[Print](javascript:window.print())

**Course Transcript**

Apache Kafka Development

**Reading from Kafka**

[1. Course Introduction](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t2)

[2. High-Level Consumer API](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t6)

[3. Simple Consumer API](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t10)

[4. Hadoop Consumer API](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t14)

**Configuration Settings**

[1. Broker Configuration Settings](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t19)

[2. Consumer Configuration Settings](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t23)

[3. Producer Configuration Settings](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t27)

[4. Configuring Compression](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t31)

**Writing to Kafka**

[1. Producer API](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t36)

[2. Sync Producers](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t40)

[3. Async Producers](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t44)

[4. Message Acknowledgement](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t48)

[5. Batching Messages](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t52)

[6. Keyed and Non-Keyed Messages](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t56)

[7. Broker Discovery](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t60)

**Testing and Serialization**

[1. Kafka Test Suites](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t65)

[2. Serialization](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t69)

[3. Building a Custom Serializer](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t73)

**Practice: Using Kafka**

[1. Exercise: Creating a Producer and Consumer](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#t78)

Course Introduction

Learning Objective

*After completing this topic, you should be able to*

* *start the course*

**1.**

Apache Kafka comes with a set of APIs, which enable you to write to and read from logs. In this course, Andrew Psaltis will explore the producer and consumer APIs. He will also cover data serialization and deserialization techniques as well as strategies for testing Kafka. Let’s take a look.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

High-Level Consumer API

Learning Objective

*After completing this topic, you should be able to*

* *describe the high-level consumer API for reading from Apache Kafka*

**1.**

In this segment, we're going to talk about the high-level consumer in Kafka. Now I've boiled down an example application to just the essence of really what does this high-level consumer look like and give us a chance to walk through and talk about the different pieces here and how it all fits together. It can, and in fact, it should be a multitiered application, and its very model revolves around the number of partitions that your topic has. And here are some specific rules when you are thinking about that. First if you provide more threads in a partition of topic, some threads are never going to see the message. Okay so let's see how that relates to this code. So in here, we're just going to - this function takes a number of threads - we're going to create this topicCountMap. Okay so these are going to be a count of all topics. We're going to go ahead and we're going to put in some topic, the first one. And we're going to put in a new Integer in that one, the number of threads passed in. Okay and then we're going to go ahead and we're going to get a Map of String to KafkaStream. And that's our consumerMap. And we're going to ask it to create a MessageStreams for us with that topicCountMap.

*[The TextWrangler text editor is open, and it has a menu bar. The high-level-consumer.java file is open, and the following lines of code are displayed : public void run(int a\_numThreads) { Map<String, Integer> topicCountMap = new HashMap<String, Integer>(); topicCountMap.put(topic, new Integer(a\_numThreads)); Map<String, List<KafkaStream<byte[], byte []>>> consumerMap = consumer.createMessageStreams(topicCountMap); List<KafkaStream<byte[], byte []>> streams = consumerMap.get(topic); // now launch all the threads // executor = Executors.newFixedThreadPool(a\_numThreads); // now create an object to consume the messages // int threadNumber = 0; for (final KafkaStream stream : streams) { executor.submit(new ConsumerTest (stream, threadNumber)); } } } } The presenter draws attention to the following elements of the code, one by one: a\_numThreads from line 1, topicCountMap from line 2, (topic, new Integer(a\_numThreads) from line 3, and all of line 4: Map<String, List<KafkaStream<byte[], byte []>>> consumerMap = consumer.createMessageStreams(topicCountMap);.]*

Once that happens, and now you'll see here that we are getting a collection of Kafka streams. Okay so if we created - let's just say we created 100 threads - but really we only have 50 partitions, we have 50 threads that are never going to see a message, so it's a waste of resources. The flip side occurs too. What happens if we had 100 partitions but a\_numThreads was actually 50? Okay in that case, you're going to have several threads that are going to receive data from multiple partitions. Just remember, Kafka deals with partitions in consumer groups. And as one goes up, one goes down; it rebounds and gets the messages there. A thing to keep in mind with that, if you have multiple partitions per thread, so say this number of threads was - just say it was ten - and we actually had 50 partitions. So everyone is reading more data.

*[The presenter highlights the seventh line of code: List<KafkaStream<byte[]>> streams = consumerMap.get(topic);.]*

The guarantee of order goes out the window once you are consuming data from multiple partitions. Okay so keep that in mind. The guarantee of order of the data is out the window once you consume all the partitions. And remember, Kafka's ordering guarantee is that you'll read the partition in the order the data is received. If you start to read data from multiple partitions, then it loses that guarantee. Okay so we've looked at this consumerMap here and we got this stream. And in this case, we're creating a ThreadPool and we're going to go ahead and get this KafkaStream. Every stream, we're going to submit it and then we're going to process this stream when it comes out. If you find yourself needing to add processes to be able to consume it fast enough, as soon as Kafka detects that there is a new consumer with the same message group or same topic as another, it will stop to rebalance if necessary. And plus, we see the assignment of a partition to a thread.

*[The presenter draws attention to the ThreadPool element in line 11 of the code: executor = Executors.newFixedThreadPool(a\_numThreads);.]*

So in this segment, we've gone over using the high-level consumer. We've gone over why I need to be concerned about having too many threads that are executing and not enough data being written. We've flipped it around, and we talked about how you could have too little data or too few threads to read a lot of data. Okay so these are some of the things to keep in mind as you are going to use the high-level consumer. And there is one other thing to be aware of. No place in here do you see calls the ZooKeeper. It's after we handled the data. Behind the scenes, it's committing offsets. So this high-level consumer really is working at a higher level.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Simple Consumer API

Learning Objective

*After completing this topic, you should be able to*

* *describe the simple consumer API for reading from Apache Kafka*

**1.**

In this segment, we're going to talk about the simple consumer API. So on the screen here, I have a function that we would use to consume data. So you can see off the bat, a little bit more complex than the high-level consumer. So let's just take a moment and walk through it. So we have this response or we're fetching this response to data. We get back the messageSet, okay, its topic, partition. And we're going to get this messageAndOffset object or we can get the offset, just asking for it to offset. And we can jump here over to do that. Alright so in this case here, if you have the currentOffset less than the read, then it's an old offset. Okay. And you can just see the readOffset, messageAndOffset.nextOffset(), the ByteBuffer payload. Okay we're going to basically read the message and the offset, get the message in here just to payload. So we're just going to suck out that data. Okay then we set up bytes, we set up payload, and we're going to end up returning here. Okay in this case, it just prints a line. So there are a couple of things you keep in mind when you're considering using a simple consumer. The first part is you could read the messages multiple times, alright. So we're sitting here controlling this loop here. We could go back around and read more data.

*[The QuickTime Player application is open. It has a menu bar and a details pane. The simple\_consumer.java file is open, and the following lines of code are displayed in the details pane: long numRead = 0; for (MessageAndOffset messageAndOffset : fetchResponse.messageSet(a\_topic, a\_partition)) { long currentOffset = messageAndOffset.offset(); if (currentOffset < readOffset) { System.out.println("Found an old offset: " + currentOffset + "Expecting: " +readOffset); continue; } readOffset = messageAndOffset.nextOffset(); ByteBuffer payload = messageAndOffset.message().payload(); byte[] bytes = new byte[payload.limit()]; payload.get(bytes); System.out.printIn(String.valueOf(messageAndOffset .offset() + ":" + new String(bytes, "UTF-8")); numRead++; a\_maxReads--; } The presenter draws attention to the following elements from the block of code, one by one: fetchResponse.messageSet(a\_topic, a\_partition)) from line 2 and messageAndOffset.offset(); from line 3. He then draws attention to lines 9 and 10 of the code, readOffset = messageAndOffset.nextOffset(); and ByteBuffer payload = messageAndOffset.message().payload();. He draws attention to the code relating to the  "for" loop in line 2.]*

You're only consuming a subset of the partitions in a topic in a given process. So if this is running on four nodes and this being, you know, this code here, then it would all be consuming independently. You may also be able to manage transactions. So you can see here, you're getting direct access to the offset. So that's one of the things that differentiates the high-level consumer from the simple consumer. The high-level consumer takes care of writing the offsets for you, and this consumer here is using the simple API. It's responsible for writing the offsets back into a foreign destination. So in this segment, we've gone over some of the differences between the simple and the high-level consumer API. And then we dug into a little bit of the particulars of the simple consumer API. With this, you should be able to make the decision to use the high-level or the simple. And in reality, a lot of times it's best to start with the high-level until it doesn't meet your needs and then look at switching.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Hadoop Consumer API

Learning Objective

*After completing this topic, you should be able to*

* *describe the Hadoop consumer API for reading from Apache Kafka*

**1.**

We'll talk about the Hadoop API that's available with Kafka. Now really it's not a separate API that you use, it's a tool that's called Camus. Now the Kafka web site links to this repo that's maintained by LinkedIn that is for this product called Camus. Now they sometimes don't get around to fixing things too often. You can see some of this is quite old. There's another version of it that is maintained by Confluent, and their version of it - you could tell it's getting a little bit more activity. They've obtained stuff 24 days ago. You know some stuff longer as that product matures, but they seem to be pretty active with it. So let's take a moment and, kind of, look at how it works. So Camus is a MapReduce job. You set up a configuration, you tell it what topics to read, and you tell it where the set time stamp is. And then it will go and read as much as it can from whatever topics are configured and put them into HDFS.

