# **Apache Kafka Essential Training: Getting Started**

### **Getting started with Apache Kafka**

Selecting transcript lines in this section will navigate to timestamp in the video

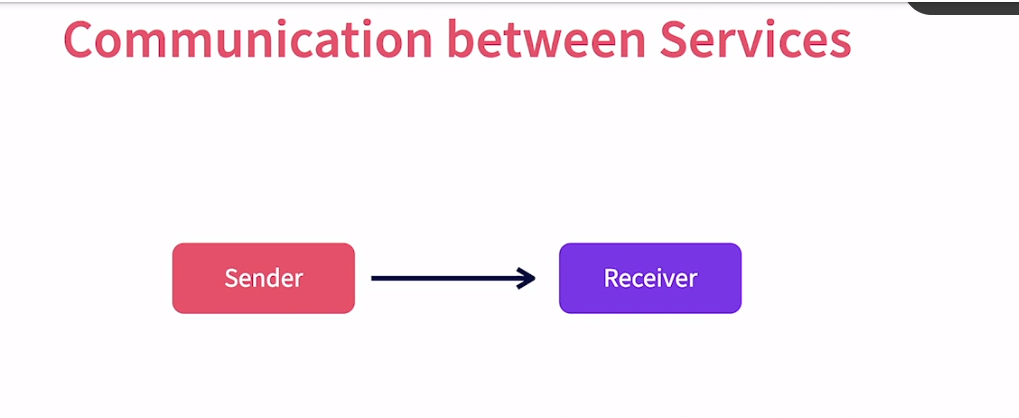
- [Instructor] Smartphones, social media, email, and yes, even memes, there is so much data in the world today. And so much of that data is valuable to big companies as they try to predict and determine customer behavior. Apache Kafka is the leading open source technology for message queuing and distribution with widespread adoption. My name is Kumaran Ponnambalam. In this course, I will introduce Apache Kafka and show you how to build data pipelines with it. Then I will use command line tools to publish and consume data. I will then discuss partitioning with Kafka. Finally, I will show you how to build Apache Kafka clients using Java. Let's now start learning about Apache Kafka.

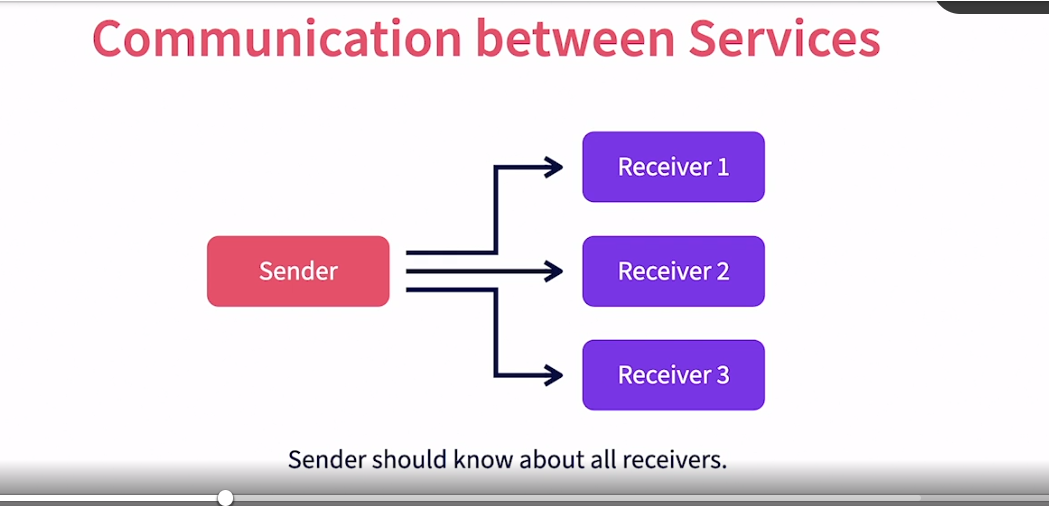
### **Message queues**

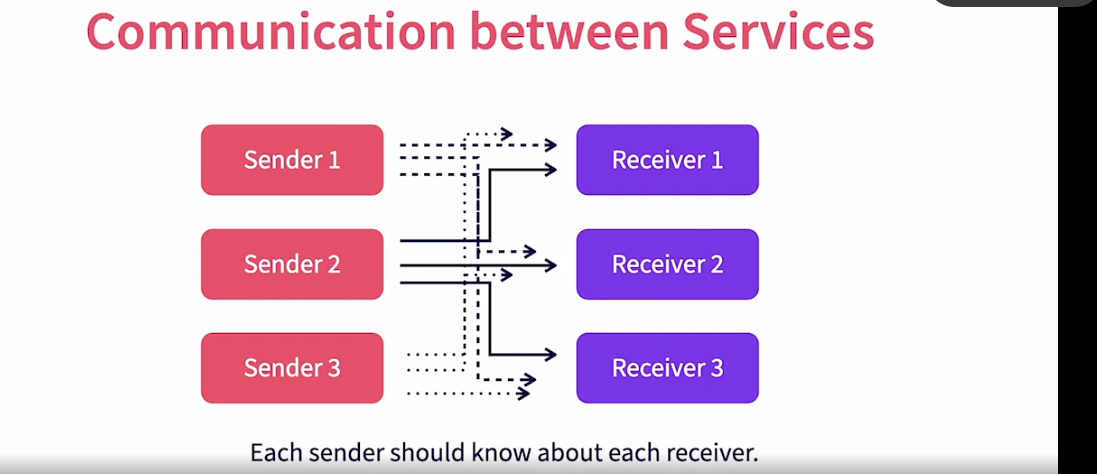
Selecting transcript lines in this section will navigate to timestamp in the video

