Chapter 4. Kafka Consumers - Reading Data from Kafka

Applications that need to read data from Kafka use a KafkaConsumer to subscribe to Kafka topics and receive messages from these topics. Reading data from Kafka is a bit different than reading data from other messaging systems and there are few unique concepts and ideas involved. It is difficult to understand how to use the consumer API without understanding these concepts first. So we'll start by explaining some of the important concepts, and then we'll go through some examples that show the different ways the consumer APIs can be used to implement applications with different requirements.

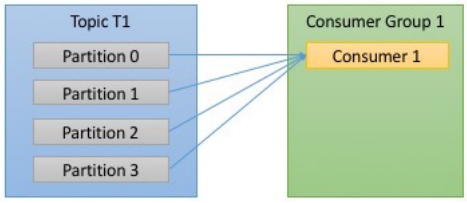
KafkaConsumer Concepts

Consumers and Consumer Groups

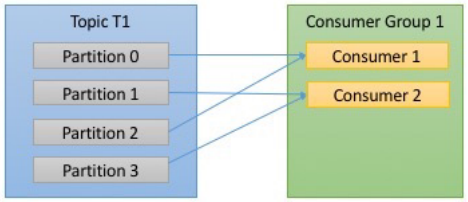
Suppose you have an application that needs to read messages from a Kafka topic, run some validations against them and write the results to another data store. In this case your application will create a consumer object, subscribe to the appropriate topic and start receiving messages, validating them and writing the results. This can work well for a while, but what if the rate at which producers write messages to the topic exceed the rate at which your application can validate them? If you are limited to a single consumer reading and processing the data, your application may fall farther and farther behind, unable to keep up with the rate of incoming messages. Obviously there is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data between them.

Kafka consumers are typically part of a `consumer group`. When multiple consumers are subscribed to a topic and belong to the same consumer group, then each consumer in the group will receive messages from a different subset of the partitions in the topic.

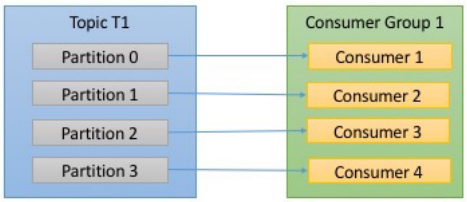
Lets take topic *t1* with 4 partitions. Now suppose we created a new consumer, *c1*, which is the only consumer in group *g1* and use it to subscribe to topic *t1*. Consumer c1 will get all messages from all four of t1 partitions.



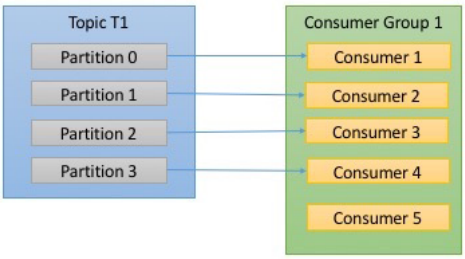
If we add another consumer, *c2* to group *g1*, each consumer will only get messages from two partitions. Perhaps messages from partition 0 and 2 go to *c1* and messages from partitions 1 and 3 go to consumer *c2*.



If *g1* has 4 consumers, then each will read messages from a single partition.



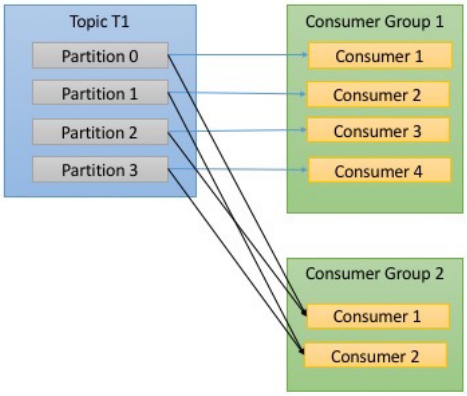
If we add more consumers to a single group with a single topic than we have partitions, than some of the consumers will be idle and get no messages at all.



The main way we scale consumption of data from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high latency operations such as write to a database or to HDFS, or a time-consuming computation on the data. In these cases, a single consumer can't possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. This is a good reason to create topics with a large number of partitions - it allows adding more consumers when the load increases. Keep in mind that there is no point in adding more consumers than you have partitions in a topic - some of the consumers will just be idle. Chapter 2 includes some suggestions on how to choose the number of partitions in a topic.

In addition to adding consumers in order to scale a single application, it is very common to have multiple applications that need to read data from the same topic. In fact, one of the main design goals in Kafka was to make the data produced to Kafka topics available for many use-cases throughout the organization. In those cases, we want each application to get all of the messages, rather than just a subset. To make sure an application gets all the messages in a topic, you make sure the application has its own consumer group. Unlike many traditional messaging systems, Kafka scales to large number of consumers and consumer groups without reducing performance.

In the example above, if we add a new consumer group *g2* with a single consumer, this consumer will get all the messages in topic *t1* independently of what *g1* is doing. *g2* can have more than a single consumer, in which case they will each get a subset of partitions, just like we showed for *g1*, but *g2* as a whole will still get all the messages regardless of other consumer groups.



To summarize, you create a new consumer group for each application that needs all the messages from one or more topics. You add consumers to an existing consumer group to scale the reading and processing of messages from the topics, each additional consumer in a group will only get a subset of the messages.

Consumer Groups - Partition Rebalance

As we've seen in the previous section, consumers in a consumer group share ownership of the partitions in the topics they subscribe to. When we add a new consumer to the group it starts consuming messages from partitions which were previously consumed by another consumer. The same thing happens when a consumer shuts down or crashes, it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. Reassignment of partitions to consumers also happen when the topics the consumer group is consuming are modified, for example if an administrator adds new partitions.

