Kafka Producers: Writing Messages to Kafka

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by creating a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles.

For example, in a credit card transaction processing system, there will be a client application, perhaps an online store, responsible for sending each transaction to Kafka immediately when a payment is made. Another application is responsible for immediately checking this transaction against a rules engine and determining

whether the transaction is approved or denied. The approve/deny response can then be written back to Kafka, and the response can propagate back to the online store where the transaction was initiated. A third application can read both transactions and the approval status from Kafka and store them in a database where analysts can later review the decisions and perhaps improve the rules engine.

**Producer Overview**

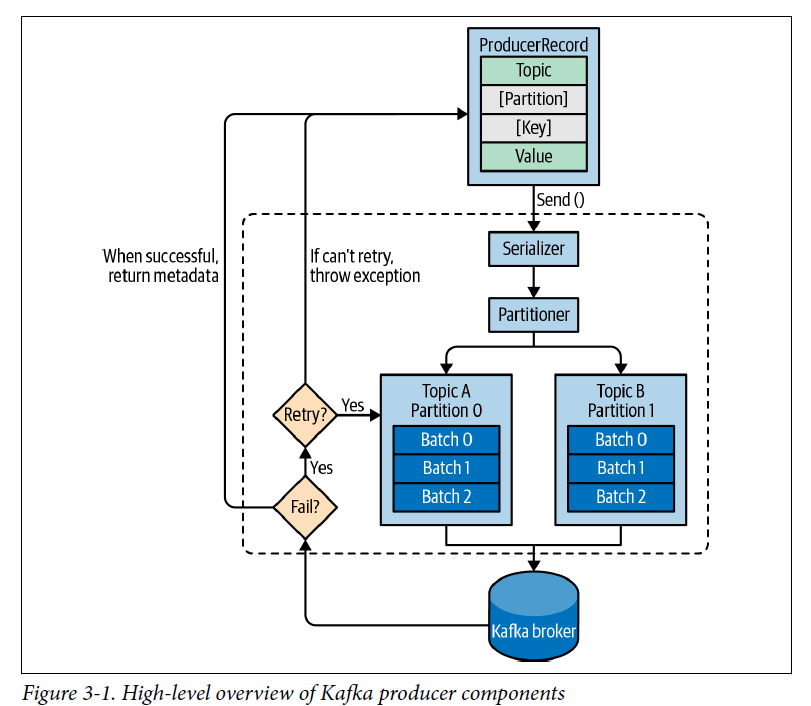
There are many reasons an application might need to write messages to Kafka: recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, communicating asynchronously with other applications, buffering information before writing to a database, and much more.

Consider a scenario where we store click information from a website. In that case, some message loss or a few duplicates can be tolerated; latency can be high as long as there is no impact on the user experience. In other words, we don’t mind if it takes a few seconds for the message to arrive at Kafka, as long as the next page loads immediately after the user clicks on a link. Throughput will depend on the level of activity we

anticipate on our website.

The different requirements will influence the way you use the producer API to write messages to Kafka and the configuration you use. While the producer API is very simple, there is a bit more that goes on under the

hood of the producer when we send data. Figure 3-1 shows the main steps involved in sending data to Kafka.



We start producing messages to Kafka by creating a **ProducerRecord**, which must include the topic we want to send the record to and a value. Optionally, we can also specify a **key, a partition, a timestamp, and/or a collection of headers**. Once we send the **ProducerRecord**, the first thing the producer will do is **serialize** the key and value objects to byte arrays so they can be **sent over the network**.

Next, if we didn’t explicitly specify a partition, the data is sent to a **partitioner**. The partitioner will choose a partition for us, usually based on the **ProducerRecord** key. Once a partition is selected, the producer knows which topic and partition the record will go to. It then adds the record to a batch of records that will also be sent to the same topic and partition. A separate thread is responsible for sending those batches of records to the appropriate Kafka brokers.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a **RecordMetadata** object with the topic, partition, and the offset of the record within the partition. If the broker failed to write the messages, it will return an error. When the producer receives an error, it may retry sending the message a few more times before giving up and returning an error.

**Construction of Kafka Producer**

The first step in writing messages to Kafka is to create a producer object with the properties you want to pass to the producer. A Kafka producer has three mandatory properties

1. **bootstrap.servers**

List of host: port pairs of brokers that the producer will use to establish initial connection to the Kafka cluster. This list doesn’t need to include all brokers, since the producer will get more information after the initial connection. But it is recommended to include at least two, so in case one broker goes down, the producer will still be able to connect to the cluster.

1. **key.serializer**

Name of a class that will be used to serialize the keys of the records we will produce to Kafka. Kafka brokers expect byte arrays as keys and values of messages. However, the producer interface allows, using parameterized types, any Java object to be sent as a key and value. This makes for very readable code, but it also means that the producer has to know how to convert these objects to byte arrays. key.serializer should be set to a name of a class that implements the org.apache.kafka.common.serialization.Serializer interface. The producer will use this class to serialize the key object to a byte array.