*[The QuickTime Player is open. It has a menu bar, two tabs - linkedin/camus and confluentinc/camus - an address field displaying the URL GitHub, Inc. [US] https://github.com/linkedin/camus; a display window; and a navigation pane on the right-hand side. A table is presented with details of updates for Camus. A number of options are listed in the navigation pane: Code, Issues, Pull Requests, Wiki, Pulse, and Graphs. There is also a HTTPS clone URL textbox, set to github.com, and buttons for Clone in Desktop and Download ZIP. The presenter clicks the confluentinc/camus tab to open its list of Camus updates. He scrolls down the table to display these.]*

Okay so this is a MapReduce job that you schedule. It consumes data from Kafka, writes it into HDFS. So that makes it unique compared to, say, the high-level consumer is that it's writing its offsets into HDFS. Okay so it's writing offsets there. With this simple consumer, you could have the right offsets wherever you'd like. So if you're interested in Camus and you do want to take data from Kafka and put it in HDFS, this would be a project to look at. And you could look at both of them and see which one makes the most sense in your organization. Okay so the API is really not a single API. It's a methodology and a way of consuming data from Kafka that goes Kafka into this consumer and then to HDFS.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Broker Configuration Settings

Learning Objective

*After completing this topic, you should be able to*

* *configure Apache Kafka brokers*

**1.**

In this segment, we're going to go over the broker configuration settings. Now there is a lot of settings, but we're just going to focus in on the ones that you're most likely to modify and some pretty important ones. Not that they're not all important, but there are a lot of them that really would not give us a lot of benefit if we just run over every single one. So let's go through this config file and talk about it as we go and highlight the important ones. Broker.id is the first thing that you see here. It's important and we talked about that before, but the broker.id, this would be unique per broker in your cluster. Okay so by default, it's a 0. The port - pretty straightforward there - right, that's the port that Kafka is going to listen. Now a couple of things that are important here. The host.name. So we have host.name and we have advertised.host.name. So the host.name of the broker will bind to. Right, it says right there. If it's not set, the server will bind to all interfaces, okay.

*[A Shell window is open. It contains Kafka configuration details in two sections:  Server Basics and Socket Server Settings. The presenter highlights broker.id=0 in the Server Basics section. He then points out the string host.name=localhost in the Socket Server Settings section.]*

And then the hostname is advertised.host.name That's the hostname the broker will advertise to producers and consumers. So if it is not set, it will use the value for the host.name if it is not configured. And so, you can see with this here. Now when these come into play, and I've seen them impact folks quite a bit, is when you run things in EC2. And you have the private IPs and the public IPs and various synchronizations thereof. Guess you don’t want to pay attention to host.name and advertised.host.name when you look at running in a cloud provider. Let's go down a bit here and let's take a look. This number of io threads, it defaults to 8. You don't want to have this, to be about as many threads as you have disks. Okay so that's going to…you're going to want that to be a number of threads that matches the amount of disks that you have, the number of disks. Okay a lot of this we're not going to worry about.

*[The presenter scrolls down the screen and displays more configuration settings in the Socket Server Settings section. He points out num.io.threads=8 from the line The number of threads doing disk I/O num.io.threads=8.]*

Okay let's look at num.partitions here. So this is interesting. Okay so this is the configured number of partitions. This comes into play if you create a topic and do not give it the partitions. Flexible…is that it will use configuration or you could have it where it's done via configuration as you create the topic. Okay but you got to remember, if you don't set the partition, that's what you'll get. And this becomes more important when you look at another property here. Okay so its log.retention, we'll come back and look at those. Okay Zookeeper, okay. One that's not in here we need to talk about is going to be auto.create.topics.enable. Okay if you have that, then you allow a producer to write a random topic, say, I just created a topic called topic ABCD and no one had that topic before. The first time I'm going to produce data into it, if this auto.create.topics.enable is true, then it will allow the producer to automatically populate the topic.

*[In the Shell window, the presenter scrolls down the page and highlights num.partitions=1 in the Log Basics section. He scrolls down the screen further and displays the Log Retention Policy and the Zookeeper sections. In the Zookeeper section, he types auto.create.topics.enable.]*

Okay so it may or may not be useful in your environment. Keep in mind, if auto.create.topics.enable is true, then it is the configured number of partitions that will be used. So if you want to use different partitions, you may not want to set this property to be true. There is another one that may be useful in your environment as well, and that's going to be delete.topic.enable. If you have that set, then you are allowing the deletion of topics in the command line. So especially during development, you may come up with a topic name. You want to get rid of it, clean things out. So you probably, you may want to have topic, delete, enable. Other thing that we see here that's important is the log.retention.

Okay by default, it's got a week's worth of data. You could set that to be milliseconds or minutes or hours. Okay it's really how long Kafka is going to hold onto that data on disk. It's not always in memory. Okay so how long that Kafka will manage that data. So we've touched on a handful of some of the more important broker configurations. At this point, you should feel comfortable changing the configurations and starting up your local broker.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Consumer Configuration Settings

Learning Objective

*After completing this topic, you should be able to*

* *configure Apache Kafka consumers*

**1.**

In this segment, we're going to go over the important aspects of the consumer configuration. Here are a lot of different configuration options. So what you are going to touch on are the ones that you're most likely to change and those that are very important to pay attention to. So let's go through this file here. So zookeeper.connect, that's going to be very important. And our consumers are going to need to talk to ZooKeeper. The connection.timeout. We're going to want to have that so we do time-out appropriately if we can't get the ZooKeeper. Now the next one, this is a very important. This is our group.id, alright. Remember it's all the consumers in a consumer group that consume the whole topic. Okay so if you have multiple consumers with a group.id test-consumer-group, then they are part of that collective and together they read the whole topic. Okay set up positions of that parallelism. The next one that we're going to take a look at is this auto.commit.enable=true or you could set it equal to false.

*[The Shell window is open. It displays configuration settings for Kafka. The presenter highlights zookeeper.connect=127.0.0.1:2181 in the string server. e.g. "127.0.0.1:3000, 127.0.0.1:3001, 127.0.0.1.3002" zookeeper.connect=127.0.0.1:2181. He then points out the following lines of code, one by one: timeout in ms for connecting to zookeeper zookeeper.connection.timeout.ms=6000, consumer group id group.id=test-consumer-group, and auto.commit.enable=true.]*

So with that one, you are telling the API that you are using for it to autocommit offsets for you back into storage. If you want to be in control of that and you want to explicitly handle it, then you should set that to false. And the last one is this auto.offset.reset. You know, I have put an extra here -- largest or smallest. What this means is that if your consumer crashes and comes back up again and say when it did, it lost its state, so say things went missing in ZooKeeper. When your client starts back up again, if it's configured to do the largest, it's going to read the latest time stamp from Kafka. If you have it set to largest, it's going to read all of it. Okay so depending upon how you do this, you can end up reprocessing data. So you can again read the smallest time set, the smallest offset, or the largest offset. Okay and that controls where in that partition you are reading. Okay so this is a handful of consumer configurations that are important to remember.

Okay. The one that you're likely to change as you're developing and debugging things is going to be this auto.offset.reset. There's going to be times when we want to read the smallest and there's going to be other times when we want to read the largest. So in this segment, we covered those four key consumer configurations: the zookeeper.connection, the group.id, auto.commit.enable, and auto.offset.reset. With this, you should be ready to configure a client.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Producer Configuration Settings

Learning Objective

*After completing this topic, you should be able to*

* *configure Apache Kafka producers*

**1.**

In this segment, we're going to discuss some of the important configuration options you have for the producer with Apache Kafka. The first one that's important is a metadata.broker.list. This takes a comma delimited list that has their hostname. It could be IP address, colon, the port number that Kafka is listening on. So in this case, I have one Kafka broker listening on localhost port 9092. And again, it would just be a comma delimited list of the brokers in the cluster. Second important property for the producer is the required.acks. Okay so your options here are 0, 1, or -1. The default is 0. And what this means is that 0 means a producer never waits for an acknowledgement from the broker at all. So this provides obviously the lowest latency but it's the weakest durability guarantee. So what that means is that when you set it the required.acks to 0 and your producer sends a message, it's basically fire and forget. It doesn't wait for any response at all.

*[The Shell window is open. It features producer configurations for Kafka, under the Producer Basics heading. The presenter draws attention to the following lines of code,  one by one: metadata.broker.list=localhost:9092 and request.required.acks=0|1|-1.]*

A 1 means a producer will get an acknowledgement after the leader replica has received the data. Okay so this provides better durability because the client is waiting until a server acknowledges the request, but not yet replicated it to other brokers. So you could still lose the data, but it's a little bit better. -1 means that the producer gets an acknowledgement after all the in-sync replicas have received the data. So this option receives the greatest level of durability. However, it doesn't necessarily completely eliminate the risk of message loss because the number of in-sync replicas may, in some cases, shrink to 1, so if you had brokers that are down. So this may have the lowest level of throughput but the highest level of durability. So 0 is a default, highest level of throughput, lowest durability, you wait for no acknowledgements. 1, little bit better durability, you wait for the leader to acknowledge, possibly a little bit lower throughput. And -1 has the highest potential of durability because you wait for all the replicas to acknowledge, receive the data, and then, possibly because of that, the lowest throughput.

Okay the third most important configuration option for the producer is the type. This could be either async or sync. The synchronous one is the default, and the difference between the two is synchronous, sends a message,  waits for a response. Of course, if the required .acks is 1 or -1. The asynchronous producer does not wait and also allows you to batch messages together. So you could send, say, 200 messages at a time. Now because this fires the message and you've to wait for the callback, you have to handle failure slightly differently than the synchronous one that sends a request and you wait for a response. The fourth most important configuration option is the serializer class they use.