The event in which partition ownership is moved from one consumer to another is called a *rebalance*. Rebalances are important since they provide the consumer group with both high-availability and scalability (allowing us to easily and safely add and remove consumers), but in the normal course of events they are fairly undesirable. During a rebalance, consumers can't consume messaged, so a rebalance is in effect a short window of unavailability on the entire consumer group. In addition, when partitions are moved from one consumer to another the consumer loses its current state, if it was caching any data, it will need to refresh its caches - slowing down our application until the consumer sets up its state again. Throughout this chapter we will discuss how to safely handle rebalances and how to avoid unnecessary rebalances.

The way consumers maintain their membership in a consumer group and their ownership on the partitions assigned to them is by sending *heartbeats* to a Kafka broker designated as the *Group Coordinator* (this broker can be different for different consumer groups). As long the consumer is sending heartbeats in regular intervals, it is assumed to be alive, well and processing messages from its partitions. Heartbeats are sent when the consumer polls (i.e. retrieves records) and when it commits records it has consumed. If the consumer stops sending heartbeats for long enough, its session will time out and the group coordinator will consider it dead and trigger a rebalance. If a consumer crashed and stopped processing messages, it will take the group coordinator few seconds without heartbeats to decide it is dead and trigger the rebalance. During those seconds, no messages will be processed from the partitions owned by the dead consumer. When closing a consumer cleanly, the consumer will notify the group coordinator that it is leaving, and the group coordinator will trigger a rebalance immediately, reducing the gap in processing. Later in this chapter we will discuss configuration options that control heartbeat frequency and session timeouts and how to set those to match your requirements.

In release 0.10.1, the Kafka community introduced a separate heartbeat thread which will send heartbeats in-between polls as well. This allows you to separate the heartbeat frequency (and therefore how long it takes for the consumer group to detect that a consumer crashed and is no longer sending heartbeats) from the frequency of polling (which is determined by the time it takes to process the data returned from the brokers). With newer versions of Kafka, you can configure how long the application can go without polling before it will leave the group and trigger a rebalance. This is to prevent a **livelock**, where the application didn't crash but is failing to make progress for some reason. This configuration is separate from `session.timeout.ms` which controls the time it takes to detect a consumer crashed and stopped sending heartbeats.

The rest of the chapter will discuss some of the challenges with the older behavior and how the programmer can handle them. If you are running Apache Kafka 0.10.1 or newer, some of the discussion regarding long-running processing are less relevant for you - you still need to keep polling to maintain group membership and stay alive, but you can increase `max.poll.interval.ms` to handle longer delays between polling without concerns that this will delay crash detection.

HOW DOES THE PROCESS OF ASSIGNING PARTITIONS TO BROKERS WORK?

When a consumer wants to join a group, it sends a JoinGroup request to the group coordinator. The first consumer to join the group becomes the group *leader*. The leader receives a list of all consumers in the group from the group coordinator (this will include all consumers that sent a heartbeat recently and are therefore considered alive) and it is responsible for assigning a subset of partitions to each consumer. It uses an implementation of PartitionAssignor interface to decide which partitions should be handled by which consumer. Kafka has two built-in partition assignment policies, which we will discuss in more depth in the configuration section. After deciding on the partition assignment, the consumer leader sends the list of assignments to the GroupCoordinator which sends this information to all the consumers. Each consumer only sees his own assignment - the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.

Creating a Kafka Consumer

The first step to start consuming records is to create a `KafkaConsumer` instance. Creating a `KafkaConsumer` is very similar to creating a `KafkaProducer` - you create a Java `Properties` instance with the properties you want to pass to the consumer. We will discuss all the properties in depth later in the chapter. To start we just need to use the 3 mandatory properties: `bootstrap.servers`, `key.deserializer` and `value.deserializer`.

The first property, `bootstrap.servers` is the connection string to Kafka cluster. It is used the exact same way it is used in `KafkaProducer`, and you can refer to Chapter 3 to see specific details on how this is defined. The other two properties `key.deserializer` and `value.deserializer` are similar to the `serializers` defined for the producer, but rather than specifying classes that turn Java objects to a byte array, you need to specify classes that can take a byte array and turn it into a Java object.

There is a fourth property, which is not strictly mandatory, but for now we will pretend it is. The property is `group.id` and it specifies the Consumer Group the KafkaConsumer instance belongs to. While it is possible to create consumers that do not belong to any consumer group, this is far less common and for most of the chapter we will assume the consumer is part of a group.

The following code snippet shows how to create a KafkaConsumer:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

Most of what you see here should be very familiar if you've read Chapter 3 on creating producers. We are planning on consuming Strings as both key and value, so we use the built-in StringDeserializer and we create KafkaConsumer with String types. The only new property here is `group.id` - which is the name of the consumer group this consumer will be part of.

Subscribing to Topics

Once we created a consumer, the next step is to subscribe to one or more topics. The `subcribe()` method takes a list of topics as a parameter, so its pretty simple to use:

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ Here we simply create a list with a single element, the topic name "customerCountries"

It is also possible to call subscribe with a regular expression. The expression can match multiple topic names and if someone creates a new topic with a name that matches, a rebalance will happen almost immediately and the consumers will start consuming from the new topic. This is useful for applications that need to consume from multiple topics and can handle the different types of data the topics will contain. It is most common in applications that replicate data between Kafka and another system.