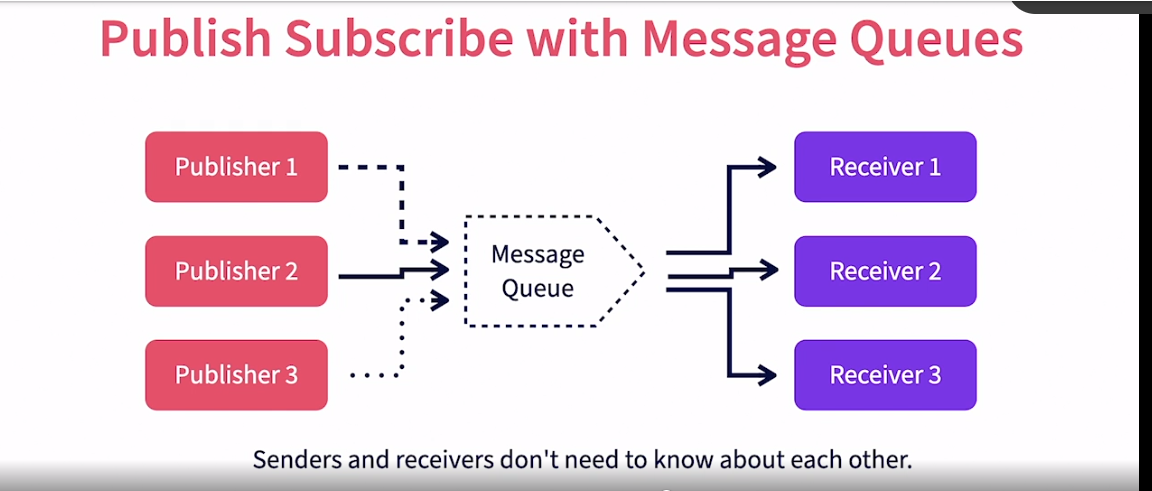
[Instructor]- Before we dive into Apache Kafka it's important to understand the publish subscribe pattern and the benefits it provides. Let's review publish subscribe with message queues in this video. When we build distributed services, we have multiple running process instances that communicate with each other over the network to exchange messages. Let's start with the simple two process network where a sender process communicates but the receiver process to exchange data. In this case the sender needs to know the address of the receiver. In addition, both the sender and the receiver needs to agree upon a common data protocol and farmer. This creates a strong static binding between the two processes. Now suppose we have three receivers each responsible for a different type of processing based on the data received from the sender. In this case, the sender needs to know about all the three receivers. Addition of new receivers require changes on the sender side to configure the new receiver. Again, this creates a strong coupling between the senders and receivers. What happens when there are multiple senders, and multiple receivers exchanging the same type of data between them? The management becomes a lot more cumbersome here but each sender should know about each receiver. Add fault tolerance into the picture and it will get more complex. **A clean and scalable solution here is to use a message queue. Each sender needs to only know about the message queue and simply publish the messages to the queue. A receiver becomes a subscriber and subscribes to the message queue. When a new message appears in the queue the subscriber is notified, who then proceeds to pull the message and use it. In this case each of the publishers and subscribers only need to know about the message queue. They are unaware of other publishers and subscribers using the same queue. This is the publish subscribe pattern also called pubsub.** This pattern has many advantages. To begin with, the publishers and subscribers are decoupled from each other. This results in easy management of the setup where publishers and subscribers can be added and removed without any changes on others in the network. It then allows to scale the publishers and subscribers easily provided the message queue can handle the load. Message queues also provide Back-pressure handling. If the publishers generate data in spikes the message queues can act as a buffer zone to cache data until the subscribers catch up and process them. Message queues can also provide reliability through persistent queue data and tracking consumption of data. Apache Kafka is such a technology that provides these advantages. Let's learn more about it in the next video.