*[In the Producer Basics section, the presenter points out two strings of code, one by one: producer.type-async|sync and message encoder serializer.class=kafka.serializer.DefaultEncoder.]*

The default encoder just takes a byte array and returns back a byte array. And this is going to be used to tell the producer how to serialize that data. So these are the four most important configuration options to be aware of with the producer. They are not the only ones. As you look through the producer config or look at the online documentation for producer configuration, you'll see many more options. But these are the four important ones to remember. So again we have the metadata.broker.list, we have the required.acks, default being 0. And again that's highest throughput all the way to -1, being the lowest throughput, highest durability. With the producer type, the default being synchronous, so you've got the option of async and the message encoder. Again the default being the default encoder, it takes a byte array, returns a byte array.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Configuring Compression

Learning Objective

*After completing this topic, you should be able to*

* *configure compression in Apache Kafka*

**1.**

In this segment, we're going to discuss configuring compression for Apache Kafka and for the producer specifically. There are two configuration settings that drive compression for the producer. The first is the compression.codec. The default for this is none. The options are gzip, snappy, or lz4, or none is the default. And you could also use old config values - 0, 1, 2, 3 - instead of the text. If this is set, this will allow that type of compression for all topics. To control it on a topic by topic basis, then you could have set the compressed.topics property. This will be a comma separated list of the topics that you want to compress. This also assumes that the compression.codec is not set to none.

*[The Shell window is open. It features Kafka compression configurations.The presenter draws attention to the following strings of code,  one by one: the old config values work as well: 0, 1, 2, 3 for none, gzip, snappy, lz4, respectively compression.codec=none and allow topic level compression compressed.topics=.]*

So if you set the compression.codec to perhaps gzip, and then you have a compressed topics list set. So let's say if we did this, if we had this to be, say, topica, topicb, and then here we had gzip, then we would be telling the producer that we want topica and topicb compressed with the gzip algorithm. If this is mistakenly left as none, then there's no compression that will happen here. And if we set it like this, and this is commented out, so it's ineffective. Then again, that would be gzip compression for all topics. If you use the new Kafka producer, which is not yet considered production-ready, instead of codec, it has type, okay, and the values are the same, same options: none, gzip, snappy, lz4. That is how you configure compression with the Kafka producer.

*[Continuing in the same Shell window, the presenter types topica, topicb after compressed.topics=. In the string, the old config values work as well: 0, 1, 2, 3 for none, gzip, snappy, lz4, respectively compression.codec=none, he replaces none with gzip. He then deletes gzip and retypes none. He again sets it to gzip and adds a # symbol before compressed.topics=topicb. Next he types compression.type= immediately below compression.codec=gzip.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Producer API

Learning Objective

*After completing this topic, you should be able to*

* *describe the producer API in Apache Kafka*

**1.**

In this segment, we're going to go over the producer API for Apache Kafka. As of Apache Kafka 8.2, it's recommended that all new applications use the new Java client, a new Java producer. So when you create your project, the artifact you're going to want to use is this Kafka client's 0.8.2.0 in a, maybe, later version in the future. So this would be the Maven artifact you want to use for anything 8.2 or later. This is a brand new Java client, has slightly higher performance, and is a little bit…I don't know easier to use, maybe just refactored and cleaner, and it just shows the evolution of Apache Kafka. So let's go over using that real quick. So I have this project where I included that dependency. I am going to go ahead and set up the Properties. Again these are the four properties we talked about in the past that are the key ones to pay attention to. I could have chosen the defaults for the producer.type and the requested request.required.acks. In this case, I chose to set them. I could've also chosen the default for the serializer. Really, the primary one I must set is going to be the metadata.broker.list. I've to tell it where the brokers are.

*[The IntelliJ IDEA application is open, and it has a menu bar and two tabs: kafka-producer-demo and KafkaProducerDemo.java. The KafkaProducerDemo.java file is open, and it displays the following code: import org.apache.kafka.clients.producer.KafkaProducer; import org-apache.kafka.clients.producer.ProducerRecord; import java.util.List; import java.util.Map; import java.util.Properties; public class KafkaProducerDemo { static void main(String [] args){ Properties props = new Properties (); props.put("serializer.class", "kafka.serializer.StringEncoder"); props.put("metadata.broker.list", "localhost:9092"); props.put("producer.type","sync"); props.put("request.required.acks",1); //This is threadsafe, if we have many threads in our application //we would want to share this. KafkaProducer kafkaProducer = new KafkaProducer (props); KafkaProducer.send (new ProducerRecord<byte []>("the-topic", "key".getBytes(), "value".getBytes ())); List partitionInfoList = kafkaProducer.partitoinsFor ("myTopic"); Map metrics = kafkaProducer.metrics(); //use the producer and then close when done kafkaProducer.close (); } } The presenter clicks the kafka-producer-demo tab and displays information relating to the groupId, artifactId, and version of Kafka, as well as a list of related dependencies. He draws attention to the following strings of code, one by one: <groupId>>org.apache.kafka</groupId> <artifactId>kafka-clients</artifactId> <version>0.8.2.0</version>. He then reopens the KafkaProducerDemo.java tab and draws attention to the elements of code relating to Properties.]*

The others have pretty decent defaults. So I set up the properties. And you may want to have these come from a file, or from your environment, or somewhere other than hard coded like I've done here. This is just an example, perhaps not the way you'd want to do it in practice. I think that you can go ahead and create the kafkaProducer that takes the properties I've set up. Now this comment here that this producer, the new Java API producer is thread safe. So if we had an application with many threads, perhaps it was some sort of service we're building, we would want to construct this once and share it amongst the many threads. So you do not need to create it multiple times. So now I'll go ahead, have it created. I would do some processing in my application and I can go ahead and call a send method. And we'll go into the send method in more detail in future segments. For now, let's just go over that we call send. We're going to pass the ProducerRecord, whatever my topic may have been, whatever the key may have been, and whatever the value would be. And in both these cases, we get the bytes. Remember the ProducerRecord just takes two byte arrays - the key and the value.

*[In the KafkaProducerDemo.java file, the presenter draws attention to the block of code relating to the producer.]*

And when it comes down to it, that's what Kafka understands is byte arrays. We may serialize the data prior to that in two different formats, but ultimately it ends up as a byte array when it's been sent to the broker. Some other functions that you may find useful in the producer are things like getting partition information for a topic. So say, I have my topic. I could go and get a list of partition information about the topic. The kafkaProducer also supplies a variety of different metrics about what it's doing and how it's performing. This uses a quota-held metrics if you're familiar with that. So if I call this kafkaProducer.metrics function, I get back a map of the metrics names and the metrics. So then you can interrogate it and build various different types of metrics monitoring applications.

*[Next the presenter draws attention to the section of code relating to list partition and map metrics.]*

When I'm all done with it and say this, anyway application is closing down, I am going to restart it, whatever may happen, then you'd want to go ahead and close the producer. If you were to close the producer when you are done with it, that would cause resource leaks. We have network connections that are open or possibly file handles. Okay so these are the general functions that exist on the producer. And in future segments, we'll go over the send in particular in configuring some other parts of the producer. For now just keep in mind it is thread-safe. When you construct it, it takes properties. There's a send call function on it. There are actually a couple of varieties of send. You can get the partition information list, you can get the metrics, and then when you're done with the producer, you go ahead and close it. Those are things to keep in mind when using the kafkaProducer.

*[The presenter points out the final part of the code that is: //use the producer and then close when done kafkaProducer.close( );.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Sync Producers

Learning Objective

*After completing this topic, you should be able to*

* *describe the SyncProducer API in Apache Kafka*

**1.**

In this segment, we're going to go over the Kafka sync producer. So I've gone ahead and I've created this application. I've got my two Properties set here, two that are required for this demo. One - the metadata.broker.list - which the reality is it's required for every demo or every time you're going to go ahead and use Kafka. And the producer.type, I am setting to sync. Again as we've mentioned before, these properties could come from a configuration file, they could come from your environment, but somehow you have to get them. And in this case, I've just hard coded them into a Properties object. And I'll go ahead and construct my KafkaProducer, pass in the properties. And then I am going to go ahead and start to use it. So the first thing I'll do is I am going to create a ProducerRecord. That is a generic that takes two byte arrays - the first for the key, the second for the value. So it takes the name of the topic, then the key, whatever the bytes are, the value, whatever the bytes are. So in this case, I just have the text string key, the text string value. Again, it's whatever your data is, and you'd want to make sure you get the bytes for that so you end up with a byte array.

*[The IntelliJ IDEA application is open. It has a menu bar and one open tab,  KafkaProducerDemo.java.  The following block of code is displayed: import org.apache.kafka.clients.producer.KafkaProducer; import org-apache.kafka.clients.producer.ProducerRecord; import org.apache.kafka.clients.producer.RecordMetadata; import java.util.Properties; import java.util.concurrent.ExecutionException; public class KafkaProducerDemo { static void main(String [] args){ Properties props = new Properties (); props.put ("metadata.broker.list", "localhost:9092"); props.put("producer.type","sync"); KafkaProducer kafkaProducer = new KafkaProducer (props); try { ProducerRecord<byte [], byte[]> producerRecord = new ProducerRecord<byte [],byte []>("the-topic", "key".getBytes (), "value".getBytes()); RecordMetadata recordMetadata = (RecordMetadata) kafkaProducer.send(producerRecord).get (); System.out.println("Record written to offset: " + recordMetadata.offset()); System.out.println ("Record written to partition: " + recordMetadata.partition()); } catch (InterruptedException e) { e.printStackTrace (); }catch (ExecutionException e) { e.printStackTrace (); } } } The presenter draws attention to the lines of code dealing with properties and then to those relating to producer.]*

Now that we have that constructed, the next thing I want to do is we want to go ahead and call the send method. That takes the ProducerRecord. By default, that send is an asynchronous call regardless of whether or not you have a synchronous or asynchronous producer. If we want to make this where it appears to be totally blocking, what we want to do is call the get method. If you don't call the get method, then this returns back a future. We will see in the future segment, no pun intended, of how to use the async functionality and to handle that future that gets back. But in this case…say we wanted to truly have a synchronous producer that has the synchronous behavior. Then we would want to call get on this future. That will enforce that you either get back the RecordMetadata or an exception gets thrown if it can't complete.

