5. Producing Messages with Kafka Producers

Contents

[1. Introduction and Setting up an Apache Kafka Development Environment 1](#_Toc129102271)

[2. Basics of Creating an Apache Kafka Producer 1](#_Toc129102272)

[3. Creating and Preparing Apache Kafka Producer Records 1](#_Toc129102273)

[4. Apache Kafka Producer Record Properties 2](#_Toc129102274)

[5. The Process of Sending Messages, Part One 2](#_Toc129102275)

[6. The Process of Sending Messages, Part Two 2](#_Toc129102276)

[7. Message Buffering and Micro-batching 2](#_Toc129102277)

[8. Message Delivery and Ordering Guarantees 2](#_Toc129102278)

[9. Demo: Creating and Running an Apache Kafka Producer Application in Java 2](#_Toc129102279)

[10. Advanced Topics and Module 4 Summary 2](#_Toc129102280)

# 1. Introduction and Setting up an Apache Kafka Development Environment

=>slides: Pg. 1

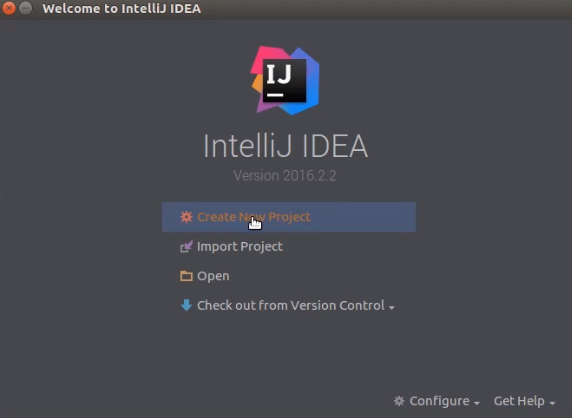
In this module, we'll be getting into more details about the Apache Kafka producer. In reality, we've covered quite a bit about what the producer does. Here, we'll look at how it does it and what resources are available to developers to write applications that publish messages to Kafka. I will walk through how to build your own Kafka producer and spend some time covering some important configuration properties that affect the message sending behavior.

=>slides: Pg. 2

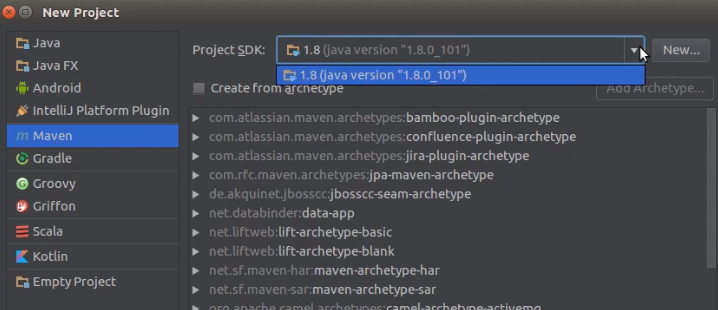
As we did in the last module, the first thing I'd like to do is get the setup out of the way. By setup, I mean specifically getting a development environment established to develop Apache Kafka producer and consumer applications. Getting a development environment set up is really straightforward. We will essentially just add the Apache Kafka client libraries using a dependency manager and import the packages into the environment. Once we've validated that the dependency manager has properly imported the packages, we will then briefly walk through the API. For a successful setup and subsequent exercises, you will need to have a standard integrated development environment such as JetBrains IntelliJ or Eclipse. There is a free version of IntelliJ called the Community Edition, and, of course, Eclipse is free as well. In this course, I will show my bias for IntelliJ. Aside from some of the user interface differences, the process should be more or less the same for getting set up in Eclipse. It should go without saying, but you'll need the latest Java JDK. Currently, the latest is Java 8. You should have the Maven dependency manager installed and plugged into your IDE to make things easy. I recommend version 3 of Maven. While not required for development per se, you should have access to a test Kafka cluster. By that, I mean at least one running Kafka broker. This will enable you to test the producer and consumer applications you build end to end. In this course, you'll notice I am developing client applications within the same virtual machine for simplicity.



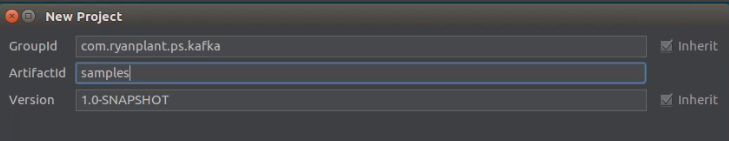
So the first thing is to launch the IDE itself.



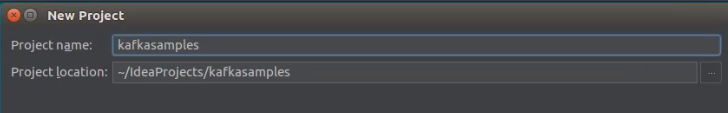
Next, select Create New Project.