To subscribe to all test topics, we can call:

```
consumer.subscribe("test.*");
```

The Poll Loop

At the heart of the consumer API is a simple loop for polling the server for more data. Once the consumer subscribes to topics, the poll loop handles all details of coordination, partition rebalances, heartbeats and data fetching, leaving the developer with a clean API that simply returns available data from the assigned partitions. The main body of a consumer will look at follows:

```
try {
  while (true) { ❶
      ConsumerRecords<String, String> records = consumer.poll(100); ❷
      for (ConsumerRecord<String, String> record : records) ❸
      {
          log.debug("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
          record.topic(), record.partition(), record.offset(), record.key(), record.value());

          int updatedCount = 1;
          if (custCountryMap.countainsValue(record.value())) {
              updatedCount = custCountryMap.get(record.value()) + 1;
          }
          custCountryMap.put(record.value(), updatedCount)

          JSONObject json = new JSONObject(custCountryMap);
          System.out.println(json.toString(4)) ❹
      }
  }
} finally {
  consumer.close(); ❺
}
```

❶ This is indeed an infinite loop. Consumers are usually a long-running application that continuously polls Kafka for more data. We will show later in the chapter how to cleanly exit the loop and close the consumer.

❷ This is the most important line in the chapter. The same way that sharks must keep moving or they die, consumers must keep polling Kafka or they will be considered dead and the partitions they are consuming will be handed to another consumer in the group to continue consuming. The parameter we pass `poll()` is a timeout interval and controls how long `poll()` will block if data is not available in the consumer buffer. If this is set to 0, `poll()` will return immediately, otherwise it will wait for the specified number of milliseconds for data to arrive from the broker.

❸ `poll()` returns a list of records. Each record contains the topic and partition the record came from, the offset of the record within the partition, and of course the key and the value of the record. Typically we want to iterate over the list and process the records individually. `poll()` method takes a timeout parameter. This specifies how long it will take `poll` to return, with or without data. The value is typically driven by application needs for quick responses - how fast do you want to return control to the thread that does the polling?

❹ Processing usually ends in writing a result in a data store or updating a stored record. Here, the goal is to keep a running count of customers from each county, so we update a hashtable and print the result as JSON. A more realistic example would store the updates result in a data store.

❺ Always `close()` the consumer before exiting. This will close the network connections and the sockets and will trigger a rebalance immediately rather than wait for the Group Coordinator to discover that the consumer stopped sending heartbeats and is likely dead, which will take longer and therefore result in a longer period of time during which no one consumes messages from a subset of the partitions.

The `poll` loop does a lot more than just get data. The first time you call `poll()` with a new consumer, it is responsible for finding the GroupCoordinator, joining the consumer group and receiving a partition assignment. If a rebalance is triggered, it will be handled inside the poll loop as well. And of course the heartbeats that keep consumers alive are sent from within the poll loop. For this reason, we try to make sure that whatever processing we do between iterations is fast and efficient.

> **TIP**
>
> You can't have multiple consumers that belong to the same group in one thread and you can't have multiple threads safely use the same consumer. One consumer per thread is the rule. To run multiple consumers in the same group in one application, you will need to run each in its own thread. It is useful to wrap the consumer logic in its own object, and then use Java's ExecutorService to start multiple threads each with its own consumer. Confluent blog has a tutorial (http://www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0.9-consumer-client) that shows how to do just that.

Configuring Consumers

So far we have focused on learning the Consumer API, but we've only seen very few of the configuration properties - just the mandatory `bootstrap.servers`, `group.id`, `key.deserializer` and `value.deserializer`. All the Consumer configuration is documented in Apache Kafka documentation (http://kafka.apache.org/documentation.html#newconsumerconfigs). Most of the parameters have reasonable defaults and do not require modification, but some have implications on performance and availability of the consumers. Lets take a look at some of the more important properties:

FETCH.MIN.BYTES

This property allows a consumer to specify the minimum amount of data that it wants to receive from the broker when fetching records. If a Broker receives a request for records from a Consumer but the new records amount to fewer bytes than `min.fetch.bytes`, the broker will wait until more messages are available before sending the records back to the consumer. This reduces the load on both the Consumer and the Broker as they have to handle fewer back-and-forward messages in cases where the topics don't have much new activity (or for lower activity hours of the day). You will want to set this parameter higher than the default if the Consumer is using too much CPU when there isn't much data available, or to reduce load on the brokers when you have large number of consumers.

FETCH.MAX.WAIT.MS

By setting `fetch.min.bytes` you tell Kafka to wait until it has enough data to send before responding to the consumer. `fetch.max.wait.ms` lets you control how long to wait. By default Kafka will wait up to 500ms. This results in up to 500ms of extra latency in case there is not enough data flowing to the Kafka topic to satisfy the minimum amount of data to return. If you want to limit the potential latency (usually due to SLAs controlling the maximum latency of the application), you can set

`fetch.max.wait.ms` to lower value. If you set `fetch.max.wait.ms` to 100ms and `fetch.min.bytes` to 1MB, Kafka will recieve a fetch request from the consumer and will respond with data either when it has 1MB of data to return or after 100ms, whichever happens first.

MAX.PARTITION.FETCH.BYTES

This property controls the maximum number of bytes the server will return per partition. The default is 1MB, which means that when `KafkaConsumer.poll()` returns ConsumerRecords, the record object will use at most `max.partition.fetch.bytes` per partition assigned to the Consumer. So if a topic has 20 partitions, and you have 5 consumers, each consumer will need to have 4MB of memory available for ConsumerRecords. In practice, you will want to allocate more memory as each consumer will need to handle more partitions if other consumers in the group fail. `max.partition.fetch.bytes` must be larger than the largest message a broker will accept (`max.message.size` property in the broker configuration), or the broker may have messages that the consumer will be unable to consumer, in which case the consumer will hang trying to read them. Another important consideration when setting `max.partition.fetch.bytes` is the amount of time it takes the consumer to process data. As you recall, the consumer must call `poll()` frequently enough to avoid session timeout and subsequent rebalance. If the amount of data a single `poll()` returns is very large, it may take the consumer longer to process, which means it will not get to the next iteration of the poll loop in time to avoid a session timeout. If this occures the two options are either to lower `max.partition.fetch.bytes` or to increase the session timeout.