*[In the KafkaProducerDemo.java file, the presenter next draws attention to the section of code dealing with RecordMetadata.]*

So we get back to RecordMetadata, it has things in there to recordMetadata.offset. It has the partition and it has some other items as well. So we could get the topic from here. We know what the topic is. But if we wanted to again, we can get the topic. So that's using the synchronous producer behavior for the KafkaProducer to, again, to enforce the synchronicity and the synchronous behavior. You're going to want to call get on the future that comes back from the send method. Okay that forces synchronous behavior and you get back a RecordMetadata object. When you get back this metadata object, you are ensured that whatever your acknowledgments were set at, 0, 1, or -1, that that was completed. Okay, so this is how we choose the synchronous KafkaProducer.

*[In the KafkaProducerDemo.java file, the presenter draws attention to the following lines of code: System.out.println("Record written to offset: " + recordMetadata.offset()); System.out.println("Record written to partition: " + recordMetadata.partition()); He then presses Enter at the end of second of these two lines and types recordMetadata, bringing up a box displaying a list of options that includes offset(), partition(), topic(), and equals(Object obj). He selects topic(), and the box closes. He then highlights the line recordMetadata.topic();.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Async Producers

Learning Objective

*After completing this topic, you should be able to*

* *describe the AsyncProducer API in Apache Kafka*

**1.**

In this segment, we're going to go over using the async KafkaProducer. So I've gone ahead here and I've set up the properties that I need, a metadata.broker.list, and I have set up the producer.type to be synchronous. I've gone ahead and I've created my KafkaProducer passing in those properties. Again as we discussed, those properties could come from a configuration file or from your environment. In this case, I've hard coded them. It may not be what you want to do in practice. I've gone ahead and created ProducerRecord. Again this takes a byte array for the key and a byte array for the value. In this demo, I'm just using the topic, and my key is a text key and the value is just a text value, and I get the bytes of both of those.

*[The TextWrangler text editor is open. It has a menu bar and a details pane in which the 83238\_Apache\_Kafka\_AsyncAckProducer.java file is open. The following lines of code are displayed: import org.apache.kafka.clients.producer.KafkaProducer; import org-apache.kafka.clients.producer.ProducerRecord; import org.apache.kafka.clients.producer.RecordMetadata; import java.util.Properties; import java.util.concurrent.ExecutionException; public class KafkaAsyncProducerDemo { static void main(String [] args){ Properties props = new Properties ( ); props.put ("metadata.broker.list", "localhost:9092"); props.put("producer.type","sync"); KafkaProducer kafkaProducer = new KafkaProducer (props); ProducerRecord<byte [], byte[]> producerRecord = new ProducerRecord<byte [],byte []>("the-topic", "key".getBytes (), "value".getBytes()); kafkaProducer.send(producerRecord, new Callback () public void onCompletion(RecordMetadata metadata, Exception e) { if(e !=null) { e.printStackTrace(); } else { System. out.println("Record written to offset: " + recordMetadata.offset()); System.out.println ("Record written to partition: " + recordMetadata.partition()); } } } }; } } The presenter draws attention to the lines of code relating to properties and then to those relating to producer. He highlights the following line: ProducerRecord<byte [], byte[]> producerRecord = new ProducerRecord<byte [],byte []>("the-topic",.]*

Now I am going to call kafkaProducer.send. If you remember from the synchronous call, we didn't have a Callback. We basically had it look like this. I've put that on the wrong line, didn't I? Now let's go back here. Okay so we basically had this, alright - kafkaProducer.send - and we did this producerRecord and passed in nothing. Alright so that's a null for the Callback. In this case, we're passing in the Callback like you see below. Okay you may want to put this in a different method or somewhere else. I'm doing it in line here as an example. So we create a new Callback. It has to implement on completion and it takes two parameters - RecordMetadata and an Exception. If the Exception is not null, then something went wrong. Okay in this case, I am just printing the StackTrace. You may want to retry, you may want to interrogate that inspection object, and take some other measures. So if it's not null, you'd want to do something.

*[In the 83238\_Apache\_Kafka\_AsyncAckProducer.java file, the presenter highlights the following line of code: kafkaProducer.send(producerRecord,. He presses Enter at the end of the following line: ProducerRecord<byte[],byte[]> producerRecord = new ProducerRecord<byte[]>("the-topic","key".getBytes(), "value".getBytes());. In the new blank line, he types kafkaProducer.send(producerRecord); and then selects the entire line and deletes it.]*

Okay so here, you know, we may have a comment here of, you know, "do something with this…retry, raise an alert, fail etc…." That's really going to be up to your business logic. On the other hand, if the Exception object is null, then things went successfully. So here again, we're just showing as an example, we're going to print out the offset that it was written to and the partition. You may do something here, maybe, notify clients that the record was written successfully. You may record it, right. And we have metrics "may record success."  Whatever it is you want to do, you would do that in this else block for if it's successful.

*[In the 83238\_Apache\_Kafka\_AsyncAckProducer.java file,  the presenter presses Enter at the end of the line if(e != null) {, and a new blank line appears. He   types //do something with this... retry, raise an alert, fail etc.... He then draws attention to the following lines of code: System.out.println("Record written to offset: " + recordMetadata.offset()); System.out.println("Record written to partition: " + recordMetadata.partition()); He then presses Enter after else {, and a new blank line appears. He types the following: //may notify clients that the record was written successfully //may record success]*

Okay again, this could on completion and this Callback object you could put totally somewhere else, but you'd just pass into or send the Callback object that is going to be get called when the send does complete. This will be for every message sent. So even if you have batches of messages that are sent, you're going to know which one was which based upon the metadata that comes back. So you'd be able to tell what message was sent and then act accordingly. Okay so to sum up, this is using the async producer. And I just noticed I had a typo here. This should be async, not sync. Okay so a couple of key things: one, we need the producer there to be async; and two, we need to pass in a Callback that will get called when the send does complete. Okay so don't do what I did of passing in a sync producer and a Callback, not what you want to do. You want the async and the Callback.

*[In the same file, the presenter goes up to the line props.put("producer.type", "sync");. He changes sync to async.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Message Acknowledgement

Learning Objective

*After completing this topic, you should be able to*

* *configure message acknowledgement, or acking, in Apache Kafka*

**1.**

In this segment, we're going to go over using the acknowledgement functionality with an async producer. When it comes down to it, the acknowledgement functionality from the broker works the same, regardless whether or not the producer is asynchronous or synchronous. All that changes is whether it's async in sending patches or it's synchronous. The acknowledgment functionality is the same. In this case, I've gone ahead with the Properties object right here in the code. The default value again is 0, meaning that it will have the highest throughput and there is no acknowledgement from any replicas, not the leader or not the in-sync replicas. So a highest level message throughput, least level of durability. So your greatest risk for losing messages. A 1 would again imply that the leader is going to acknowledge, so a slightly less throughput because we're waiting for an acknowledgement, a little bit more durability because we at least know that the leader replica received the message and is committed.

*[The TextWrangler text editor is open. It has a menu bar, and the 83238\_Apache\_Kafka\_AsyncAckProducer.java file is open. The following block of code is displayed: import org.apache.kafka.clients.producer.RecordMetadata; import java.util.Properties; import java.util.concurrent.ExecutionException; public class KafkaAsyncAckProducerDemo { static void main(String [] args){ Properties props = new Properties (); props.put ("metadata.broker.list", "localhost:9092"); props.put("producer.type","async"); props.put("request., required.acks", 1); KafkaProducer kafkaProducer = new KafkaProducer (props); ProducerRecord<byte [], byte[]> producerRecord = new ProducerRecord<byte [],byte []>("the-topic", "key".getBytes (), "value".getBytes()); kafkaProducer.send(producerRecord, new Callback () public void onCompletion(RecordMetadata metadata, Exception e) { if(e !=null) { //do something with this... retry, raise an alert, fail etc.... e.printStackTrace(); } else { //may notify clients that the record was written successfully //may record success System. out.println("Record written to offset: " + recordMetadata.offset()); System.out.println ("Record written to partition: " + recordMetadata.partition()); } } } }; } The presenter draws attention to the section of code relating to properties. In the line props.put("request.required.acks",1);, he changes the value 1 to 0. He then changes it back to 1.]*

-1 is going to be the least amount of throughput of the three options but the most amount of durability - potential for durability. You could still lose your message. But in this case, the leader and all the in-sync replicas will respond before the client gets back a response. So if we're using 0 here and we come down to this Producer.send call, our Callback, onCompletion is going to get called with a response before any of the broker is acknowledged. So we're not going to wait for the leader or the replicas. This code is going to get executed pretty quickly. If it's a 1, we're waiting for the leader, then we know that our code here is onCompletion. It's only going to get called when the leader acknowledges receipt of the message. And then again, if this is -1, then our code here is only going to get called when we know that all replicas - leader and in-sync - have responded.