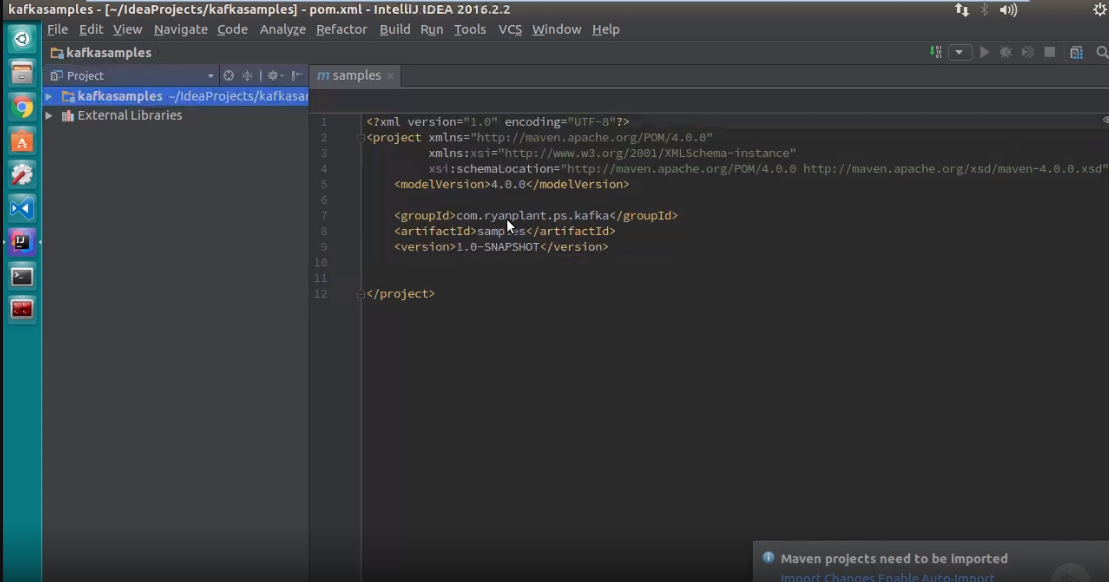
Select Maven. If you haven't already, select your project SDK or add one. This is where you should specify the latest version of the Java JDK, Java 8 preferably. Select Next.



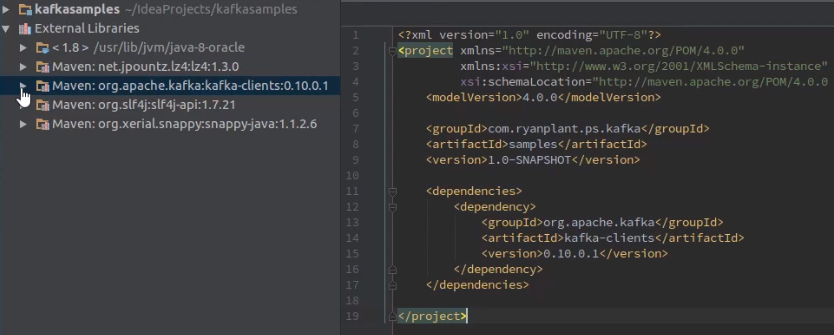
This is where you will add your own details related to your Kafka applications project.



When you're finished, you continue through and hit Finish.

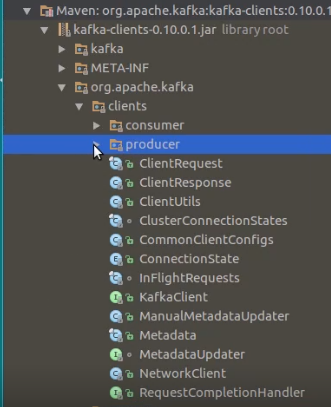


With your project open, navigate to your project's POM file. Here we will need to add the Kafka dependencies so Maven can import the packages into the project.



Now let's add the dependencies. This is pretty typical, but what you're going to do here is you're going to want to add the org.apache.kafka as the groupId, kafka‑clients as the artifactId, and then the version is going to be 0.10.0.1 Now let's take a look at the API we'll be using in this and subsequent modules.

If the version number appears red in color, it means the user missed to enable the '**Auto-Import**' option. If so, go to **View > Tool Windows > Maven**. A Maven Projects Window will appear on the right side of the screen. Click on the 'Refresh' button appearing right there. This will enable the missed Auto-Import Maven Projects. If the color changes to black, it means the missed dependency is downloaded.



Next, locate and expand the kafka‑clients external library node. Next, expand the org.apache.kafka node. Here you'll see the clients and common namespaces. Feel free to browse the common namespace, but we'll be going into the clients namespace for most of the time. In the clients namespace, you'll find all of the objects you'll be working with directly in creating either producer or consumer clients. For producer development, you'll be using the Producer namespace. For the consumer development, you'll use the Consumer namespace. The Producer namespace, you'll see the main classes and interfaces will be looking at,



particularly the KafkaProducer class and the ProducerRecord class.

**=**>slides: Pg. 3

So far, we've really just focused on the Apache Kafka producer externals, the high‑level component view of a producer and its interaction topology with the rest of the Kafka cluster of brokers.

**=**>slides: Pg. 4

Now let's explore the high level of what goes on within the producer. Since the producer is a piece of software, what we'll cover here is a logical representation of the key components and how they work together to send messages. We'll use this as a map to go further into each component throughout this module. Admittedly, this is a busy diagram, so let's go into it piece by piece and cover what each component does and at what point in the producer lifecycle they come into the picture.

# 2. Basics of Creating an Apache Kafka Producer

**=**>slides: Pg. 5

When creating a Kafka producer client application, you'll first need an object to represent the required configuration properties needed to start up a producer. As indicated here, there are three required properties needed, bootstrap.servers and both key and value serializers. Let's take a look at the code and describe them a bit more.