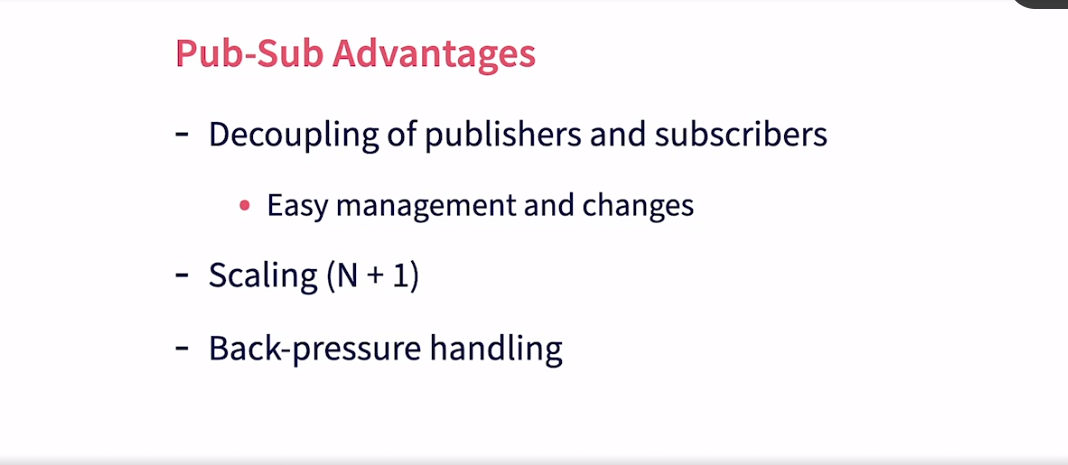












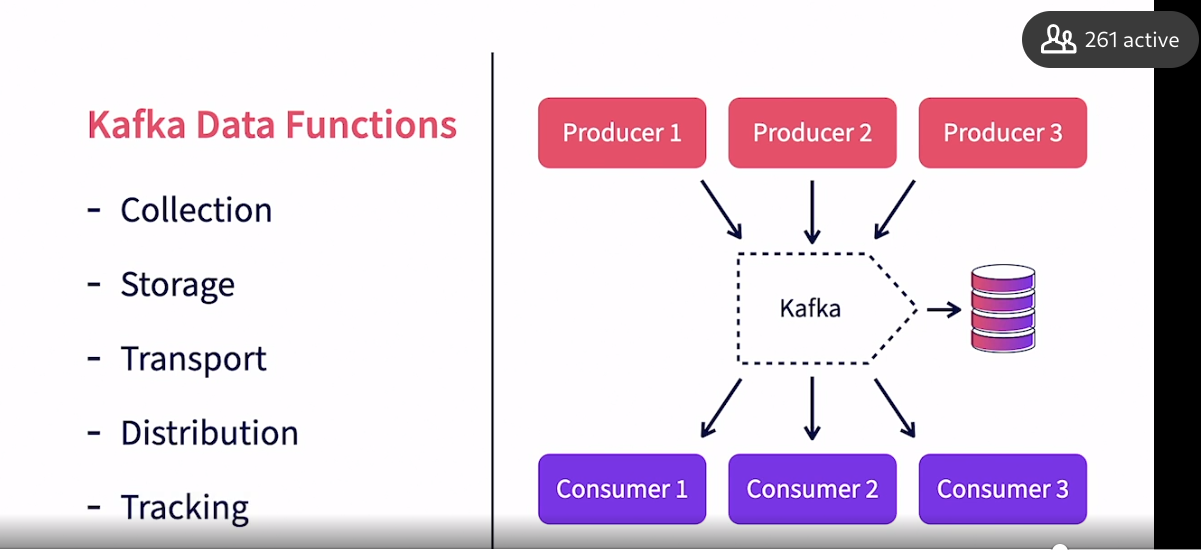
REliability

### **What is Kafka?**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] Having reviewed the benefits of the Publish-subscribe pattern in message queues, let's dive into the most popular technology in that domain. Apache Kafka. What is Apache Kafka? Kafka is an event streaming platform, events or messages represent the actual data that is exchanged through Kafka. The terms events and messages are used interchangeably in Kafka's context. It is a critical piece of the Big Data puzzle, and plays an integral part in many big data pipelines. Kafka is open source, and can be downloaded and deployed free of cost. There are also commercial options that provide support and serverless capabilities. It's arguably the most popular messaging platform in the world. In Kafka's world, there are data publishers called Producers, which push messages into Kafka. And there are subscribers called Consumers, which listen to and receive messages. Producers and Consumers are the standard terms in the Kafka world to represent publishers and subscribers. What capabilities does Kafka provide for data exchange? It collects messages from multiple producers concurrently. It provides persistent storage of the messages received. This provides fault tolerance capabilities. It transports data across from producers to consumers. With mirroring capabilities, it can also transport across networks. It distributes data to multiple concurrent consumers for a downstream processing. Finally, it provides tracking of message consumption by each consumer. This ensues at least once delivery of messages, even if the consumers go down and come back again. We will discuss more details about these features and see them in action, throughout the course.