SESSION.TIMEOUT.MS

The amount of time a consumer can be out of contact with the brokers while still considered alive, defaults to 3 seconds. If a consumer goes for more than `session.timeout.ms` without sending a heartbeat to the group coordinator, it is considered dead and the group coordinator will trigger a rebalance of the consumer group to allocate partitions from the dead consumer to the other consumers in the group. This property is closely related to `heartbeat.interval.ms`. `heartbeat.interval.ms` controls how frequently the KafkaConsumer `poll()` method will send a heartbeat to the group coordinator, while `session.timeout.ms` controls how long can a consumer go without sending a heartbeat. Therefore, thoese two properties are typically modified together - `heatbeat.interval.ms` must be lower than `session.timeout.ms`, and is usually set to a 1/3 of the timeout value. So if `session.timeout.ms` is 3 seconds, `heartbeat.interval.ms` should be 1 second. Setting `session.timeout.ms` lower than default will allow consumer groups to detect and recover from failure sooner, but may also cause unwanted rebalances as result of consumers taking longer to complete the poll loop or garbage collection. Setting `session.timeout.ms` higher will reduce the chance of accidental rebalance, but also means it will take longer to detect a real failure.

AUTO.OFFSET.RESET

This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid (usually because the consumer was down for so long that the record with that offset was already aged out of the broker). The default is "latest", which means that lacking a valid offset the consumer will start reading from the newest records (records which were written after the consumer started running). The alternative is "earliest", which means that lacking a valid offset the consumer will read all the data in the partition, starting from the very beginning.

ENABLE.AUTO.COMMIT

We discussed the different options for committing offsets earlier in this chapter. This parameter controls whether the consumer will commit offsets automatically and defaults to `true`. Set it to `false` if you prefer to control when offsets are committed, which is necessary to minimize duplicates and avoid missing data. If you set `enable.auto.commit` to `true` then you may also want to control how frequently offsets will be committed using `auto.commit.interval.ms`.

PARTITION.ASSIGNMENT.STRATEGY

We learned that partitions are assigned to consumers in a consumer group. A `PartitionAssignor` is a class that, given consumers and topics they subscribed to, decides which partitions will be assigned to which consumer. By default Kafka has two assignment strategies:

- Range - which assigns to each consumer a consecutive subset of partitions from each topic it subscribes to. So if consumers C1 and C2 are subscribed to two topics, T1 and T2 and each of the topics has 3 partitions. Then C1 will be assigned partitions 0 and 1 from

topics T1 and T2, while C2 will be assigned partition 2 from those topics. Because each topic has uneven number of partitions and the assignment is done for each topic independently, the first consumer ended up with more partitions than the second. This happens whenever Range assignment is used and the number of consumers does not divide the number of partitions in each topic neatly.

- RoundRobin - which takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one. If C1 and C2 described above would use RoundRobin assignment, C1 would have partitions 0 and 2 from topic T1 and partition 1 from topic T2. C2 would have partition 1 from topic T1 and partitions 0 and 2 from topic T2. In general, if all consumers are subscribed to the same topics (a very common scenario), RoundRobin assignment will end up with all consumers having the same number of partitions (or at most 1 partition difference). `partition.assignment.strategy` allows you to choose a partition assignment strategy. The default is `org.apache.kafka.clients.consumer.RangeAssignor` which implements the Range strategy described above. You can replace it with `org.apache.kafka.clients.consumer.RoundRobinAssignor`. A more advanced option will be to implement your own assignment strategy, in which case `partition.assignment.strategy` should point to the name of your class.

CLIENT.ID

This can be any string, and will be used by the brokers to identify messages sent from the client. It is used in logging, metrics and for quotas.

MAX.POLL.RECORDS

This controls the maximum number of records that a single call to `poll()` will return. This is useful to help control the amount of data your application will need to process in the polling loop.

RECEIVE.BUFFER.BYTES AND SEND.BUFFER.BYTES

These are the sizes of the TCP send and receive buffers used by the sockets when writing and reading data. If these are set to -1, the operating system defaults will be used. It can be a good idea to increase those when producers or consumers communicate with brokers in a different data center - since those network links typically have higher latency and lower bandwidth.

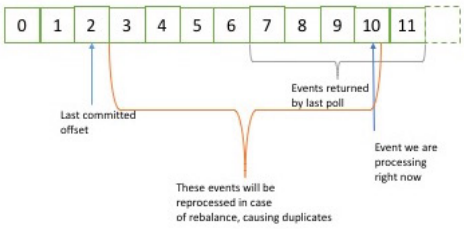
Commits and Offsets

Whenever we call `poll()`, it returns records written to Kafka that consumers in our group did not read yet. This means that we have a way of tracking which records were read by a consumer of the group. As we've discussed before, one of Kafka's unique characteristics is that it does not track acknowledgements from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their position (offset) in each partition.

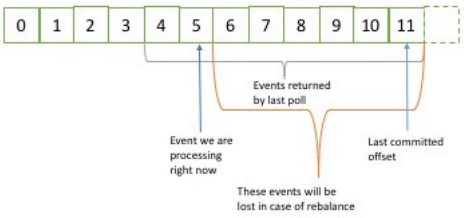
We call the action of updating the current position in the partition a `commit`.