**=**>slides: Pg. 6

Configuration items are generally key‑value pairs, so to construct a dictionary of key‑value pairs that represent the configuration settings for the Kafka producer, the easiest way to do it is to use the Properties class from the core java.util's library. You'll recall from the previous module that when we used the Kafka producer shell program, we simply needed to supply a list of brokers for the producer to connect to. This corresponds to the bootstrap.servers configuration setting needed for the producer to start up. The producer doesn't connect to every broker referenced in this list, just the first available one. It uses the broker it connects to for discovering the full membership of the cluster, which, of course, can change at any time. It uses this list to determine the partition owners or leaders so that when it's ready to send messages, it can do so immediately. It is a best practice to provide more than one broker in the broker list in the unlikely event that the first broker specified is unavailable. Next is the key and value serializers. If you recall in the last module, I mentioned how the message content is encoded as binary. This is to optimize the size of the messages not only for network transmission, but for storage and even compression. Since it is the producer that serves as the beginning of a message's lifecycle, it is responsible for describing how the message contents are to be encoded so the consumer can know how to decode them. In this example, you'll notice that for both the key and value.serializer, we're using the StringSerializer class, which is the most common serializer scheme used in Kafka. You're probably wondering what is meant by a key and value, and why are they so important that a producer requires their serialization strategy to be established up front? That's a good question. We'll get to that shortly. These are but three of the many configuration settings that can be set. For a full list of settings, always refer to the producerconfigs section of the Kafka documentation site. We will cover more of the important but optional settings as we continue.

**=**>slides: Pg. 7

Like any standard application, you need to have an entry point. In this case, we'll be hosting a Kafka producer within a standard Java console application, and the boilerplate code for this should be evident. Here, you'll see the creation and setting of the Properties dictionary for configuration items, as described in the last slide, followed by the primary class instantiation statement that makes this generic console application an actual Kafka producer application. There are different approaches to writing this instantiation statement, but this is by far the most simple. The other options are based on what values you want to provide to the instantiation and parameters you would pass to the constructor. By exploring the documentation and writing applications, you'll get to know these options on your own.

**=**>slides: Pg. 8

When instantiating a Kafka producer with a Properties object, as illustrated in the last slide, you are effectively setting things up for the Kafka producer to start sending messages with the basic defaults. In our case, we instantiated an object of type KafkaProducer and called it myProducer and passed it a properties object named props. If you look inside the implementation of the KafkaProducer, you will notice a type called ProducerConfig. When the KafkaProducer object is created, the properties are used to instantiate an instance of the ProducerConfig class, and from there, all producer configuration is defined and referenced internally. It is from this object that the internal fields for key and value.serializer are initialized. So, when providing these values in the Properties object, you're indirectly, through the ProducerConfig, setting the internal fields of the producer to expect message values for the key and value of type string. This is essentially establishing a type‑safe contract between the instance of the KafkaProducer and the message specifications it is configured to produce. This contract extends to the consumer who, when reading messages from a topic, needs to know the message specifications and its type contract, which is why the configuration properties are required from the onset. This is good, but all we've really done here is create a KafkaProducer object with its default settings, and that's it. Not very exciting, right? A producer's job is to produce. What does it produce? Messages, of course. So let's get to that next.

# 3. Creating and Preparing Apache Kafka Producer Records

**=**>slides: Pg. 9

From the point of view of the Kafka Producer, it doesn't really send messages. In fact, you won't find a single type in the entire Kafka API called message. What you will find is a critical class called ProducerRecord, and it represents what will be published by the Kafka Producer. A producer record is also fairly basic and straightforward, it only requires two values to be set in order for it to be considered a valid record that can be sent by the Kafka Producer. These two values are the topic and the value. The other optional values of partition, timestamp, and key will be covered shortly. Let's take a closer look.

**=**>slides: Pg. 10

It doesn't take much to actually get messages flowing to Apache Kafka. You saw how simple it was using the producer shell application, that's because the API was designed to require the bare minimum to get started. This is represented not only by the simplicity of getting a Kafka Producer instantiated, but now also the producer record. The first required value should be self‑explanatory at this point, it's the topic to which these record is destined. The value is really just the contents of the message that are to be serialized using the specific serializer in the configuration settings.

**=**>slides: Pg. 11

In the last module when we used the Kafka Producer Shell Program, you'll recall we just had to specify two parameters, the broker List and the topic. By taking in these values, the Kafka Producer was setting its own configuration properties for Bootstrap servers and then taking the topic value to set the required topic filled in the producer record. In the Shell program, it hardcoded the default serializer to be a string serializer class. The message provided in the input stream from the terminal provided the values for the Kafka Producer to send to the broker.

**=**>slides: Pg. 12

Back to that last property, the reason it is called the value is because it must correspond to the serializer type specified in the configuration properties for the Kafka Producer instance. If you were to try and create a producer record that didn't match the serializer type specification for the producer, the producer would generate a runtime serialization exception, stating the type provided doesn't match what was expected as per the value.serializer property.