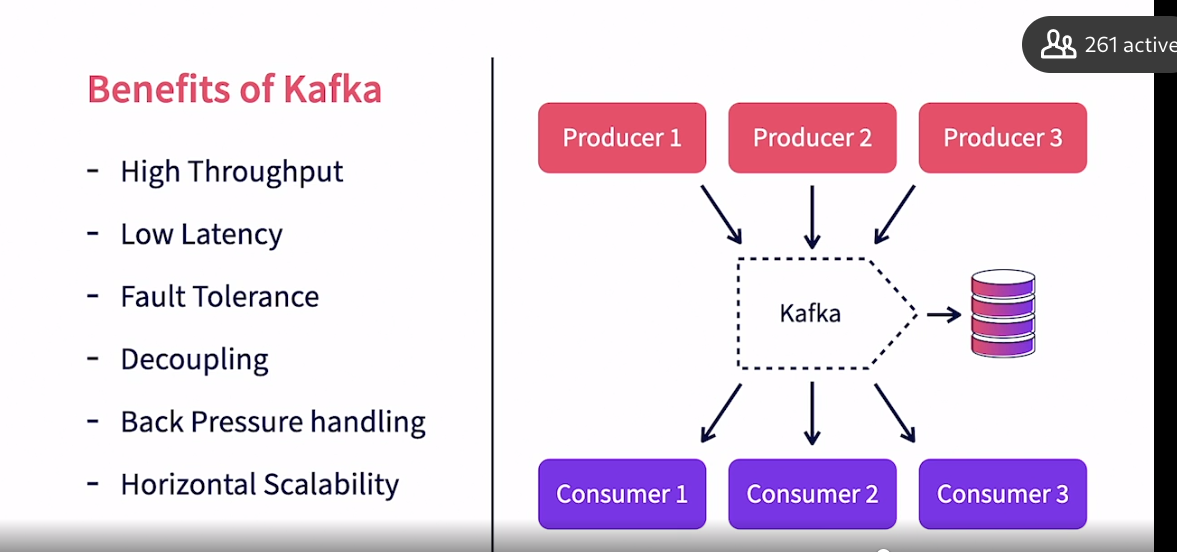


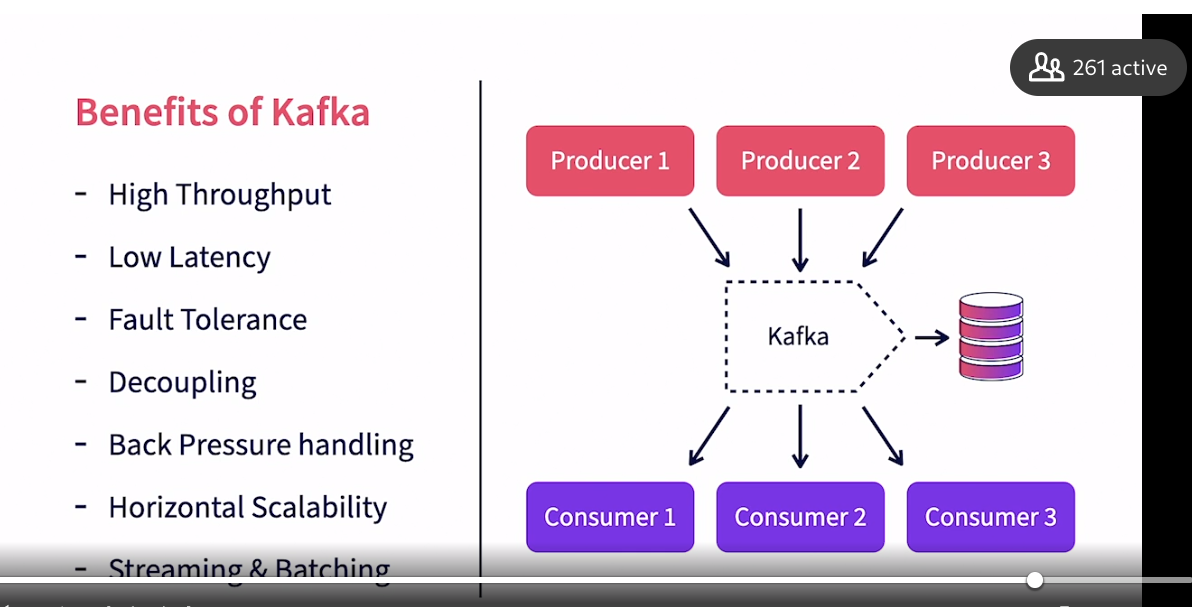


### **Benefits of Kafka**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] What are some of the key benefits of Apache Kafka? Kafka provides high throughput and can handle large quantities of data. It supports multiple concurrent producers that can push data and multiple subscribers for the same data. It can provide such high throughput with low latency of a few milliseconds. This enables a number of realtime use cases where the latency is not noticeable end to end. Kafka provides excellent fault tolerance against failures, either within Kafka or with consumers. This ensures that messages are reliably stowed and distributed, in spite of issues with individual processes. Kafka decouples producers and consumers. They do not need to know about each other. This enables ease of configuration and management. It also makes software development a lot simpler, as the developer of the producer or consumer program only has to deal with Kafka and doesn't have to worry about other producers or consumers of the same data. One benefit of decoupling and storage capabilities is back pressure handling. Even if the producers produce data in spikes, the consumers can catch up at their own pace, as Kafka provides the buffer in the middle to hold data until it gets consumed. Kafka provides horizontal scalability within its architecture, as to allow for producers and consumers. The system can scale with addition of notes without significant choke points. This is essential for big data processing. Kafka's low latency enables string processing while it's store and forward capability enables badge applications. The consumers need not have to consume the message immediately, as Kafka acts as a buffer. Now that we have looked into Kafka's benefits, let's review the most popular use cases for Kafka in the next video.