1. **value.seriallizer**

Name of a class that will be used to serialize the values of the records we will produce to Kafka. The same way you set key.serializer to a name of a class that will serialize the message key object to a byte array, you set value.serializer to a class that will serialize the message value object.

The following code snippet shows how to create a new producer by setting just the mandatory parameters and using defaults for everything else:

Properties kafkaProps = new Properties();

kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

kafkaProps.put("value.serializer","org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps);

Once we instantiate a producer, it is time to start sending messages. There are three primary methods of sending messages:

1. **Fire and Forget:**

We send a message to the server and don’t really care if it arrives successfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, in case of nonretriable errors or timeout, messages will get lost and the application will not get any information or exceptions about this.

1. **Asynchronous send**

We call the send() method with a callback function, which gets triggered when it

receives a response from the Kafka broker.

1. **Synchronous send**

Technically, Kafka producer is always asynchronous—we send a message and the send() method returns a Future object. However, we use get() to wait on the Future and see if the send() was successful or not before sending the next record.

The simplest way to send a message is as follows:

ProducerRecord<String, String> record =

new ProducerRecord<>("CustomerCountry", "Precision Products",

"France");

try {

producer.send(record);

} catch (Exception e) {

e.printStackTrace();

}

The producer accepts ProducerRecord objects, so we start by creating one. ProducerRecord has multiple constructors, which we will discuss later. Here we use one that requires the name of the topic we are sending data to, which is always a string, and the key and value we are sending to Kafka, which in this case are also strings. The types of the key and value must match our key serializer and value serializer objects.

**Send message synchronously**

ProducerRecord<String, String> record =

new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

try {

producer.send(record).get();

} catch (Exception e) {

e.printStackTrace();

}

Here, we are using Future.get() to wait for a reply from Kafka. This method will throw an exception if the record is not sent successfully to Kafka. If there were no errors, we will get a RecordMetadata object that we can use to retrieve the offset the message was written to and other metadata.

**Send messages Asynchronously.**

To send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record. Here is an example of how we use a callback:

private class DemoProducerCallback implements Callback {

@Override

public void onCompletion(RecordMetadata recordMetadata, Exception e) {

if (e != null) {

e.printStackTrace();

}

}

}

ProducerRecord<String, String> record =

new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");

producer.send(record, new DemoProducerCallback());

1. To use callbacks, you need a class that implements the org.apache.kafka.
2. clients.producer.Callback interface, which has a single function—onCompletion().
3. If Kafka returned an error, onCompletion() will have a nonnull exception. Here we “handle” it by printing, but production code will probably have more robust
4. error handling functions.
5. The records are the same as before. And we pass a Callback object along when sending the record.

**Headers**

Records can, in addition to key and value, also include headers. Record headers give you the ability to add some metadata about the Kafka record, without adding any extra information to the key/value pair of the record itself. Headers are often used for lineage to indicate the source of the data in the record, and for routing or tracing messages based on header information without having to parse the message itself (perhaps the message is encrypted and the router doesn’t have permissions to access the data). Headers are implemented as an ordered collection of key/value pairs. The keys are always a String, and the values can be any serialized object—just like the message value.

Here is a small example that shows how to add headers to a ProduceRecord:

ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

record.headers().add("privacy-level","YOLO".getBytes(StandardCharsets.UTF\_8));

**Interceptors**

There are times when you want to modify the behavior of your Kafka client application without modifying its code, perhaps because you want to add identical behavior to all applications in the organization. Or perhaps you don’t have access to the original code.

Kafka’s ProducerInterceptor interceptor includes two key methods:

ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)

This method will be called before the produced record is sent to Kafka, indeed before it is even serialized. When overriding this method, you can capture information about the sent record and even modify it. Just be sure to return a valid ProducerRecord from this method. The record that this method returns will be

serialized and sent to Kafka.

void onAcknowledgement(RecordMetadata metadata, Exception exception)

This method will be called if and when Kafka responds with an acknowledgment for a send. The method does not allow modifying the response from Kafka, but you can capture information about the response.

Common use cases for producer interceptors include capturing monitoring and tracing information; enhancing the message with standard headers, especially for lineage tracking purposes; and redacting sensitive information.

As we mentioned earlier, producer interceptors can be applied without any changes to the client code. To use the preceding interceptor with kafka-console-producer, an example application that ships with Apache Kafka, follow these three simple steps:

1. Add your jar to the classpath:

export CLASSPATH=$CLASSPATH:~./target/CountProducerInterceptor-1.0-SNAPSHOT.jar

1. Create a config file that includes:

interceptor.classes=com.shapira.examples.interceptors.CountProducer

Interceptor counting.interceptor.window.size.ms=10000

1. Run the application as you normally would, but make sure to include the configuration

that you created in the previous step:

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

interceptor-test --producer.config producer.config