**=**>slides: Pg. 13

When you define and instantiate a Kafka Producer, you are doing so with the type of messages it will send. This is established up front with the requirement of setting the key and value serializers. Kafka Producers send very specific producer records, and the type specification of the key and value must match that of the producer that is going to send it. Trying to attempt otherwise will cause exceptions to be thrown and nothing will get sent to the Kafka cluster. This is something to keep in mind as you're designing your Kafka Producer applications. Initially, you may think of this as a limitation, but given all of the other configuration properties that you can set on a per Kafka Producer basis, having the delineation between different Kafka Producer instances for very specific categories of messages, in other words, topics, it isn't that limiting at all, but rather a powerful ability, allowing you to have per topic flexibility at the producer level.

# 4. Apache Kafka Producer Record Properties

**=**>slides: Pg. 14

I told you I would cover the optional properties, so here are two of them, partition and timestamp. Yes, the partition refers to a specific partition within a topic. When creating a producer record, you can set the value of this to a specific partition that you want messages to be sent to. Doing this is an advanced scenario, but an important one when it comes to how the producer decides which broker to send its messages to. Hang in there for a minute and we'll get back to this. The timestamp is a new addition to Kafka, starting in the current .10 version. It allows for the explicit setting of a timestamp to the producer record. Its presence is somewhat controversial because the timestamp is transmitted with the message. And since it is a long data type, it carries with it the additional overhead of 8 bytes, which can affect performance and throughput in high‑volume situations. This property is nuanced, however, because the actual timestamp that will be logged for a message will be based on settings defined in the broker server.properties file, specifically the log.message.timestamp.type setting. There are two modes available for determining which timestamp the message should have. If the setting is CreateTime, which is the default, the timestamp applied to the message is set by the producer and will be what is committed to the log. It doesn't matter if you choose to set this timestamp explicitly. Even if you ignore this setting, the producer will automatically apply the timestamp to every outgoing message. These alternative mode is LogAppendTime, which will overwrite whatever the timestamp is coming from the producer with the timestamp of the broker at the time the message is appended to the log. From a design standpoint, the mode for which to establish the message's time is not a trivial matter. The ability to establish time, where and what to do with it, are all very important considerations.

**=**>slides: Pg. 15

This last optional property is actually pretty important. Let me define it first, and then I'll discuss why. The key is a value that, if present, will determine how and to which partition within a topic the Kafka producer will be sending the message to.

**=**>slides: Pg. 16

Do you remember this slide from the last module when I taught you about producers writing to multiple partitions within a topic? You had a question I said I would answer in this module, and now is the time. Well, almost. Let me finish out the discussion on producer record first and show how it works with the instance of KafkaProducer to make messaging magic.

**=**>slides: Pg. 17

Even though the key attribute is optional, I would urge you to avoid leaving it blank or null. The key serves to very useful purposes. It can be used as additional information in the message that can be used to make processing decisions later. And as we will soon see, it can strongly influence the manner in which messages are routed to the partitions. However, a possible downside to using a key is the payload overhead introduced when a key is added, which can depend on the type of serializer used. Once again, as with other design decisions, there are trade‑offs to be considered.

# 5. The Process of Sending Messages, Part One

**=**>slides: Pg. 18

So now we're back in our PowerPoint IDE, and in it we are adding an object of type, ProducerRecord, with the basic required attributes, but with a key, and next, we're calling the send method on the myProducer instance and passing the myRecord object we just instantiated as its required parameter. Since the send operation can be unsuccessful, it is always a good practice to wrap the call with a try..catch block and use structured exception handling. I didn't illustrate it here because of space limitations, but you'll see it in the demo.

**=**>slides: Pg. 19

Now that we have a producer record for the producer to send, let's see what actually happens internally. I like to look at the message sending process in two parts. For the next few slides, we'll discuss the first part. When calling the send method, the producer will reach out to the cluster using the bootstrap.servers list to discover the cluster membership. The response comes back as metadata, containing detailed information related to the topics, their partitions and their managing brokers on the cluster. This metadata is used to instantiate a metadata object in the producer and throughout the producer's lifecycle, it will keep this object fresh with the latest information about the cluster. Additionally, a pseudo processing pipeline within the Kafka producer is engaged. With the producer now having an actual producer record to work with, the first step in this pipeline will be to pass the message through the serializer using the configured serializer. Remember in our case, we're just using the string serializer. The next step in the pipeline is the partitioner, whose job it is to determine what partition to send the record to. Here the producer can employ different partitioning strategies, depending on the values being passed to it in the producer record, and the information it has regarding the cluster membership. This is where I finally get around to answering that all important question related to how the producer distributes messages to partitions.