*[The presenter then draws attention to the following lines of code: kafkaProducer.send(producerRecord, new Callback() { public void on Completion(recordMetadata metadata, Exception e) { if(e != null) { //do something with this... retry, raise an alert, fail etc.... e.printStackTrace(); } else { //may notify clients that the record was written successfully //may record success System.out.println("Record written to offset: " + recordMetadata.offset()); System.out.println("Record written to partition: " + recordMetadata.partition()); } Next he goes up to the line props.put("request.required.acks",1); and changes the value 0 to 1. He draws attention to the line public void on Completion(recordMetadata metadata Exception e) { He goes back up to the line props.put("request.required.acks",1); and changes the value 1 back to -1 and highlights the line public void on Completion (recordMetadata metadata, Exception e) {.]*

So which type of acknowledgement you use for your application really depends on your business and your level of comfort with the potential durability or a lack thereof. So if you're after just pure speed and it's okay if some message gets lost, then you may want the acks to be 0. If you want…if you'll give up a little bit of speed for at least having some level of safety that the leader acknowledged it, then you may want to go with a 1. If you need to make sure that at all cost you have the highest level of durability, then -1 is most likely what you're looking for. So again, this is an example of using an async producer with acknowledgments.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Batching Messages

Learning Objective

*After completing this topic, you should be able to*

* *batch messages in Apache Kafka*

**1.**

In this segment, we're going to go over batch messaging with the async producer. Now I've got to set up some Properties here for some data - metadata.broker.list - which we need all the time. I've set the producer.type to be async. I am setting the request.required.acks to be 1. Again our options are 0, no acknowledgements; 1,  leader acknowledgement; and -1, acknowledgement from the leader in all in-sync replicas. And the next four Properties here all affect the batching. So first one is the queue.buffering.max.ms. The default value there is 5000. And this is the time, the maximum time that the producer will buffer data when using async mode. So say, for example, we set it to 100. It will try to batch together 100 milliseconds worth of messages to send it one time. So it may improve throughput, but it adds messages delivery latency due to the buffering. Okay so in this case, buffering.max.ms of 5000 milliseconds in 5 seconds. All the messages that you produce in 5 seconds will be buffered together.

*[The TextWrangler text editor is open. It has a menu bar and a details pane in which the 83238\_Apache\_Kafka\_AsyncAckProducer.java file is open. The following block of code is displayed: import java.util.Properties; import java.util.concurrent.ExecutionException; public class KafkaAsyncBatchProducerDemo { static void main(String  ] args){ Properties props = new Properties (); props.put ("metadata.broker.list", "localhost:9092"); props.put("producer.type","async"); props.put("request., required.acks", 1); props.put("queue.buffering.max.ms",5000); props.put("queue.buffering.max.messages",10000); props.put("queue.enqueue.timeout.ms",-1); props.put("batch.num.messages",200); KafkaProducer kafkaProducer = new KafkaProducer (props); ProducerRecord<byte [], byte[]> producerRecord = new ProducerRecord<byte [],byte []>("the-topic", "key".getBytes (), "value".getBytes()); kafkaProducer.send(producerRecord, new Callback () public void onCompletion(RecordMetadata metadata, Exception e) { if(e !=null) { //do something with this... retry, raise an alert, fail etc.... e.printStackTrace(); } else { //may notify clients that the record was written successfully //may record success System.out.println("Record written to offset: " + recordMetadata.offset()); System.out.println ("Record written to partition: " + recordMetadata.partition()); } } } }; } The presenter draws attention to each of the properties listed in the code.]*

The queue.buffering.max.messages is set at 10000. So this is the maximum number of unsent messages that could be queued up when the producer is in async mode before it either must block or data must be dropped. So you are going to get two choices there. So if you produce more than 10000 in 5 seconds, then you are either going to end up with messages being…with it being blocked, that it can't take more messages or it's going to have to drop messages. This queue.enqueue.timeout.ms, I set that to a -1. So that is the default as well. So this is the amount of time to block before dropping messages when the queue has reached that buffering.max.messages size. So if the queue is full, then it has to either…if it's 0, so that's set to 0. And the queue is full, it will immediately be dropped.

If it's set to -1, the producer will block indefinitely. And it's never willing to drop a send. Okay so we've it set to -1. It'll block indefinitely and never drop. If this is set to a 0, then it will either be enqueued immediately if we have not reached its max.messages or it will be dropped. And this is the bags batch, the batch.num.messages. And the default, which is set here, is 200. So this is the number of messages to send in one batch. Okay so the producer is either going to wait until this number of messages are ready to send or the queue.buffering.max.ms is reached, whichever comes first. So if we reach 5 seconds before we've produced 200 messages, however many number of messages have been produced, those are going to be sent. If on the other hand, we hit 201 messages or hit 200 messages and we've only gone 1 second, then that batch of 200 will be sent.

And again this only applies to the batching, which only applies when there is an async producer. Okay so here we've produced this record, and here is where we're using the async and getting that Callback. So again we had to produce 200 of the messages in less than 5 seconds, or less than 205 seconds elapse and a batch will be sent. So that is how we would set up and use the batch message sending feature of the async producer. And again these four Properties here - queue.buffering.max.ms, queue.buffering.max.messages, queue.enqueue.timeout.ms, and the batch.num.messages - are the four Properties that you would want to manipulate, as well as making sure you have an async producer when you want to use async producer and batch processing.

*[The presenter highlights the following lines of code: ProducerRecord<byte[],byte[]> producerRecord = new ProducerRecord<byte[]>("the-topic", "key".getBytes(), "value".getBytes( )); and public void onCompletion(RecordMetadata metadata, Exception e) {.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Keyed and Non-Keyed Messages

Learning Objective

*After completing this topic, you should be able to*

* *specify keyed and non-keyed messages in Apache Kafka*

**1.**

In this segment, we're going to discuss keyed and non-keyed messages with a KafkaProducer. So I go ahead here, I've a metadata.broker.list, one property that we always need for sure. And I've left all the other ones to be the default. I've constructed my KafkaProducer, and now I have three different records here. The first one I've named producerWithPartitionRecord. Now in this case, I have set the-topic is the first parameter. I set a 1 for the partition ID - partition number. I add a key. It will give me the bytes and the value. When I construct a ProducerRecord in this fashion, I am instructing the producer that I want it to send the data to this partition. It will ignore where the key may make the message land and will send it to the partition. This next  - producer with a keyed record - in this case, again the topic, always required. I sent the key or I set the key and the value. What will happen here since a partition is not used is it will be the hash of the key that's used to determine the partition that it's been written to.

*[The TextWrangler text editor is open. It has a menu bar and a details pane in which the 83238\_Apache\_Kafka\_AsyncAckProducer.java file is open. The following block of code is displayed:  import org.apache.kafka.clients.producer.KafkaProducer; import org-apache.kafka.clients.producer.ProducerRecord; import org.apache.kafka.clients.producer.RecordMetadata; import java.util.Properties; import java.util.concurrent.ExecutionException; public class KafkaKeyedProducerDemo { static void main(String [] args){ Properties props = new Properties ( ); props.put ("metadata.broker.list", "localhost:9092"); KafkaProducer kafkaProducer = new KafkaProducer (props); ProducerRecord<>producerWithPartitionRecord = new ProducerRecord<byte []>("the-topic", 1, "key".getBytes (), "value".getBytes()); ProducerRecord<>producerWithPartitionRecord = new ProducerRecord<byte []>("the-topic", "key".getBytes (), "value".getBytes()); ProducerRecord<>producerNoPartition0KeyRecord = new ProducerRecord<byte [],byte []>("the-topic", "value".getBytes ()); kafkaProducer.send(producerREcord).get(); } } The presenter draws attention to the section of code referring to the properties. He then points to sections of code relating to elements of the KafkaProducer:  ProducerRecord, producerWithPartitionRecord, and producerWithKeyRecord.]*

In this case, I have key and bytes. This is always going to return back the same byte. It's always going to return back the same hash code, which means that all of these messages would end up at the same partition. If you had a reason to, perhaps you were storing here, you had customer IDs, and you wanted all the customers to be processed together, then you could use the customerID for the key. That will ensure that all customer IDs go to the same partition. If you remember from some of our earlier segments, the level of parallelism with Kafka is the partition, and you are guaranteed to read all the data in a partition in order. So if you did have the customer IDs, then you could put those as a key if you want to process a customer's data, all in order. Or if you didn't want to and you just want it to be randomly distributed, perhaps you'd want to use a UUID or some other piece of data that would give you a nice distribution of hash codes so that the data gets spread across all the partitions. Keep in mind, using null, the same thing, right. If you had just a string of null and pass in the bytes, you are going to get it going to the same place.

*[The presenter continues to explain the function of producerWithKeyRecord and then types //customerID, UUID, null after "key".getBytes( ).  The complete line now reads as follows: ProducerRecord<> producerWithKeyRecord = new ProducerRecord<byte[],byte[]>("the-topic, 1,"key".getBytes(), //customerID, UUID, null. .]*

Now our third way that we could construct this ProducerRecord is NoPartitionOrKey. Again the topic required, and I just set the value. In this case, since there is no key or a partition, then the partition will be assigned in a round-robin fashion using all the partitions. Okay so really if I use the UUID here or some other random data, it's almost the same effect as if I didn't have the key at all because it's just going to be a round-robin, which should spread the data out fairly evenly as well. So those are the three different ways we can create this ProducerRecord and this is the way we could use keyed and unkeyed data. So keyed here that we're specifying, say, a customerID, so we're getting the data group together. Unkeyed - we don't pass in a key. And for all intents and purposes, this is almost unkeyed because we're setting a partition that we want the data to go to regardless of the key. So those are the three ways to construct a ProducerRecord to get the data to route to the right broker - a right partition - more importantly that you are interested in.