How does a consumer commits an offset? It produces a message to Kafka, to a special `__consumer_offsets` topic, with the committed offset for each partition. As long as all your consumers are up, running and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will *trigger a rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice.



If the committed offset is larger than the offset of the last message the client actually processed, all messages between the last processed offset and the committed offset will be missed by the consumer group.



Clearly managing offsets has large impact on the client application.

The KafkaConsumer API provides multiple ways of committing offsets:

Automatic Commit

The easiest way to commit offsets is to allow the consumer to do it for you. If you configure `enable.auto.commit = true` then every 5 seconds the consumer will commit the largest offset your client received from `poll()`. The 5 seconds interval is the default and is controlled by setting `auto.commit.interval.ms`. As everything else in the consumer, the automatic commits are driven by the poll loop. Whenever you poll, the consumer checks if its time to commit, and if it is, it will commit the offsets it returned in the last poll.

Before using this convenient option, however, it is important to understand the consequences.

Consider that by defaults automatic commit occurs every 5 seconds. Suppose that we are 3 seconds after the most recent commit and a rebalance is triggered. After the rebalancing all consumers will start consuming from the last offset committed. In this case the offset is 3 seconds old, so all the events that arrived in those 3 seconds will be processed twice. It is possible to configure the commit interval to commit more frequently and reduce the window in which records will be duplicated, but it is impossible to completely eliminate them.

With auto-commit enabled, a call to poll will always commit the last offset returned by the previous poll. It doesn't know which events were actually processed, so it is critical to always process all the events returned by poll before calling poll again (or before calling close(), it will also automatically commit offsets). This is usually not an issue, but pay attention when you handle exceptions or otherwise exit the poll loop prematurely.

Automatic commits are convenient, but they don't give developers enough control to avoid duplicate messages.

Commit Current Offset

Most developers use to exercise more control over the time offsets are committed. Both to eliminate the possibility of missing messages and to reduce the number of messages duplicated during rebalancing. The consumer API has the option of committing the current offset at a point that makes sense to the application developer rather than based on a timer.

By setting `auto.commit.offset = false`, offsets will only be committed when the application explicitly chooses to do so. The simplest and most reliable of the commit APIs is `commitSync()`. This API will commit the latest offset returned by `poll()` and return once the offset is committed, throwing an exception if commit fails for some reason.

It is important to remember that `commitSync()` will commit the latest offset returned by `poll()`, so make sure you call `commitSync()` after you are done processing all the records in the collection, or you risk missing messages as described above. When rebalance is triggered, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice.

Here is how we would use `commitSync` to commit offsets once we finished processing the latest batch of messages:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                          record.topic(), record.partition(), record.offset(), record.key(), record.value()); ❶
    }
    try {
      consumer.commitSync(); ❷
    } catch (CommitFailedException e) {
        log.error("commit failed", e) ❸
    }
}
```

❶ Lets assume that by printing the contents of a record, we are done processing it. Your application will be much more involved, and you should determine when you are "done" with a record according to your use-case.

❷ Once we are done "processing" all the records in the current batch, we call `commitSync` to commit the last offset in the batch, before polling for additional messages.

❸ `commitSync` retries committing as long as there is no error that can't be recovered. If this happens there is not much we can do except log an error.

Asynchronous Commit

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increasing the number of potential duplicates that a rebalance will create.

Another option is the asynchronous commit API. Instead of waiting for the broker to respond to a commit, we just send the request and continue on.

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                          record.topic(), record.partition(), record.offset(), record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

❶ Commit the last offset and carry on.

The drawback is that while `commitSync()` will retry the commit until it either succeeds or encounters a non-retriable failure, `commitAsync()` will not retry. The reason it does not retry is that by the time `commitAsync()` receives a response from the server, there may have been a later commit which was already successful. Imagine that we sent a request to commit offset 2000. There is a temporary communication problem, so the broker never gets the request and therefore never respond. Meanwhile, we processed

another batch and successfully committed offset 3000. If `commitAsync()` now retries the previously failed commit, it may succeed in committing offset 2000 *after* offset 3000 was already processed and committed. In case of a rebalance, this will cause more duplicates.

We are mentioning this complication and the importance of correct order of commits, because `commitAsync()` also gives you an option to pass in a callback that will be triggered when the broker responds. It is common to use the callback to log commit errors or to count them in a metric, but if you want to use the callback for retries, you need to be aware of the problem with commit order.

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                          record.topic(), record.partition(), record.offset(), record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
      }); ❶
}
```

❶ We send the commit and carry on, but if the commit fails, the failure and the offsets will be logged.

> **TIP**
>
> A simple pattern to get commit order right for asynchronous retries is to use a monotonically increasing sequence number. Increase the sequence number every time you commit and the add the sequence number at the time of the commit to the commitAsync callback. When you're getting ready to send a retry, check if the commit sequence number the callback got is equal to the instance variable, if it is - there was no newer commit and it is safe to retry. If the instance sequence number is higher, don't retry since a newer commit was already sent.

Combining Synchronous and Asynchronous commits

Normally, occasional failures to commit without retrying are not a huge problem, since if the problem is temporary the following commit will be successful. But if we know that this is the last commit before we close the consumer, or before a rebalance, we want to make extra sure that the commit succeeds.

Therefore a common pattern is to combine `commitAsync()` with `commitSync()` just before shutdown. Here is how it works (We will discuss how to commit just before rebalance when we get to the section about rebalance listeners):

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                              record.topic(), record.partition(), record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

❶ While everything is fine, we use commitAsync. It is faster, and if one commit fails, the next commit will serve as a retry.

❷ But if we are closing, there is no "next commit". We call `commitSync()`, because it will retry until it succeeds or suffers unrecoverable failure.