**=**>slides: Pg. 20

Between the time the send operation is invoked to the time a message is received by a broker, quite a few things happen. We discussed the serialization step. Now is the all important partition routing step, which is determined by four possible strategies. First, the Kafka producer looks at the producer record contents, especially the partition field. It will look if there's a value provided for that partition field. If it has, the next question will be if the proposed partition is actually a valid partition, for example, for the topic being requested, is there a partition that matches the one proposed? For this answer, the producer refers to the metadata object that maintains the cluster metadata, including a list of topics, their partitions and the leaders for each. If the value proposed does not match a known partition for the topic, or if that partition is unavailable, which is unlikely if replication is enabled, then an exception will be thrown and the send operation will abort. If the proposed partition is valid, then the producer will add the producer record object to the specific partition buffer for the topic, where it will, on a separate thread, await the actual send to the broker leader of that specified partition. We'll get into this buffering step as part of the second part of the message sending process, but for now let's continue. If a partition was not specified in the producer record, the next question to determine the routing strategy is whether a key was provided in the producer record, because, as you will recall, it is an optional value. If the answer is no, as was the case in the last module when using the Kafka Producer Shell program, the message will be routed using a round robin strategy that attempts to evenly distribute the message across all the partitions in the topic. Now, technically speaking, this scheme is defined in the default partitioner class we'll talk about in a few more steps. If there is a key provided, the next qualifying question is whether a custom non‑default partitioner class was provided as part of the configuration properties provided to instantiate the Kafka producer. For this, the producer references the producer config object and looks for a specific value called PARTITIONER\_CLASS\_CONFIG, which represents the optional partitioner.class setting provided in the properties object. If there is nothing provided, which is the common default scenario, the routing will be done through a key‑based partitioning scheme, which Kafka provides as a default implementation of the partitioner interface. The default partitioner class takes a MurmurHash of the key and then applies a modulus function by the total number of partitions for the topic, and that's how it determines what partition to send it to. I suppose you could call that a fancy way of describing a murmur‑based mod hash. Some use cases may call for a custom key‑based partitioning scheme, and that's when you would need to develop your own partition or implementation, add that implementation class to the class path and specify the class type as the partitioner.class property setting. If that has been done, it is that custom scheme that will be used. I hope the way to answer that question was worth it and now you have a good idea as to how the producer determines what partition to direct messages to. This knowledge is very important for designing Kafka applications, which is why I was keen to spend some time on it. But this is an advanced topic, and like many advanced topics that is slightly beyond the scope of this course, I would encourage further study on it.

# 6. The Process of Sending Messages, Part Two

**=**>slides: Pg. 21

Now back to our map. We left off with the partitioner. In our walk‑through example, we didn't specify a partition, but we did provide a key, and therefore, according to the routing strategy flow we just covered, the key‑based partitioning scheme will be used, which again, is defined in the default partitioner class. This officially brings us to the second and final part of describing the message sending process inside of the Kafka producer.

**=**>slides: Pg. 22

With the partitioning scheme established, the producer can now dispatch the producer record onto an in‑memory queue‑like data structure called a RecordAccumulator. The RecordAccumulator is a fairly low level object that has a lot of complexity. We will not go into it in a tremendous amount of detail during this introductory course, but I will describe at a high level what it does and why.

**=**>slides: Pg. 23

But first, let's talk about efficiency. Each time you send, persist, or read a message, resource overhead is incurred. In high throughput systems, this overhead can dramatically impact the performance, reliability, and overall throughput of the system. And the more that overhead is incurred on handling fewer units of work, the less efficient that system is. Think of it this way, suppose you have a garage full of boxes and you need to move all of those boxes to a new destination. If the goal is efficiency, as far as how much you can get moved using the least amount of resources, like time and energy, what type of vehicle would you choose? Would it be a four passenger car, or would it be a moving truck? Overall, the answer would be a moving truck because you can transport more at once. Assuming you have an equal number of loaders and unloaders, you'll likely consume less time and energy with the truck because the smaller vehicle will need to make more trips. Thus likely incurring higher overall costs of time and energy. Of course, this metaphor can get out of hand, but I hope the point is illustrated nonetheless. This is Kafka's approach to addressing common inefficiencies in messaging systems, micro‑batching. Whether it be on the producing side, the broker side, or the consumer side. Apache Kafka was designed with the means of being able to rapidly queue, or batch up requests, to send, persist, or read in flexibly bound memory buffers that can take advantage of modern day operating system functions, such as Pagecache and the Linux sendfile() system call. By batching, the cost overhead of transmission, flushing to disk, or doing a network fetch is amortized over the entire batch.

**=**>slides: Pg. 24

The RecordAccumulator gives the producer its ability to micro‑batch records intended to be sent at high volumes and high frequencies. When a producer record has been assigned to a partition through the partitioner, it will get handed over to a RecordAccumulator, where it will be added to a collection of record batch objects for each topic partition combination needed by the producer instance. Each of these RecordBatch objects, as the name suggests, is a small batch of records that is going to be sent to the broker that owns the assigned partition. There are a lot of factors that determine how many producer records are to be accumulated and buffered into a RecordBatch before it is sent off to the brokers. Most of these factors are based on advanced configuration settings to find at the producer level that error set using a properties object, similar to the way the other properties were set. Let's take a look at a few of the important settings.