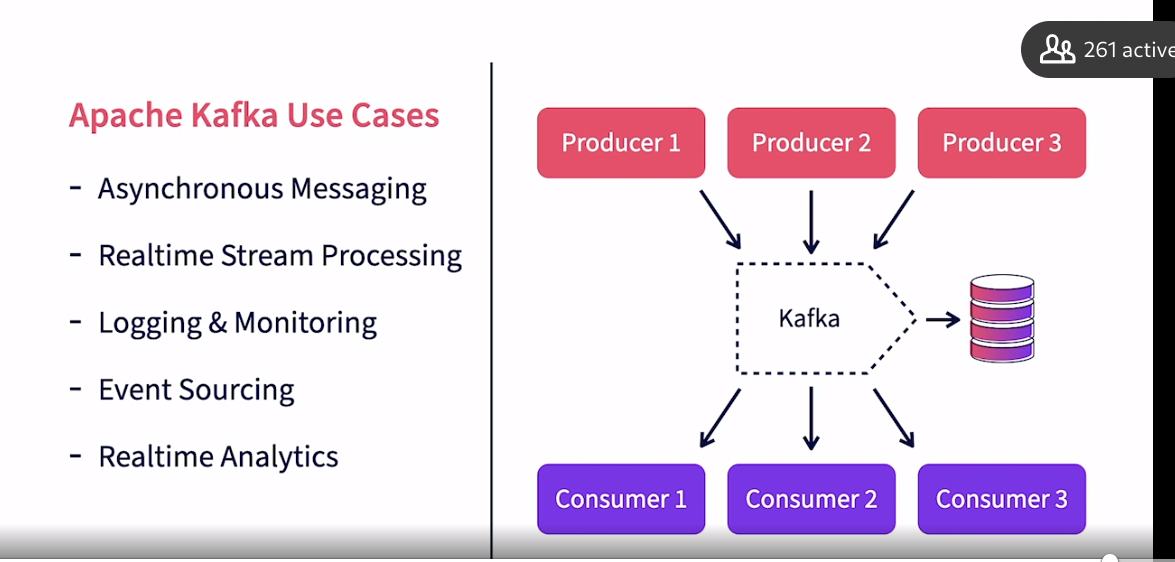




### **Kafka use cases**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] What are some of the most popular use cases for Apache Kafka? Kafka enables a variety of use cases both in batch processing and real-time streaming. To begin with, it can provide asynchronous messages between services. Rather than use dressed or GRPC, a client service can send requests through Kafka and receive responses through Kafka. It provides better reliability and manages back pressure on the server side. It has low-latency, enough for even browser-based realtime applications. It enables realtime stream processing. As messages are received in Kafka, consumers can consume them in real time, process them and trigger real time actions. Kafka queues can be used to log messages and alerts. One consumer can archive these messages in a persistent store. Another can look for key exceptions in realtime and trigger alerts. Kafka enables the event sourcing pattern where the state of an entity can be determined using the evens generator about the states for that entity. This is a popular pattern for big data. Finally, Kafka enables realtime analytics. An Apache Spark or Apache Flink consumer can listen to the messages in real time and generate windows aggregations, analyze trends and generate triggers, metrics, and actions. This can be used to update realtime dashboards. Having discussed the benefits and use cases of Kafka, let's now install the required software and get set up for the rest of the course.



### **Setting up the exercise files**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, I will help you set up the exercise files for this course. In my case, I have downloaded the files into my documents slash exercise files directory. Please don't load in a similar location in your setup. There are two exercise files here. The Java project that is bundled into the kafka-getting-started.zip file and the Docker compose YAML file called kafka-single-node.yml. Let's first start to set up the Java project under kafka-getting-started.zip. Unzip the file kafka-getting-started.zip. This will create the folder Kafka Getting Started with the project contents under it. Open IntelliJ IDEA. Select Open or Import. Choose the route folder for the exercise files. Now IntelliJ IDEA would proceed the combined the project. It will download all the dependencies if it is not already available in your local Maven repository. Please make sure to select the right SDK in the Project Structure option. Please make sure that the Project language level is eight. Make sure that the project compiles without any errors. The project contains both the Java classes and also instructions to execute various command lane steps. The command line rater steps are available under the resources directory. Please explore the same. We will use this as we progress through the course. Now let's proceed to set up Kafka for our exercises in the next video.

### **Setting up Kafka**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] We will now be setting up Kafka for our exercises using Docker. The instructions for setting up Kafka are available in the file resources/chapteronesettingupKafka.txt. Please take a moment to review these steps. We are going to use Kafka in Docker. If you are using macOS or Windows OS, please install Docker desktop from the Docker website, docker.com/products/dockerDesktop. Kafka requires two containers, one for Zookeeper, and another for Kafka. The Docker compose configuration for the same is available in the Kafka single-node.yml file. Let's explore this file now. There are two services needed for the setup. The first one is Zookeeper. Zookeeper is a prerequisite for Kafka, and we will discuss more on this later in the course. We will be using the Docker image from Bitnami for Zookeeper. This exposes the port 2181. The container name is Zookeeper. Next, we have the Kafka service, which is also from Bitnami. We are using the latest container, so it is possible that the version you will be setting up could be different than the version I will be using, which is 2.7 While we expect most of the semantics to be the same, do watch out for any mismatches. We will be exposing two ports in Kafka. Please pay attention to the difference, since this is important and can cause confusion. One port 29092, will be used for other Docker containers, to communicate with this container. This port should only be used when the client is another Docker container. A second port 9092, will be used when accessing Kafka from the host system. Inside the environment construct, we will actually be configuring two separate environments, internal and external. We start with setting up the listener security protocol, which is plain text. Then we set up two listeners internal and external, pointing to the corresponding ports. These are the ports that Kafka will listen for connections. The advertise listeners specify the client FQDN. For internal, it uses the name Kafka. This means if you're accessing it from another container, you will use Kafka:29092. The external is configured as localhost:9092. This means if you are accessing from your Docker host, which is any program running directly on your desktop, like IntelliJ IDEA, you will use this URL. Kafka needs Zookeeper, and we will specify the Zookeeper FQDN in the Zookeeper connect parameter. Finally, we set up a dependency on the Zookeeper container for Kafka. Let's start the containers now. Open the terminal window. If you're using Windows, please use the PowerShell Windows. First, navigate to the directory where the Kafka single-node.yml file exists. Now, Randy Docker Compose command. The command is Docker-compose minus f if it indicates take the configuration from the file, Kafka single-node.yml. App says that bring up all the containers and -D means run as a demon. Let's run the command now. This will download the images, if they are not already on your local Docker, and then proceed to start the containers. Depending on the download process, it may take some time to start up. Now, check if both the containers are running using the Docker PS command. You should see both the Zookeeper and the Kafka containers App. If you need to shut down the containers, you can bring the containers down with the command Dockercompose-Kafkasinglenode.yml down command. Please note that when the containers are removed, data in the Kafka logs are lost. For now, I am leaving them running. Let's now deep dive into Kafka concepts in the next chapter.