Commit Specified Offset

Committing the latest offset only allows you to commit as often as you finish processing batches. But what if you want to commit more frequently than that? What if `poll()` returns a huge batch and you want to commit offsets in the middle of the batch to avoid having to process all those rows again if a rebalance occures? You can't just call `commitSync()` or `commitAsync()` - this will commit the last offset returned, which you didn't get to process yet.

Fortunately, the consumer API allows you to call `commitSync()` and `commitAsync()` and pass a map of partitions and offsets that you wish to commit. If you are in the middle of processing a batch of records, and the last message you got from partition 3 in topic "customers" has offset 5000, you can call `commitSync()` to commit offset 5000 for partition 3 in topic "customers". Since your consumer may be consuming more than a single partition, you will need to track offsets on all of them, so moving to this level of precision in controlling offset commits adds complexity to your code.

Here is what commits of specific offsets looks like:

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>(); ❶
int count = 0;

....

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                          record.topic(), record.partition(), record.offset(), record.key(), record.value()); ❷
        currentOffsets.put(new TopicPartition(record.topic(), record.partition()), new OffsetAndMetadata(record.offset
        if (count % 1000 == 0)  ❹
            consumer.commitAsync(currentOffsets, null); ❺
        count++;
    }
}
```

❶ This is the map we will use to manually track offsets

❷ Remember, `println` is a stand-in for whatever processing you do for the records you consume

❸ After reading each record we update the offsets map with the offset of the next message we expect to process. This is where we'll start reading next time we start.

❹ Here, we decide to commit current offsets every 1000 records. In your application you can commit based on time or perhaps content of the records.

❺ I chose to call `commitAsync()`, but `commitSync()` is also completely valid here. Of course, when committing specific offsets you still need to perform all the error handling we've seen in previous sections.

Rebalance Listeners

As we mentioned in previous section about committing offsets, a consumer will want to do some cleanup work before exiting and also before partition rebalancing.

If you know your consumer is about to lose ownership of a partition, you will want to commit offsets of the last event you've processed. If your consumer maintained a buffer with events that it only processes occasionally (for example, the `currentRecords` map we used when explaining `pause()` functionality), you will want to process the events you accumulated before losing ownership of the partition. Perhaps you also need to close file handles, database connections and such.

The consumer API allows you to run your own code when partitions are added or removed from the consumer. You do this by passing a `ConsumerRebalanceListener` when calling the `subscribe()` method we discussed previously. ConsumerRebalanceListener has two methods you can implement:

- `public void onPartitionsRevoked(Collection<TopicPartition> partitions)` is called before the rebalancing starts and after the consumer stopped consuming messages. This is where you want to commit offsets, so whoever gets this partition next will know where to start.
- `public void onPartitionsAssigned(Collection<TopicPartition> partitions)` is called after partitions has been re-assigned to the broker, but before the consumer started consuming messages.

This example will show how to use `onPartitionsRevoked()` to commit offsets before losing ownership of a partition. In the next section we will show a more involved example that also demonstrates the use of `onPartitionsAssigned()`.

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) { ❷
    }

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Lost partitions in rebalance. Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records)
        {
            System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                              record.topic(), record.partition(), record.offset(), record.key(), record.value());
            currentOffsets.put(new TopicPartition(record.topic(), record.partition()), new OffsetAndMetadata(record.of
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // ignore, we're closing
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
  }
```

❶ We start by implementing a ConsumerRebalanceListener

❷ In this example we don't need to do anything when we get a new partition, we'll just start consuming messages.

❸ However, when we are about to lose a partition due to rebalancing, we need to commit offsets. Note that we are committing the latest offsets we've processed, not the latest offsets in the batch we are still processing. This is because a partition could get

revoked while we are still in the middle of a batch. We are committing offsets for all partitions, not just the partitions we are about to lose - since the offsets are for events that were already processed, there is no harm in that. And we are using `commitSync()` to make sure the offsets are committed before the rebalance proceeds.

❹ The most important part - pass the ConsumerRebalanceListener to `subscribe()` method so it will get invoked by the consumer.

Seek and Exactly Once Processing

So far we've seen how to use `poll()` to start consuming messages from the last committed offset in each partition and to proceed in processing all messages in sequence. However, sometimes you want to start reading at a different offset.

If you want to start reading all messages from the beginning of the partition, or you want to skip all the way to the end of the partition and start consuming only new messages, there are APIs specifically for that: `seekToBeginning(TopicPartition tp)` and `seekToEnd(TopicPartition tp)`.

However, the Kafka API also lets you seek to a specific offset. This ability can be used in a variety of ways, for example to go back few messages or skip ahead few messages (perhaps a time-sensitive application that is falling behind will want to skip ahead to more relevant messages), but the most exciting use-case for this ability is when offsets are stored in a system other than Kafka.

Think about this common scenario: Your application is reading events from Kafka (perhaps a clickstream of users in a website), processes the data (perhaps clean up clicks by robots and add session information) and then store the results in a database, NoSQL store or Hadoop. Suppose that we really don't want to lose any data, nor do we want to store the same results in the database twice.

In these cases the consumer loop may look a bit like this:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        currentOffsets.put(new TopicPartition(record.topic(), record.partition()),
                           record.offset());
        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(currentOffsets);
    }
}
```

In this example, we are very paranoid, so we commit offsets after processing each record. However, there is still a chance that our application will crash after the record was stored in the database but before we committed offsets, causing the record to be processed again and the database to contain duplicates.

This could be avoided if there was only a way to store both the record and the offset in one atomic action. Either both the record and the offset are committed, or neither of them are committed. As long as the records are written to a database and the offsets to Kafka, this is impossible.

But what if we wrote both the record and the offset to the database, in one transaction? Then we'll know that either we are done with the record and the offset is committed or we are not, and the record will be reprocessed.

Now the only problem is: if the record is stored in a database and not in Kafka, how will our consumer know where to start reading when it is assigned a partition? This is exactly what `seek()` can be used for. When the consumer starts or when new partitions are assigned, it can look up the offset in the database and `seek()` to that location.

Here is a skeleton example of how this may work. We use the ConsumerRebalanceLister and `seek()` to make sure we start processing at the offsets stored in the database.