# 7. Message Buffering and Micro-batching

**=**>slides: Pg. 25

Each RecordBatch has a limit of how many ProducerRecords can be buffered. This limit is not based on the number of records, but rather by a configuration setting named batch.size, whose value represents the maximum number of bytes that can be buffered per each RecordBatch. Furthermore, across all buffers, there is a configuration setting that establishes a ceiling or threshold for how much memory can be used to buffer records waiting to be sent to the brokers. This setting is called buffer.memory, and like batch.size, its value represents the number of bytes. If the high volume of records being buffered reaches the threshold established by the buffer.memory setting, the max.block.ms setting comes into effect. This setting determines how many milliseconds the send method will be blocked for. This blocking contingency is intended to force back pressure on the thread the producer is using to send more ProducerRecords onto the buffer. The hope is that within the provided number of milliseconds, the buffered contents will be transmitted and free up more buffer memory to enable more records to be enqueued. When records get sent to a RecordBatch, they will wait around until one of two things happen. First, if record accumulation occurs and the total buffer size reaches the per buffer batch size limit, the records are sent immediately in a batch. This optimizes the overhead associated with transferring the page cache bytes to the network socket. This is the micro‑batching intention at its best. Simultaneous to this, new records are being dispatched to other accumulators and other record buffers. The second threshold that determines when buffered messages are sent is a configuration setting called linger.ms, which represents the number of milliseconds an unfull buffer should wait before transmitting whatever records are waiting. For example, if in one buffer, there is a single record waiting to be transmitted rather than to incur the overhead for a single message, the linger.ms setting will wait around for the specified number of milliseconds to pass before the actual transmission. For high‑frequency partitions whose buffers are being filled rapidly, the linger.ms setting generally does not come into play. We covered a lot of details and complexity with regard to the various configuration settings and how they can be set to affect the producer behavior, which will have a big influence on the overall performance of the system. Again, I don't expect you to come away from this an expert. In fact, after your head stops spinning, I would once again encourage further study and experimentation on this subject as it is an advanced topic.

**=**>slides: Pg. 26

Finally, the last part of the message‑sending flow is when the batched records finally get transmitted to the brokers, and the result of the transmission is sent back as a RecordMetadata object, which essentially contains information about the records that were successfully or unsuccessfully received.

# 8. Message Delivery and Ordering Guarantees

**=**>slides: Pg. 27

To ensure the best chance of delivery, there are some additional settings that should be considered, which are set at the producer level. We'll cover some of these here. First, when sending messages, the producer can specify what level of acknowledgement it expects from the receiving broker. This is a setting appropriately named acks and can be set using the property‑setting method discussed at the beginning. The first and most risky option is setting the acks value to an integer of 0. This essentially represents a fire‑and‑forget mode of sending messages because no acknowledgement whatsoever is sent by the broker. This approach is definitely the fastest in terms of request latency, but not very reliable, especially if there's an issue with a broker that prevents it from logging the message. The producing application has really no way of knowing if the message got there. Now this may be okay if the type of messages being sent with this mode can be lossy, such as possible clickstream data. The second middle‑of‑the‑road option is setting the property value to 1. With this, the producer is only asking for the leader broker to confirm receipt and persistence instead of waiting for all replica members in the quorum to confirm. This option offers a good balance of performance and reliability, providing the cluster settings employ appropriate replication. The third and final is when the property is set to 2 and thereby requesting from the cluster that all in‑sync replicas confirmed the receipt before counting the message as successfully sent. Obviously, this option offers the highest level of assurance that the message was successfully sent and received, but at the cost of performance, which can be unpredictable based on the possible changes in the cluster membership and thus replication topology. When any error is sent back, the producer needs to decide what to do with it. The first line of defense is to employ the retries configuration setting, which controls how many times a producer will, you guessed it, retry to send the message before aborting. Closely associated with the retry setting is the retry.backoff.ms setting, which allows you to specify the wait period in milliseconds between retries.