## **Question 1 of 4**

One of the pain points of having direct communication between the senders and receivers is that \_\_\_\_\_.

* Senders required more memory
* binary data exchange is not possible
* Receivers require more CPU
* each Sender should know about each receiver  
  Correct

## **Question 2 of 4**

Which of the following is NOT a Kafka data function?

* processing  
  Correct
* storage
* tracking data consumption
* collection

## **Question 3 of 4**

What does back pressure handling in Kafka mean?

* Kafka can survive with minimal CPU resources.
* Kafka can cache data until the consumer is ready to receive & process it.  
  Correct
* Kafka drops messages when the number of pending messages increase beyond a threshold.
* Messages can get from producers to consumers within milliseconds.

## **Question 4 of 4**

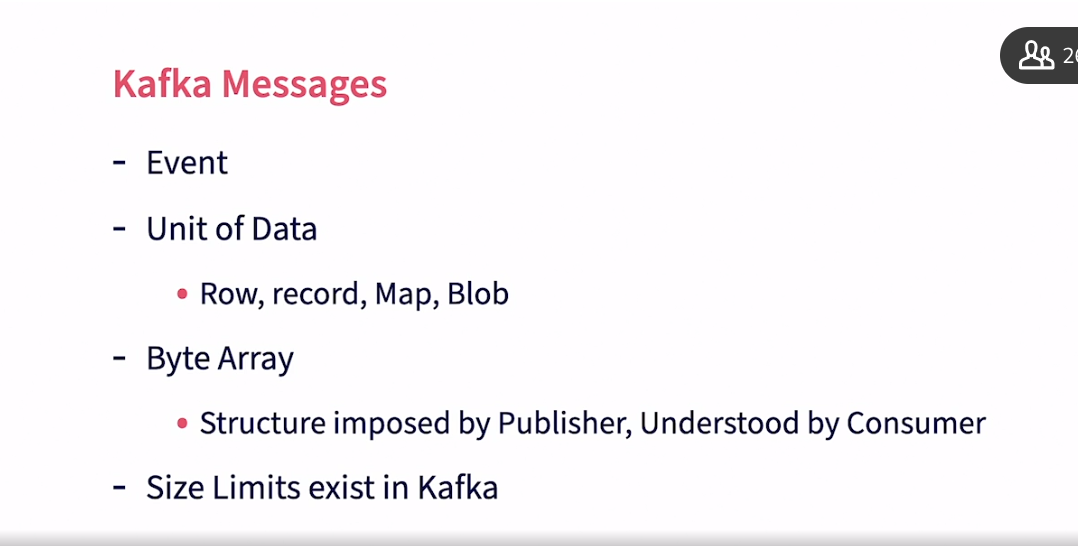
Which of the following is an important use case for Kafka

* state management  
  Incorrect
* graph database  
  Incorrect
* event sourcing  
  Correct
* video player

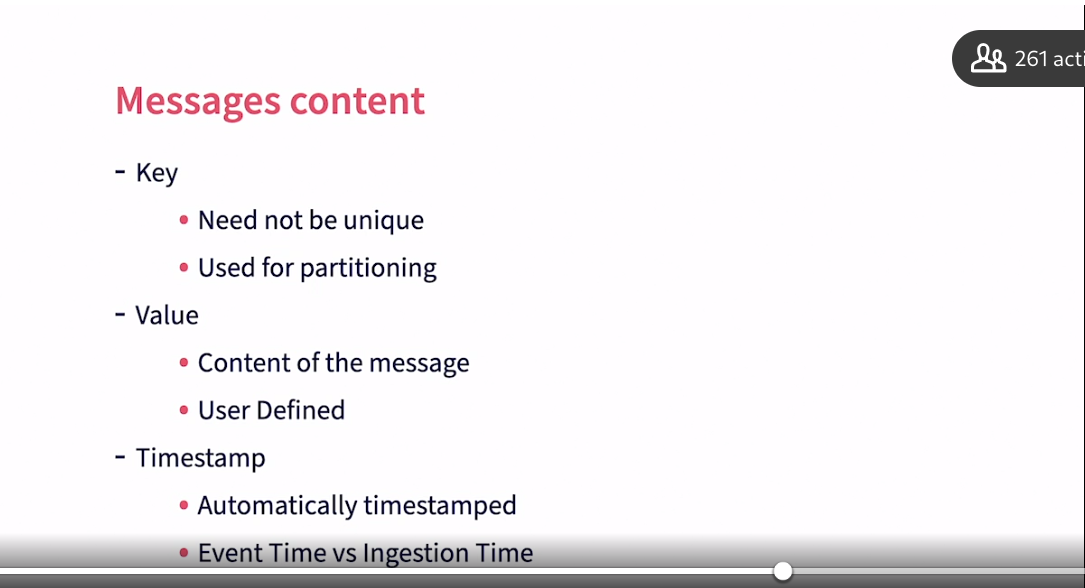
### **Messages**

Selecting transcript lines in this section will navigate to timestamp in the video

