsonObject = new JSONObject();
    JSONObject nestedJsonObject = new JSONObject();

    try {
        /*
        Adding some random data into the JSON object.
         */
        jsonObject.put("index", index);
        jsonObject.put("message", "The index is now: " +
index);
```

```
        /*
        We're adding a field in the nested JSON object.
         */
        nestedJsonObject.put("nestedObjectMessage", "This is
a nested JSON object with index: " + index);

        /*
        Adding the nexted JSON object to the main JSON
object.
         */
        jsonObject.put("nestedJsonObject",
nestedJsonObject);

    } catch (JSONException e) {
        logger.error(e.getMessage());
    }

    /*
    We'll now serialize the JSON object we created above,
and send it to the same topic in Kafka, using the same
function we used earlier.
    You can use any JSON library for this, just make sure it
serializes your objects properly.
    A popular alternative to the one I've used is Gson.
     */
    sendKafkaMessage(jsonObject.toString(), producer,
theTechCheckTopicName);
}
```

## The Kafka Consumer

We're done with producing messages. We've had enough of it. Now, let's start consuming those messages. Creating a Kafka consumer is a bit more complex compared to how we created a producer. This is because, after creating the configuration, we have to start the consumer in a thread. This is so that we have a process running forever and listening to all the messages coming in to the topic. But if you take out the thread part, the consumer is pretty easy.

We'll start with creating the consumer properties, like below.

```
/*
 * Defining Kafka consumer properties.
 */
Properties consumerProperties = new Properties();
consumerProperties.put("bootstrap.servers",
kafkaBootstrapServers);
consumerProperties.put("group.id", zookeeperGroupId);
consumerProperties.put("zookeeper.session.timeout.ms",
"6000");
consumerProperties.put("zookeeper.sync.time.ms","2000");
consumerProperties.put("auto.commit.enable", "false");
consumerProperties.put("auto.commit.interval.ms", "1000");
consumerProperties.put("consumer.timeout.ms", "-1");
consumerProperties.put("max.poll.records", "1");
consumerProperties.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
consumerProperties.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
```

The *zookeeperGroupId* variable is created as follows:

```
@Value("${zookeeper.groupId}")
private String zookeeperGroupId;
```

We'll now create a thread, inside which we'll create an instance of a custom class, and start a single worker of that class. We'll then start the thread so that we have the consumer ready.

```
/*
 * Creating a thread to listen to the kafka topic
 */
Thread kafkaConsumerThread = new Thread(() -> {
    logger.info("Starting Kafka consumer thread.");

    SimpleKafkaConsumer simpleKafkaConsumer = new
SimpleKafkaConsumer(
            theTechCheckTopicName,
            consumerProperties
    );

    simpleKafkaConsumer.runSingleWorker();
});

/*
 * Starting the first thread.
 */
kafkaConsumerThread.start();
```

We create the KafkaConsumer object in the constructor of the class SimpleKafkaConsumer that we have written. The constructor looks something like this:

```
private KafkaConsumer<String, String> kafkaConsumer;

public SimpleKafkaConsumer(String theTechCheckTopicName,
Properties consumerProperties) {

    kafkaConsumer = new KafkaConsumer<>(consumerProperties);

kafkaConsumer.subscribe(Arrays.asList(theTechCheckTopicName)
);
}
```

As you can see, we have subscribed to one topic, which we passed as a constructor parameter. After we create the object, we call the runSingleWorker() method on the object. This is where we actually receive messages from Kafka and process them. This method might look a bit intimidating, but I've included comments in the code to make it easier to understand.

```
/**
 * This function will start a single worker thread per
topic.
 * After creating the consumer object, we subscribed to a
list of Kafka topics, in the constructor.
 * For this example, the list consists of only one topic.
But you can give it a try with multiple topics.
 */
public void runSingleWorker() {

    /*
     * We will start an infinite while loop, inside which
we'll be listening to
     * new messages in each topic that we've subscribed to.
     */
    while(true) {

        ConsumerRecords<String, String> records =
kafkaConsumer.poll(100);

        for (ConsumerRecord<String, String> record :
records) {

            /*
            Whenever there's a new message in the Kafka
topic, we'll get the message in this loop, as the record
object.
```

```
            */

           /*
           Getting the message as a string from the record
object.
            */
           String message = record.value();

           /*
           Logging the received message to the console.
            */
           logger.info("Received message: " + message);

           /*
           If you remember, we sent 10 messages to this
topic as plain strings. 10 other messages were serialized
JSON objects. Now we'll deserialize them here. But since we
can't make out which message is a serialized JSON object and
which isn't, we'll try to deserialize all of them. So,
obviously, we'll get an exception for the first 10 messages
we receive. We'll just log the errors and not worry about
them.
            */
           try {
               JSONObject receivedJsonObject = new
JSONObject(message);

               /*
               To make sure we successfully deserialized
the message to a JSON object, we'll log the index of JSON
object.
                */
               logger.info("Index of deserialized JSON
object: " + receivedJsonObject.getInt("index"));
           } catch (JSONException e) {
               logger.error(e.getMessage());
           }

           /*
           Once we finish processing a Kafka message, we
have to commit the offset so that we don't end up consuming
the same message endlessly. By default, the consumer object
takes care of this. But to demonstrate how it can be done,
we have turned this default behaviour off, instead, we're
```

```
going to manually commit the offsets.
            The code for this is below. It's pretty much
self explanatory.
             */
            {
                Map<TopicPartition, OffsetAndMetadata>
commitMessage = new HashMap<>();

                commitMessage.put(new
TopicPartition(record.topic(), record.partition()),
                        new
OffsetAndMetadata(record.offset() + 1));

                kafkaConsumer.commitSync(commitMessage);

                logger.info("Offset committed to Kafka.");
            }
        }
    }
}
```

That's all. You have successfully created a Kafka producer, sent some messages to Kafka, and read those messages by creating a Kafka consumer. You can refer to the project from which I've take code snippets for this post on my GitHub page. And if you have any doubt about how this works, or need any clarification with the code, or have any suggestions for how I can improve this code, please leave comments below and we can have a chat.