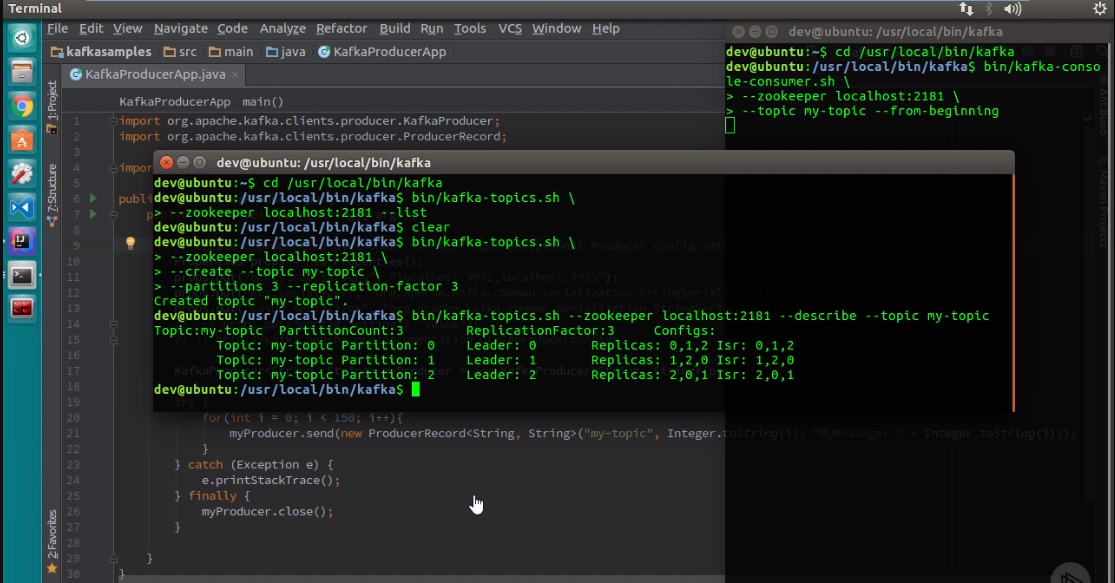
**=**>slides: Pg. 28

Depending on your application, message ordering can be important. If it is, these points are important to consider. I made a brief mention in the last module that message order is only preserved within a given partition. If the producer sends messages to a partition in a specific order, that order will be the order in which the broker appends them to the log and it will be the order that the consumers will read them from the log. Messages sent to multiple partitions, however, will not have a global order. Now this should be expected and understood at this point given our discussion about partitioning strategies. But to derive a global order across partitions, the order logic will have to be handled at the consumer level or even beyond. Regardless of the ordering assurances at the partition level, errors can complicate matters for expected reasons. If the configuration setting retries is enabled, and the retry.backoff.ms setting is set too low, you may have a situation where the first message is sent and a success acknowledgement is not received, causing a retry to happen. But before the retry can be sent, the second message is sent and successfully received while the retry first is sent and ultimately acknowledged. Now the result would be a reverse order within a single partition. The only way to avoid this, but at a high cost of throughput, would be to set the max.in.flight.request.per.connection setting to 1, which would effectively tell the producer that at any given time, only one request could be made. Ouch. But that may be what is needed. A combination of these settings will determine the message delivery semantics required by your system. It is possible to achieve either an at‑least‑once, an at‑most‑once, and an only‑once message delivery assurance, but only with the design that carefully considers the settings available at all three component members of the system, the producer, the broker, and the consumer.

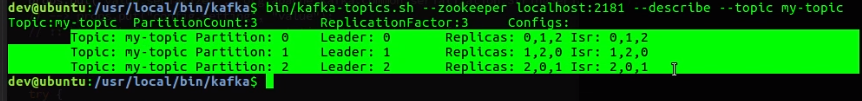
# 9. Demo: Creating and Running an Apache Kafka Producer Application in Java

**=**>slides: Pg. 29

It's the long‑awaited demo time where I will endeavor to show you how you can start to build a custom Kafka producer application in Java. In this, I will cover some of the highlights of what we've been discussing in this module. But overall, the scenario will closely resemble that of using the Kafka producer shell program. We will use a basic producer configuration against a cluster setup, consisting of a topic with three partitions, three member nodes, and ensured with a replication factor of 3. In this, look for evidence of the default partitioner being used. This will be seen when the single consumer that we use reads from the topic. You'll notice that there will not be a global order.



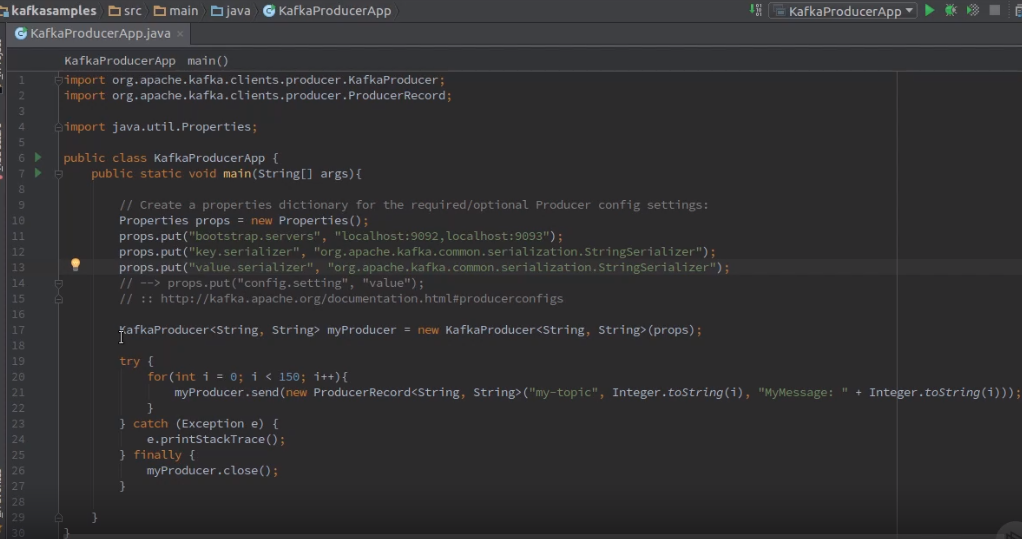
Okay, so in this demo, I've got the details of our topics up here.



As you can see, we have our topic, my‑topic, with a partition count of 3, replication factor of 3, as we said. And you'll notice that each partition has its own line. And because there's three nodes, each one is a leader for a partition. And over here, we've got a terminal window listening for messages.



Additionally, within the IDE, we have the terminal, which is also running a consumer waiting for messages to arrive. Okay, so let's minimize that and look at the code.



Alright, so in the code window, you'll see that we have Kafka producer. But before that, we have our properties, and we're instantiating some of those important required properties, just like we saw in the slides.



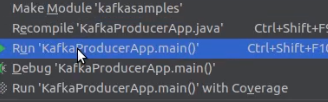
But here, we have a Kafka producer instantiation, and you're going to notice that the signature is a little bit different. The signature, basically, is indicating that it wants strings for keys and values in the signature itself, and that's one of those optional instantiations that I suggested you could do earlier in the slides moving beyond the Kafka producer instantiation.