```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                commitDBTransaction(); ❶
```

```
        }

    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for(TopicPartition partition: partitions)
            consumer.seek(partition, getOffsetFromDB(partition)); ❷
        }
    }
}


consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);

for (TopicPartition partition: consumer.assignment())
  consumer.seek(partition, getOffsetFromDB(partition));    ❸

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        processRecord(record);
        storeRecordInDB(record);
        storeOffsetInDB(record.topic(), record.partition(), record.offset()); ❹
    }
    commitDBTransaction();
}
```

---

❶ We use an imaginary method here to commit the transaction in the database. The idea here is that the database records and offsets will be inserted to the database as we process the records, and we just need to commit the transaction when we are about to lose the partition to make sure this information will be persisted.

❷ We also have an imaginary method to fetch the offsets from the database, and then we `seek()` to those records when we get ownership of new partitions.

❸ When the consumer first starts, after we subscribed to topics, we call `poll()` once to make sure we join a consumer group and get assigned partitions and then we immediately `seek()` to the correct offset in the partitions we are assigned to. Keep in mind that `seek()` only updates the position we are consuming from, so the next `poll()` will fetch the right messages. If there was an error in `seek()` (for example the offset does not exist), the exception will be thrown by `poll()`;

❹ Another imaginary method - this time we update a table storing the offsets in our database. Here we assume that updating records is fast, so we do an update on every record, but commits are slow, so we only commit at the end of the batch. However this can be optimized in different ways.

There are many different ways to implement exactly-once semantics by storing offsets and data in an external store, but all of them will need to use the `ConsumerRebalanceListener` and `seek()` to make sure offsets are stored in time and that the consumer starts reading messages from the correct location.

But How Do We Exit?

Earlier in this chapter, when we discussed the poll loop, I asked you not to worry about the fact that the consumer polls in an infinite loop and that we will discuss how to exit the loop cleanly. So, lets discuss how to exit cleanly.

When you decide to exit the poll loop, you will need another thread to call `consumer.wakeup()`. If you are running the consumer loop in the main thread, this can be done from a `ShutdownHook`. Note that `consumer.wakeup()` is the only consumer method that is safe to call from a different thread. Calling wakeup will cause `poll()` to exit with `WakeupException`, or if `consumer.wakeup()` was called while the thread was not waiting on poll, the exception will be thrown on the next iteration when poll is called. The `WakeupException` doesn't need to be handled, it was just a way of breaking out of the loop, but it is important that before exitting the thread, you will call `consumer.close()`, this will do any last commits if needed and will send the group coordinator a message that the consumer is leaving the group, so rebalancing will be triggered immediately and you won't need to wait for the session to time out.

Here is what the exit code will look like if the consumer is running in the main application thread. This example is a bit truncated, you can view the full example here.

```
Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                System.out.println("Starting exit...");
                consumer.wakeup(); ❶
                try {
                    mainThread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

...

try {
            // looping until ctrl-c, the shutdown hook will cleanup on exit
            while (true) {
                ConsumerRecords<String, String> records = movingAvg.consumer.poll(1000);
                System.out.println(System.currentTimeMillis() + "  --  waiting for data...");
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key(), record.val
                }
                for (TopicPartition tp: consumer.assignment())
                    System.out.println("Committing offset at position:" + consumer.position(tp));
                movingAvg.consumer.commitSync();
            }
        } catch (WakeupException e) {
            // ignore for shutdown ❷
        } finally {
            consumer.close(); ❸
            System.out.println("Closed consumer and we are done");
        }
    }
```

❶ ShutdownHook runs in a seperate thread, so the only safe action we can take is to call wakeup to break out of the poll loop

❷ Another thread calling wakeup will cause poll to throw a WakeupException. You'll want to catch the exception, to make sure your application doesn't exit unexpectedly, but there is no need to do anything with it.

❸ Before exiting the consumer, make sure you close it cleanly.

Deserializers

As discussed in the previous chapter, Kafka Producers require *serializers* to convert objects into byte arrays that are then sent to Kafka. Similarly, Kafka Consumers require *deserializers* to convert byte arrays recieved from Kafka into Java objects. In previous examples, we just assumed that both the key and the value of each message are Strings and we used the default StringDeserializer in the Consumer configuration.

In the previous chapter about the Kafka Producer, we've seen how to serialize custom types and how to use Avro and AvroSerializers to generate Avro objects from schema definitions and then serialize them when producing messages to Kafka. We will now look at how to create custom deserializers for your own objects and how to use Avro and its deserializers.

It should be obvious that the serializer that was used in producing events to Kafka must match the deserializer that will be used when consuming events. Serializing with IntSerializer and then deserializing with StringDeserializer will not end well. This means that as a developer you need to keep track of which serializers were used to write into each topic, and make sure each topic only contains data that the deserializers you use can interpret. This is one of the benefits of using Avro and the Schema Repository for serializing and deserializing - the AvroSerializer can make sure that all the data written to a specific topic is compatible with the schema of the topic, which means it can be deserialized with the matching deserializer and schema. Any errors in compatibility - in the producer or

the consumer side will be caught easily with an appropriate error message, which means you will not need to try to debug byte arrays for serialization errors.

We will start by quickly showing how to write a custom deserializer, even though this is the less recommended method, and then we will move on to an example of how to use Avro to deserialize message keys and values.

CUSTOM DESERIALIZERS

Lets take the same custom object we serialized in Chapter 3, and write a deserializer for it.