[Tutor] - In this chapter, we will explore a number of basic concepts and entities that make up Kafka architecture. We start off with a Kafka message. **A Kafka message is the unit of data that is collected, stored and distributed by Kafka. Let's explore more about messages in this video. A Kafka message is also called an event. A message is record of a real world event at a point in time. But that definition does not constrain what a message is.** It can be any piece of data. A message is equivalent to a row or record in a database. It can have attributes and values like a map. It can also be a blob that contains an image or an audio snippet. **Kafka treats all messages as a byte array. It does not try to associate any semantics on the content of the message. That is the job of the producer and the consumer. Producers and consumers need to agree upon the content and format of the message. And be able to serialize and deserialize them. Kafka merely takes in binaries and distributes them. Size limits exist in Kafka and the maximum size of the message. It is configurable and the default size is One MB.** **While producing and consuming messages, the producers and consumers can do batch processing for efficiency.** What are some of the key contents of a message? Kafka does have some predefined attributes. Messages in Kafka have a key. The key is defined by the producer of the message. Key are not mandatory and they also need not be unique. Keys are used for partitioning data. We will discuss partitioning further in the course. The value attribute of the message contains the actual message. It is a binary and the semantics of the value is user defined. Kafka does not infer anything from the message contents. Another key attribute to be noted is the timestamp. Every message is automatically timestamped by Kafka. Kafka supports two type of automatic timestamping. Event time is when the message producer creates a timestamp. Ingestion time is where the Kafka broker timestamps it, when it stores the record. This option is configurable. Now let's look at some examples for messages. The first message is a map with attribute names and values. In this case, it's an employee record in Jason. The message key is set to the employee ID. The second message is a web server log stored in CSV format. It has no explicit key. Kafka assigns a random key when a key is not provided by the producer. The third message is an image. It has the customer ID as the key. The content is raw bytes. Note that all these messages are internally stored by Kafka as by binaries. Hence the content can take any form. **As long as the producers and consumers agree on the format. Messages are stored in topics**. Let's explore topics in the next video.