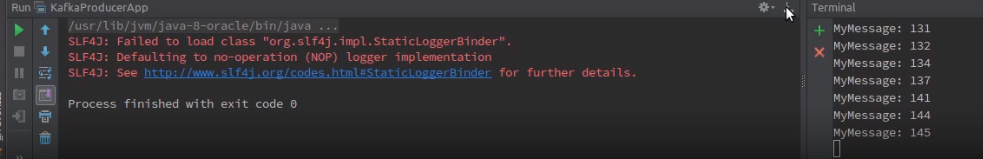
Here within a try, catch, finally block, we have basically a bounded loop where we take my producer instance, and we actually call the send method, passing in a new producer record. You'll also notice that again its signature is specifying the key and the value serializer upfront. And in this case, it's passing the my‑topic and passing in a key and value. Both of them are strings as the signature suggests. The key is being derived from within the loop as just an integer that we're casting to a string. And then my message with the integer represents the message and the number of that message. All right, so let's look at the catch. We're just doing a standard catch, catching for an exception in case of an error occurs in the send. And then we have our finally block here.



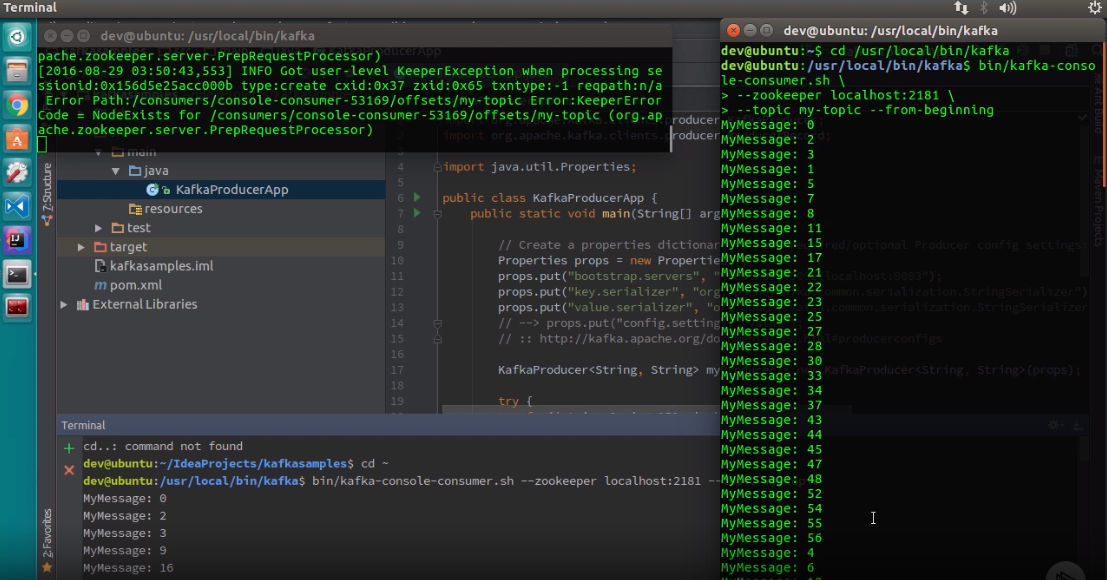
And the finally block we didn't talk much about in the slides, but we should have. Really, it's the opportunity to gracefully close down the producer. If you do not do that, like other types of resource‑intensive, network‑aware code, you can cause memory leaks, and it can cause all sorts of problems. So it's just a general good idea to make sure you're closing the producer gracefully as to avoid any sort of leaks whatsoever. All right, now let's actually run our application.



Let's run the producer itself and see what happens. Okay, so it's compiling, it's running,



and as you can see in this terminal window over here, and let me make that a little bit bigger, you can see all of the messages. Now they're not in order, and the reason why is because the partitioner is taking the rights from the producer and spreading them across three different partitions, and it's doing it in a different order. It's not exactly even.



And you can verify that in this terminal window as well, which was also a consumer, and you can see that it is definitely not in order. But I hope that this illustrated what we intended to do, and that is to show a very simple Java‑based producer that is producing messages and using the default partitioner, which is that key‑based partition because we did provide a key, and that it's doing so across multiple partitions.

# 10. Advanced Topics and Module 4 Summary

**=**>slides: Pg. 30

We covered a lot, but there are some things we just weren't able to cover in this introductory course. But I wanted to give you a highlight of what they are so you can use them as topics to explore more. Custom serializers, why and how to create customer serializers. Customs partitioners, why and how to create custom partitioning schemes. There's options to send messages using an asynchronous callback and a future. It would have been nice to show you these, but I think it's something that you can explore on your own. Applying compression options, which also falls into the category of advanced settings and combinations for optimal throughput and performance.

**=**>slides: Pg. 31

Throughout this module, we focused on the internals of a Kafka producer. We started with the high‑level map and started to drill down component by component and, in the process, covered properties and how they are represented as ProducerConfig objects. How we think of a message is really an instance of a producer record class. We discussed the processing pipeline when sending a message using the KafkaProducer class and how the producer record goes through a serialization step and a partition assignment process. We spent some time talking about how Kafka optimizes message throughput through microbatch ing, and we walked through the internals of the Kafka producer with the record accumulator and record buffer as the means it can accomplish micro batching with related configuration settings. We touched on message delivery and ordering guarantees offered by Kafka and some relevant configuration properties to consider when designing your applications. Finally, we ended with a brief demo on a basic Java‑based producer. Next, we'll cover the other type of client application, the Kafka consumer. I'll most likely take the same approach as I did with the Kafka producer in this module. So I hope you enjoyed it and learned enough to be anxious to start exploring on your own and continuing in this course to learn more.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*