```
public class Customer {
        private int customerID;
        private String customerName;

        public Customer(int ID, String name) {
                this.customerID = ID;
                this.customerName = name;
        }

  public int getID() {
    return customerID;
  }

  public String getName() {
        return customerName;
  }
}
```

The custom deserializer will look as follows:

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements Deserializer<Customer> { ❶

        @Override
  public void configure(Map configs, boolean isKey) {
        // nothing to configure
  }

  @Override
  public Customer deserialize(String topic, byte[] data) {

    int id;
    int nameSize;
    String name;

    try {
      if (data == null)
        return null;
      if (data.length < 8)
        throw new SerializationException("Size of data received by IntegerDeserializer is shorter than expected");

      ByteBuffer buffer = ByteBuffer.wrap(data);
      id = buffer.getInt();
      String nameSize = buffer.getInt();

      byte[] nameBytes = new Array[Byte](nameSize);
      buffer.get(nameBytes);
      name = new String(nameBytes, 'UTF-8');

      return new Customer(id, name); ❷

    } catch (Exception e) {
          throw new SerializationException("Error when serializing Customer to byte[] " + e);
    }
  }
```

```
  @Override
  public void close() {
         // nothing to close
  }
}
```

❶ The consumer also needs the implementation of Customer class, and both the class and the serializer need to match on the producing and consuming applications. In a large organization with many consumers and producers sharing access to the data, this can become challenging.

❷ We are just reversing the logic of the serializer here - we get the customer ID and name out of the byte array and use them to construct the object we need.

The consumer code that uses this serializer will look similar to this example:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.CustomerDeserializer");

KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);

consumer.subscribe("customerCountries")

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(100);
    for (ConsumerRecord<String, Customer> record : records)
    {
    System.out.println("current customer Id: " + record.value().getId() + " and current customer name: " + record.valu
    }
}
```

Again, it is important to note that implementing custom serializer and deserializer is not a recommended practice. It tightly couples producers and consumers and is fragile and error-prone. A better solution would be to use a standard message format such as JSON, Thrift, Protobuf or Avro. We'll now see how to use Avro deserializers with the Kafka consumer. For background on Apache Avro, its schemas and schema-compatibility capabilities, please refer back to Chapter 3.

USING AVRO DESERIALIZATION WITH KAFKA CONSUMER

Lets assume we are using the implementation of Customer class in Avro that was shown in Chapter 3. In order to consume those objects from Kafka, you want to implement a consuming application similar to this:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer consumer = new KafkaConsumer(createConsumerConfig(brokers, groupId, url));
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(1000); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " + record.value().getName()); ❹
    }
    consumer.commitSync();
}
```

❶ We use KafkaAvroDeserializer to deserialize the Avro messages

❷ `schema.registry.url` is a new parameter. This simply points to where we store the schemas. This way the consumer can use the schema that was registered by the producer to deserialize the message.

❸ We specify the generated class, Customer, as the type for the record value

❹ `record.value()` is a Customer instance and we can use it accordingly

Stand Alone Consumer - Why and How to Use a Consumer without a Group

So far we discussed consumer groups, where partitions are assigned automatically to consumers and are rebalanced automatically when consumers are added or removed from the group. Typically, this behavior is just what you want, but in some cases you want something much simpler. Sometimes you know you have a single consumer that always needs to read data from all the partitions in a topic, or from a specific partition in a topic. In this case there is no reason for groups or rebalances, just assign the consumer specific topic and/or partitions, consume messages and commit offsets on occasion.

If this is the case, you don't *subscribe* to a topic, instead you *assign* yourself few partitions. A consumer can either subscribe to topics (and be part of a consumer group), or assign itself partitions, but not both at the same time.

Here is an example of how a consumer can assign itself all partitions of a specific topic and consume from them:

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(), partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(1000); ❸

        for (ConsumerRecord<String, String> record: records) {
            System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n",
                              record.topic(), record.partition(), record.offset(), record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

❶ We start by asking the cluster for the partitions available in the topic. If you only plan on consuming a specific partition, you can skip this part.

❷ Once we know which partitions we want, we call `assign()` with the list.

Other than the lack of rebalances and the need to manually find the partitions, everything else is business as usual. Keep in mind that if someone adds new partitions to the topic, the consumer will not be notified. You will need to handle this by checking `consumer.partitionsFor()` periodically or simply bounce the application whenever partitions are added.

Older consumer APIs

In this chapter we discussed the Java KafkaConsumer client that is part of `org.apache.kafka.clients` package. At the time of writing this chapter, Apache Kafka still has two older clients written in Scala that are part of `kafka.consumer` package which is part of the core Kafka module. These consumers are called SimpleConsumer (which is not very simple. It is a thin wrapper around the Kafka APIs that allow you to consume from specific partitions and offsets) and the High Level Consumer, also known as ZookeeperConsumerConnector, which is somewhat similar to the current consumer in that it has consumer groups and it rebalances

partitions - but it uses Zookeeper to manage consumer groups and it does not give you the same control over commits and rebalances as we have now.

Because the current consumer supports both behaviors and gives much more reliability and control to the developer, we will not discuss the older APIs. If you are interested in using them, please think twice and then refer to Apache Kafka documentation to learn more.

PREV
3. Kafka Producers - Writing Messages to Kafka

NEXT
5. Kafka Internals