*[The presenter draws attention to the next string of code: ProducerRecord<> producerNoPartition0rKeyRecord = new ProducerRecord<byte[],byte[]>("the-topic,"value". getBytes()); He then highlights UUID, null elements from the producerWithKeyRecord line and deletes them. He points at the following elements, one by one: "key".getBytes(), //customerID from the producerWithKeyRecord line; "value".getBytes() from producerNoPartition0rKeyRecord line; and "the-topic",1,"key".getBytes() from the producerWithPartitionRecord line.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Broker Discovery

Learning Objective

*After completing this topic, you should be able to*

* *configure broker discovery in Apache Kafka*

**1.**

In this segment, we're going to discuss Kafka broker discovery. You know, sometimes you may not want to pass in a list of all the brokers that are part of your Kafka cluster. Perhaps you already have a connection to Zookeeper or it's easier just to pass in the Zookeeper connection string. You have a couple of options here if you want to do dynamic discovery of brokers. One is, there's a project out on GitHub, that's called kafka-broker-discovery. You can go ahead and clone this repo and build it. It works really simply by connecting to Zookeeper. You give it a zookeeperHost, the Port, and you call it KafkaBrokerDiscoverer. You give it that host and port and you turn around and get back the ConnectionString. And then you go ahead and close. So that will be one way to go ahead and get the broker ConnectionString. Now if you didn't want to use this source code, it's not available as of now that I know of on a Maven - public Maven repo. So you've to download it, build it, deploy it somewhere, or see what it's doing. If you want to do something similar yourself, it's really easy to do. Let's go ahead and I'll show you where you can get the same information that this is using to build that ConnectionString.

*[Two overlapping windows are open on the screen: the Chrome browser and the Shell interface. Chrome is active, and it displays the github.com web page for kafka-broker-discovery:  https://github.com/jstanier/kafka-broker-discovery.  It has introductory material and Usage and Command line usage sections. The following code is displayed in the Usage section: String zookeeperHost = "zookeeper1"; String zookeeperPort + "4444"; KafkaBrokerDiscover discoverer = new KafkaBrokerDiscoverer(zookeeperHost, zookeeperPort); discoverer.getConnectionString(); discoverer.close(); The presenter draws attention to the lines of code in the Usage section. Next he scrolls down the page to reveal the Command line usage section, which contains the following code: mvn clean install mvn dependency: copy-dependencies java -cp target/kafka-broker-discovery-0.0.1-SNAPSHOT.jar:target/dependency/\* com.brandwatch.ka.]*

You can see this other window. I have opened a Shell that's going to go. I am going to launch our zookeeper-shell. It's going to connect locally, so I am there. Okay if I go here, let's see if I look at evidence available in Zookeeper. You could see I have brokers. So if I do ls /brokers, there's the ids, there's the topics. Okay so ls /brokers/ids gives me id 0, which is a broker I have. I only have one broker that's running. So let's go ahead and get /brokers/ids/0. Okay so you could see the host and you could see the port that's listening on it. So if you wanted to do similar broker discovery dynamically on your own and you had a connection to Zookeeper where it was easier to get a connection to a Zookeeper core, then you could just iterate over the list of brokers in this ids path and you could go ahead and get the host and the port that it is listening on it.

*[The presenter clicks the Shell window, which contains the following code: quit Quitting... kafka\_2.10-0.8.2.0 -> clear kafka\_2.10-0.8.2.0 -> bin/zookeeper-shell.sh localhost:2181 He presses Enter, and the following lines are returned : Connecting to localhost:2181 Welcome to ZooKeeper! JLine support is disabled WATCHER:: WatchedEvent state:SyncConnected type:None path:null He then types ls and hits enter and then / and hits Enter, and a list of data is returned: ZooKeeper -server host: port cmd args stat path [watch] set path data [version] ls path [watch} delquota [-nl-b] path ls2 path [watch] setAcl path acl setquota -nl -b val path history redo cmdno printwatches on|off delete path [version] sync path listquota path rmr path get path [watch] create [-s] [-e] path data acl addauth scheme auth quit getAcl path close connect host:port. The presenter types ls / and presses Enter, and the following message is returned: [controller\_epoch, controller, brokers, zookeeper, admin, consumers, config]. He then types ls /broker and presses Enter, and   [ids, topics] is returned. Next he types ls /brokers/ids and presses Enter, and a value of  [0] is returned. The presenter types get /brokers/ids/0 and presses Enter, and the following data is returned: {"jmx\_port":-1, "timestamp":"1435624656719","host":"192.168.1.66","version":1,"port":9092} cZxid = 0x97 ctime = Mon Jun 29 18:37:36 MDT 2015 mZxid = 0x97 mtime = Mon Jun 29 18:37:36 MDT 2015 pZxid = 0x97 cversion = 0 dataVersion = 0 aclVersion = 0 ephemeralOwner = 0x14e41e514bc0000 dataLength = 89 numChildren = 0]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Kafka Test Suites

Learning Objective

*After completing this topic, you should be able to*

* *use Apache Kafka test suites for testing*

**1.**

In this segment, we're going to talk about some of the Kafka testing utilities that exist in the code base that you could use to make your testing with Kafka producers and consumers much easier. Now again this is not the end-all, be-all. You could easily do something similar but this is a great start. So let's first start with using a MockProducer. So I'll go ahead and set these Properties. Really I don't need them in this case. So I probably could just do this, just get rid of those. Okay so there is a MockProducer, which doesn't care about them. So I can go ahead and create this MockProducer(true) as we'll see and we'll go over the source code for that in a moment. It tells it whether or not to autocomplete. We go ahead and create the record, just like we normally would. Again I'm just using key and value the text for my key, the text for my value. I am going to go ahead and call producer.send and get back the metadata.

*[The IntelliJ IDEA application is open, running the KafkaTestingDemo.java - kafka-producer-demo file. It has three tabs: KafkaTestingDemo.java, MockConsumer.java, and MockProducer.java. The KakfaTestingDemo.java tab is open, and it displays two sections containing code. In the first section, the following code is presented: import org. apache.kafka.clients.producer.ProducerRecord; import org. apache.kafka.clients.producer.RecordMetadata; import java.util.Properties; import java.util.concurrent.ExecutionException; import java.utilconcurrent.Future import static org.junit.Assert.\*; public class KafkaTestingDemo { static void main(String[] args) throws Execution Exception, InterruptedException { Properties props = new Properties(); props.put("metadata.broker.list", "localhost:9092"); props.put("producer.type","sync"); String topic = "myTopic"; MockProducer producer = new MockProducer (true); ProducerRecord<byte [ ], byte [ ]> record = new ProducerRecord<byte [ ], byte [ ]> (topic, "key".getBytes (), "value".getBytes ()); assertTrtue ("Send should be immediately complete", metadata,isDone ()): assertFalse ("Send should be successful", isError(metadata)); assertEquals("Offset should be 0", 0, metadata.get () offset()); assertEquals(topic, metadata,get().topic()); } The following code is displayed in the second section: static boolean isError(Future<?> future) { try { future.get (); return false; } catch (Exception e) { return true: } } } The presenter highlights and deletes the section of code relating to properties: Properties props = new Properties(); props.put("metadata.broker.list", "localhost:9092"); props.put("producer.type","sync");. The first four lines of code now reads as follows: import org.apache.kafka.clients.consumer.MockConsumer; import org.apache.kafka,clients, consumer.MockProducer; import org.apache.kafka.clients, consumer.ProducerRecord; import org.apache.kafka,clients, consumer.RecordMetadata; He then draws attention to the block of code in the first section relating to the MockProducer.]*

Now I told this the autocomplete is true, so I get this assertTrue here that the send should automatically complete and it should complete immediately, so I can check that isDone(). I could also see that it's successful - this is what is Error function - which just does the future to try and do a get(). If that…remember if that get() fails, it's either going to return back the value or it's going to throw. Okay so if this get(), it's either going to return back a value or it's going to throw an Exception, in which case it will return true. If we don't get() an Exception, return back false, it's not an error. And the Offset should be 0. Okay so we get an offset() from it. The topic should be the same topic that we wanted. Okay let's take a quick look at that MockProducer code. Again this is right from the Kafka code base. Okay and this is using Kafka 8.2. So you could see here we could pass in a Cluster to it. And we could tell it to autoComplete or just tell it to autoComplete and it won't use any Cluster. It will just have an empty() list. Or you could pass it nothing and it will automatically do the same thing of autoComplete feature.

*[In the KakfaTestingDemo.java file, the presenter draws attention to following lines of code: assertTrue("Send should be immediately complete", metadata.isDone()); assertFalse("Send should be successful", isError(metadata)); He then points to the following lines in the second section of code: future.get(); return false; }catch (Exception e) { return true; } He next points out the following code: assertEquals("Offset should be 0", 0, metadata.get().offset()); and assertEquals(topic, metadata.get().topic());. The presenter clicks the MockProducer.java tab to open it. It contains MockProducer base code in different sections. He draws attention to the following lines of code, one by one: public MockProducer(Cluster cluster, boolean autoComplete) {...}, public MockProducer(boolean autoComplete) { this(Cluster.empty(), autoComplete); }, and public MockProducer() { this(true); }.]*

Some of the things that are nice, right, we saw the send function. But what's nice about it that you could do is you could tell it, you know, it's going to give you nextOffset, the partitions, the metrics(). Where it starts to help you is you could tell it whether or not to complete the next one. Okay so you could tell it that you want to errorNext. So you could tell it to do a variety of things, so like error the Next function. So when you do call and say you're going to send a message, it will error the very next one. So you can work on your error handling of what happens when the KafkaProducer does throw an error. Okay so this is the MockProducer, which could come in handy. Maybe it doesn't fulfill all of your needs, but it's a good start and gives you an idea of how you could mock the producer and may be a good, you know, starting ground for what your MockProducer may be if you need something slightly different.

*[The presenter points out the send function in the following line of code: public synchronized Future<RecordMetadata> send(ProducerRecord<byte[], byte[]> record) {. He then scrolls down the page and draws attention to following lines of code: private long nextOffset(TopicPartition tp) {....}, public List<PartitionInfo> partitionsFor(String topic) {return this.cluster.partitionsForTopic(topic); }, and public Map<MetricName, Metric> metrics() { return Collections.emptyMap(); } He scrolls down further and points out the following lines: \*Complete the earliest uncompleted call with the given error. \* \*@return true if there was an uncompleted call to complete \*/ public synchronized boolean errorNext(RuntimeException e) {.]*

Now just like there's a MockProducer, there's a MockConsumer. You could do some interesting things with it as well. Subscribe to topics, a list of topics. You could poll, okay, so that you want to go and read a message, and in this case, if you look at this code, it's just going to go ahead and hand back a dummy record, one per topic you registered for. So a nice little, you know, helper here. You can tell it to commit all sorts of things, right, to seek to somewhere. It implements that whole consumer interface except it's not really going anywhere. So now you could put your code in place that uses a MockConsumer. It allows you to isolate your business logic. Again,  this may not have everything you need but hopefully would give you a good starting point and some good ideas of how you may want to build a MockConsumer. So these are at least two of the testing facilities built into Kafka that you could use. And if you want to do more sophisticated stuff with, say, Zookeeper, you'll find that there are in-memory Zookeeper quorums you could use and other things to help you mock the whole environment. So again these are two ways that you could help test your Kafka code: one with a MockConsumer and one with a MockProducer.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Serialization

Learning Objective

*After completing this topic, you should be able to*

* *configure serialization and deserialization in Apache Kafka*

**1.**

In this segment, we're going to go over serializing Json with the KafkaProducer. So what I have here is a project that has this KafkaJsonDemo, the JsonEncoder, and JsonDecoder. Let's walk through this. So remember when it comes down to it and data goes across the wire, KafkaProducer is going to end up sending a byte array. So there is a byte array for the keys and a byte array for the value. And I should really say a byte array for the key, not the keys. So the key is a byte array and the value is a byte array. Now perhaps you have an application where you're passing Json around and you don't want to end up having to convert that Json to a byte array all the time in your code. It looks messy. It may not be as clean or elegant as you want. So what you could do is you could implement the Encoder interface. So here I've this JsonEncoder. I'm going to use a Jackson ObjectMapper. You could use any Json library you want, it really doesn't matter. I chose to use Jackson in this case.

*[The IntelliJ IDEA application is open, running the KafkaTestingDemo.java - kafka-producer-demo file. It has a menu bar, a navigation pane on the left-hand side, and a details pane. The navigation pane displays two main folders: kafka-producer-demo and External Libraries. The kaka-producer-demo has two subfolders: .idea and src. The src folder is expanded to display main and test folders. The main folder has a java subfolder with three files: JsonDecoder, JsonEncoder, and KafkaJsonDemo. There are three tabs in the details pane, KafkaJsonDemo.java, JsonEncoder.java, and JsonDecoder.java. The KafkaJsonDemo file is open, and it displays two sections of code. The first section contains the following code: import com.fasterxml.jackson.databind.ObjectMapper; import org.apache.kafka.clients.producer.KafkaProducer; import org.apache.kafka.clients.producer.ProducerRecord; import java.util.Properties; public class KafkaJsonDemo { private static final ObjectMapper objectMapper = new Object Mapper ();. The second section contains the following lines: static void main(String[ ] args) throws Exception { Object jsonObject = objectMapper,.readTree("\"myvalue\":\"my first value\"}"); Properties props = new Properties (): props.put("serializer.class", "JsonEncoder"); KafkaProducer kafkaProducer = new KafkaProducer (props); kafkaProducer.send(new ProducerRecord<byte[ ],Object>("the-topic", "key".getBytes (), JsonObject )); } }. The presenter selects the JsonEncoder folder in the navigation pane, and the corresponding tab becomes active in the details pane. It contains three sections of code. In the first section, the following code is displayed: import fasterxml.jackson.core.JsonProcessingException; import com.fasterxml.jackson.databind.ObjectMapper; import kafka.serializer.Encoder; import kafka.utils.VerifiableProperties; public class JsonEncoder implements Encoder<Objects> { private static final ObjectMapper objectMapper = new Object Mapper ();. The second section contains the following string: public JsonEncoder(VerifiableProperties verifiableProperties) { /\* This constructor must be present for successful compile. \*/ }. The third section contains the following code: public byte[ ] toBytes(Object object) { try{ return objectMapper,writeValueAsString(object).getBytes (); } catch (JsonProcessingException e) { e.printStackTrace(): } return "".getBytes(); } }. The presenter points out the following code from the first section: public class JsonEncoder implements Encoder<Object> { private static final ObjectMapper = new ObjectMapper ();]*

It's going to take the VerifiableProperties that must be present just for implementing this interface; what the constructor needs. And then you have this toBytes method. This is going to take in an Object and in this case, I am just going to turn around. I am going to writeValueAsString and getBytes(). Now obviously this may not be Json. So you've this Exception that may occur, printStackTrace(), and we end up producing an empty message. It may not be exactly how you want to handle error conditions. It's purely just an example. The key is this toBytes method. It takes an Object, returns a byte array. The way that we'll use this is in our producer setup, and again this could be a config file, your environment, or in code, you have the serializer.class. And you put the fully qualified class name of that class. In my case, there's no package name. It's just right here - JsonEncoder.

*[In the JsonEncoder file, the presenter draws attention to the following line from the second section of code: public JsonEncoder(VerifiableProperties verifiableProperties) {. Next he draws attention to the third section of code. He then selects the KafkaJsonDemo file in the navigation pane and the corresponding the KafkaJsonDemo.java tabs becomes active again. He draws attention to the second section of code: Object jsonObject = objectMapper.readTree  ("{\"myvalue\":\"my first value\"}"); Properties props = new Properties(); props.put("serializer.class" "JsonEncoder"); KafkaProducer kafaProducer = new KafkaProducer(props); KafkaProducer.send(new ProducerRecord<byte[],Object>("the-topic", "key".getBytes(), jsonObject)); } }.]*

If it was a separate jar, again a fully qualified name and you'd want to have the jar on the path. When this is set up and I now call this ProducerRecord and I call send, what's going to end up happening is the KafkaProducer will end up calling the toBytes method on my encoder. So now I can go ahead and I can just pass this jsonObject in to the producer all the time. Okay so say, for instance, this was happening somewhere else in my code and this is all set up. I could just be passing in Json and not worrying about it. My encoder will take care of it. Now there's going to be times when you're going to want to read the data and it's in Json. You have two choices there. So you're going to give back a byte array where you're consuming data from Kafka. You could in turn just convert it to a jsonObject or you could use Decoder where you would do the opposite from bytes, and you go ahead and decode it, again returning the Object. So a very similar way, you get the encoding going in, the decoding coming out. So that is how you could work with Json using KafkaProducer and the consumer.

*[Still describing the second section of code in the KafkaJsonDemo.java tab, the presenter points out the following code: KafkaProducer.send(new ProducerRecord<byte[],Object>("the-topic", "key".getBytes(), jsonObject));. He then highlights the entire first line: Object jsonObject = objectMapper.readTree  ("{\"myvalue\":\"my first value\"}");. The presenter selects the JsonDecoder file in the navigation pane, and the corresponding JsonDecoder.java tab becomes active in the details pane. Two sections of code are displayed. The first section contains the following lines: import com.fasterxml.jackson.databind.ObjectMapper; import kafka.serializer.Decoder; import java.io.IOIException; public class JsonDecoder implements Decoder<Object> { private static final ObjectMapper objectMapper = new ObjectMapper ();. The second section contains the following code: public Object fromBytes(byte[] bytes) { try { return objectMapper.readTree(bytes); } catch (IOException e) { e.printStackTrace(); return null; } } }. The presenter draws attention to the second section of code.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Building a Custom Serializer

Learning Objective

*After completing this topic, you should be able to*

* *build a custom serializer in Apache Kafka*

**1.**

In this segment, we're going to discuss configuring serialization and deserialization in Apache Kafka with Avro. There's no built-in Avro serializer and deserializer with Kafka. But you have a couple of options. Most likely you are considering using Avro because of the schema support it provides and probably in particular, the schema evolution, which is a fantastic feature of Avro where you could evolve schemas of the data as your data changes as your business changes. If that's the case, perhaps an option, maybe one of the best options at this time for you is to use the Confluent IO Platform, which provides Avro Serializers and a Schema Registry that satisfies most likely all the needs you'll have for schema evolution.

*[The Chrome browser is open. It has a menu bar and an Application Development tab open at : confluent.io/docs/current/app-development.html. The Confluent page contains a navigation pane on the left-hand side and a details pane. The navigation pane contains a Confluent Platform search box and the following headings: What is the Confluent Platform, Data Serialization and Evolution, Installation, Confluent Quickstart, Application Development, Kafka Operations, Schema Registry, Kafka REST Proxy, Camus, and Operations. Application Development is expanded to display two options, Java Applications: Serializers and Non-Java Applications: REST Proxy. The Java Applications: Serializers page is open in the details pane.]*

So that's one way to go. If you went that route, here is where you'd find the documentation under the Application Development where you could configure for the serializer, the confluent.kafka.serializers.KafkaAvroSerializer. And you would have the same thing for a deserializer where it would be KafkaAvroDeserializer. If you use those - serializer and deserializer - it will allow you to pass in Avro data that you have in hand to the producer directly and allow the consumer to deserialize while Kafka returns to the byte array the Avro data and give it back to you. This handles all the interaction with the Schema Registry and everything to do with decoding Avro data to a byte array and basically reversing a byte array into Avro data.

*[The presenter draws attention to the following information on the Java Applications: Serializers page on the Confluent site: Java applications can use the standard Kafka producers and consumers, but will substitute the default ByteArraySerializer with io.confluent.kafka.serializers.kafkaAvroSerializer (and the equivalent deserializer), allowing Avro data to be passed into the producer directly and allowing the consumer to deserialize and return Avro data.]*

And all you do with this is you include these dependencies and again the versions may change by the time you're going to use this. So make sure you do check to see if it's updated. And then you set up some Properties that you want the serializer class to be the KafkaAvroSerializer and you want the value to be the serializer. And you set up the schema.registry.url, that really is it. And once your producer is set, then it's really as simple as sending data. Okay so here is an example they have. Give this User, which may be a Java Poser that you have, that's in Avro. You setName, you call producer.send with that User. That's it. It couldn't be any simpler. If this does not meet your needs and you have another reason for using Avro, may be your schema is a fix, maybe you don't need the evolution or the registry, then similar to what we did with our Json encoding and decoding, you could just use Avro decoder and an Avro encoder. Either way, you have choices of using the Confluent IO Platform or building your own encoder and decoder.

*[The presenter refers to the following dependencies, which are listed on the on the Java Applications: Serializers page on the Confluent site: <dependency> <groupId>io.confluent</groupId> <artifactId>kafka-avro-serializer</artifactId> <version>1.0</version> </dependency> <dependency> <groupId>org.apache.kafka</groupId> <artifactId>kafka\_2.10</artifactId> <version>0.8.2.0-cp</version> <scope>provided</scope> </dependency> He scrolls down the page and draws attention to the following three properties: props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG,io.confluent.kafka.serializers.kafkoAvroSerializer.class); props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,io.confluent.kafka.serializers.kafkoAvroSerializer.class); props.put(“schema.registry.url”, http://localhost:8081");. He then scrolls further down the page and explains the following lines of code: User user1 = new User(); user1.setName("Alyssa"); user1.setFavoriteNumber(256); Future<RecordAndMetadata> resultFuture = producer.send(user1);.]*

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

Exercise: Creating a Producer and Consumer

Learning Objective

*After completing this topic, you should be able to*

* *configure a broker and create a producer and a consumer in Apache Kafka*

**1.**

**Exercise Overview**

In this exercise, you'll configure a broker for Apache Kafka. You'll create and configure a producer in Apache Kafka and you'll create and configure a consumer in Apache Kafka. Now go ahead and pause the video and perform the exercise. And when you come back, we'll go over a possible solution that I have.

**Solution**

In this exercise, the first step you are supposed to do was to configure the Apache Kafka broker. Now here's the configuration I have. It looks similar to what comes out of the box. Did you remember about a broker.id? In this case, I chose 0. I could easily have chosen any number I wanted to. What about the other configurations that are important to remember? The log.dirs, remember here. I have mine going to /tmp/kafka-logs. Again and that could be anywhere you want it to go. The last configuration that's really important is zookeeper, alright. We'll find that here, then we tell it where to go. In this case, it's going to localhost. You could configure that to go to any server, and you could have configured it to go to a number of different hosts. The second step was to configure a producer.

*[The TextWrangler Window is open, and it has a menu bar, a navigation pane, and a details pane. The navigation pane shows Currently Open documents, displaying three files: 8324x\_Apache\_Kafka\_Exercise\_Consumer, 8324x\_Apache\_Kafka\_Exercise\_Producer, and server.properties. Server.properties is selected, and the server.properties file is open in the details pane. It lists server properties under such headings as Log Basics, Log Flush Policy, and Socket Server Settings.  The presenter scrolls up the page to the Server Basics section and points out broker.id=0. He moves to the Log Basics section and points out log.dirs=/tmp/kafka-logs. He then moves to the Zookeeper section and points out zookeeper.connect=localhost:2181.]*

So I have my producer here. I have these Properties. Again you could have used a Properties file or you could take these values from your environment. However you want to get them, it's perfectly valid as long as you have a Properties object to hand to the Producer. And then again, I'm just simply producing a record to the-topic with the bytes of key as my key and the bytes of the string value as my value. And then when I'm done, I am closing it. Hopefully you got close to something like this or may be try some other different ways you could create and use a producer. The last step was creating and configuring a consumer. So my consumer here is a little bit more elaborate. We'll start from the bottom here. I just have this main. I pass in zooKeeper, pass in that groupId I want to be part of, pass in the topic, pass the number of threads I want to create. Now a quick question for you. Did you remember the groupId? Alright. Each consumer is always part of a group. But you could have n number of consumers belonging to a single group, which is how you could allow your consumers to scale out their consumption of the topic.

*[The presenter selects the 8324x\_Apache\_Kafka\_Exercise\_Producer file in the navigation pane, and the file opens in the details pane displaying the following code: import org.apache.kafka.clients.producer.KafkaProducer; import org-apache.kafka.clients.producer.ProducerRecord; import java.util.List; import java.util.Map; import java.util.Properties; public class KafkaFinalProducerDemo { static void main(String [] args){ Properties props = new Properties (); props.put ("metadata.broker.list", "localhost:9092"); KafkaProducer kafkaProducer = new KafkaProducer (props); kafkaProducer.send(new ProducerRecord<byte[]>("the-topic", "key".getBytes(), "value".getBytes())): //use the producer and then close when done kafkaProducer.close(); } }. The presenter draws attention to the lines of code relating to properties and KafkaProducer. He selects the 8324x\_Apache\_Kafka\_Exercise\_Consumer file in the navigation pane, and the file opens in the details pane. displaying multiple blocks of code. He scrolls down the page and draws attention to the following lines of code: public static void main(String[] args) { String zooKeeper = args[0]; String groupId = args[1]; String topic = args[2]; int threads = Integer.parseInt(args[3]); KafkaConsumerExample example = new KafkaConsumerExample(zooKeeper, groupId, topic);.]*

So you got the groupId. And I just create this example I have as I sit there and I sleep, waiting for it to run, and then check down. My example is nothing fancy. I created JavaConsumerConnector. I am going to createConsumerConfig. Just really, I want to just hide this shutdown() for now, it's not important. My Config, just the Properties; again a groupId, which is important; return back ConsumerConfig, that creates for me my consumer. And then I just go ahead and I'm going to run the number of threads, get my topicMap, get the count. I go out and create the streams that I want and I launched the streams or get a List of streams, if you will, tell my executor to spin off that number of threads that were passed in, and then I just sit there and I submit them and that threadNumber. Okay I am away, I go consuming.

*[Still in the 8324x\_Apache\_Kafka\_Exercise\_Consumer.java file, the presenter comments on the following block of code: example.run(threads); try { Thread.sleep(10000); } catch (InterrupedException ie) { } example.shutdown(); }. He then scrolls up the screen and draws attention to the following code: public KafkaConsumerExample(String zookeeperConn, String groupId, String topicName) { consumer = kafka.consumer.consumer.createJavaConsumer Connector( createConsumerConfig(zookeeperConn groupId)); this.topic = topicName; }. He collapses the set of code relating to public void shutdown() {. He then draws attention to the following set of code: private static ConsumerConfig createConsumerConfig(String zookeeperConn, String groupId) { Properties props = new Properties(); props.put("zookeeper.connect", zookeeperConn); props.put("group.id", groupId); return new ConsumerConfig(props); }. Next he points out each of the following lines: public void run(int a\_numThreads) { Map<String, Integer> topicCountMap \_ new HashMap<String, Integer>(); topicCountMAp.put(topic, new Integer(a\_numThreads)); Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCountMap); List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic); //now launch all the threads // executor = Executors.newFixedThreadPool(a\_numThreads); //now create an object to consume the messages // int threadNumber = 0; for (final KafkaStream stream : streams) { executor.submit(new ConsumerTest(stream, threadNumber)); threadNumber++; } }.]*

Perhaps you got something that's similar to this, maybe a slightly different implementation. It's just fine. Some of the key things that you want to remember: you need a topic, you need a groupId, and you need the zookeeper. If you want the groupId, you'll get one randomly assigned. So you're going to want one. And you definitely need a zookeeper or you can't connect and know where the partitions are for the topics that you're interested in reading. So hopefully this exercise worked out well for you to configure the broker, configure and create a producer, and configure and create a consumer. With all of that up and going, local zookeeper running, local broker running, you should be able to produce and consume data from Kafka.

[Back to top](https://library.skillport.com/courseware/Content/cca/df_apka_a02_it_enus/output/html/course_transcript.html#top)

© 2018 Skillsoft Ireland Limited