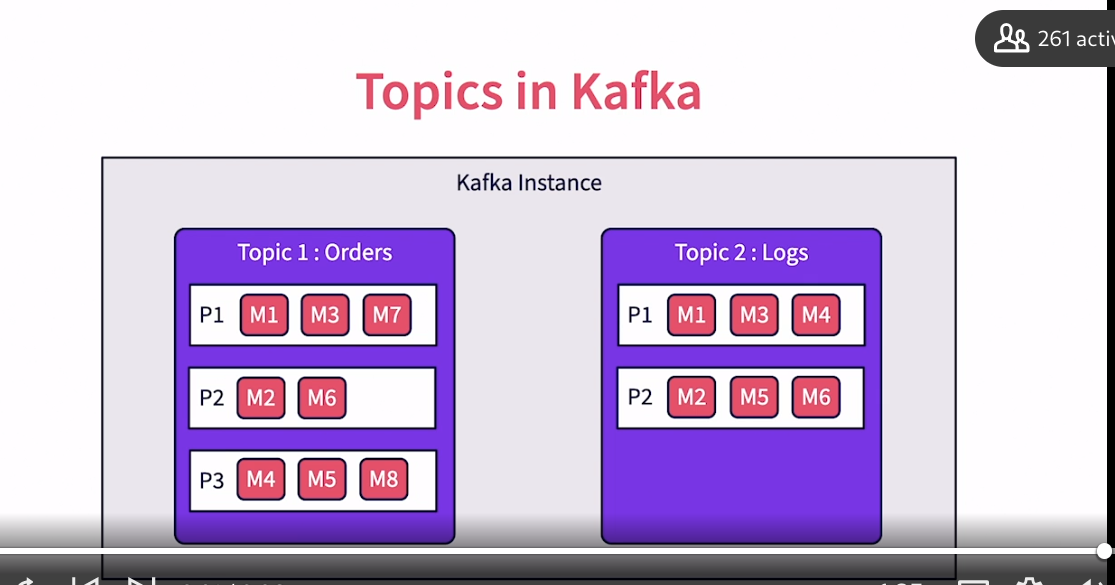


### **Topics**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] Topics in Kafka hold and manage messages. Let's explore more on topics in this video. A topic in Kafka is an entity that holds messages. It's similar to a file that contains papers. When messages represent the papers, it's similar to a database table that contains records, where messages represent the records. Topics can be considered as a queue, for similar messages. A topic itself does not pose any limitation, on what the content of its message should be. So technically, a topic can contain all kinds of messages with different formats, but in practice, they're used to hold similar messages. What does similar mean in this case? Usually, the content of the messages are similar and have the same set of producers and consumers. Examples of topics include; sales transactions, audit logs, video files, etcetera. Kafka support multiple topics, per Kafka instance. So topics can be created based on specific use cases. And multiple application domains can create topics, in the same Kafka instance. Each topic support multiple producers to publish data to the topic concurrently. Similarly, multiple consumers can consume data from this topic. Each topic has multiple partitions that physically split data across multiple files. Here is an example of how topics exist in Kafka. In this case, we have one Kafka instance with two topics, orders and logs. The orders topic has three partitions, P1, P2 and P3. There are eight messages in the topic named M1 to M8. The messages are distributed across three partitions. Each message, will only be stored in one partition. Similarly, the logs topic has two partitions and six messages. We will discuss partitions in detail, later in the course. In the next video, let's discuss Kafka brokers.





### **Kafka brokers**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] **A kafka broker is the center brain behind everything that kafka does. Let's explore more about brokers in this video. A broker is nothing but a running Kafka instance. It is a physical process that runs on the base operating system and executes all Kafka functions. A Kafka process listens on a specific port. Usually this is 9092 but it's configurable.** **The Kafka broker receives messages from producers and stores them locally in logs. Consumers subscribe to specific topics within the Kafka broker. The broker keeps track of all active consumers. It knows about the last message that was sent to a consumer**. So only sends any new messages in the subscribe topic to that consumer. It also keeps a heartbeat with every consumer. So when a consumer dies, it can track and reset. Kafka brokers manage the life cycle of topics. They track and manage topic partitions. They also manage the corresponding partition logs. Multiple Kafka brokers can be clustered together to form a single Kafka cluster. Within a Kafka cluster, there is one Kafka broker instance that will act as the active controller for that cluster. In addition, each partition will have a corresponding Kafka broker as its leader. The leader then manages that specific partition. A kafka broker also takes care of replicating topic partitions across multiple brokers. So even if one broker goes down, the other brokers can take over the corresponding topic partitions. This provides fall tolerance for Kafka. In the next video, let's explore Kafka logs.

### **Logs in Kafka**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Where does the data that is sent by producers stored physically? It is in the Kafka logs. Kafka logs are the physical files in which data is stored before they are consumed by the consumers. Logs are managed by Kafka brokers. Each broker has an assigned log directory where it stores the log files. There are multiple log files created in Kafka. Each broker will have its own log territory. And in these directory, there are separate files for each topic and partition. These are rolling files so when a file gets filled up it's rolled over and a new file is created to continue with the logging process. S**o each partition will have multiple log files in the log directory. Data in Kafka is only kept for a configured interval of time. The default is seven days. A separate thread in Kafka keeps pruning files that are over this period. Log files are an important consideration for managing a Kafka instance since they influence the amount of physical space that needs to be provisioned.** Lack of space would lead to the broker rejecting data from producers and a breakdown of data processing pipelines. All configuration for Kafka is in the server.properties file under the conflict folder of the Kafka installation root folder. The log.dirs parameter is used to set the path of the log directory. Now, let's explore more around the producers and consumers in the next video.

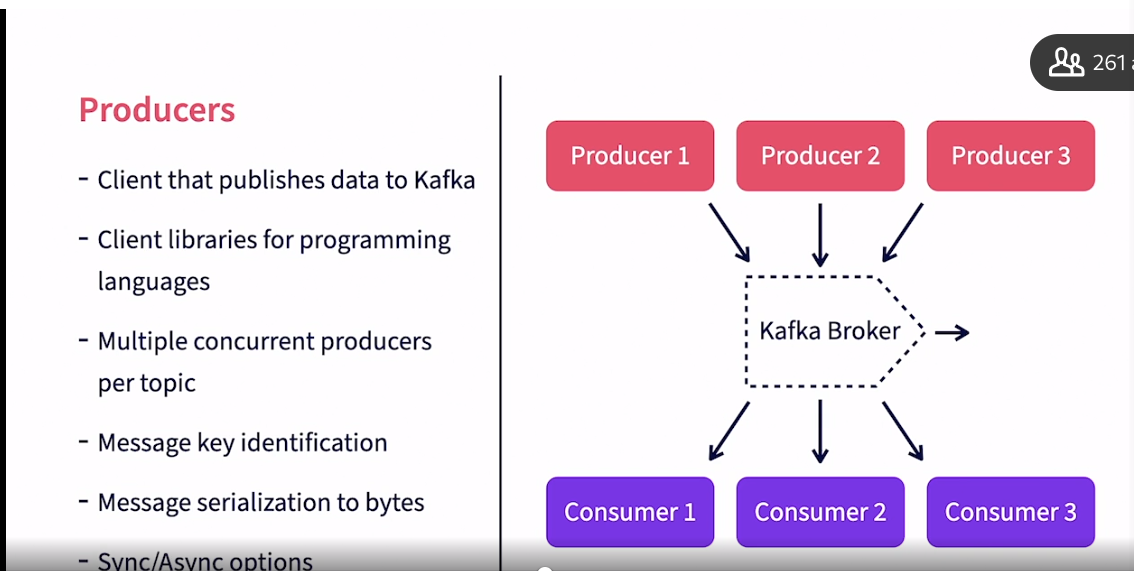


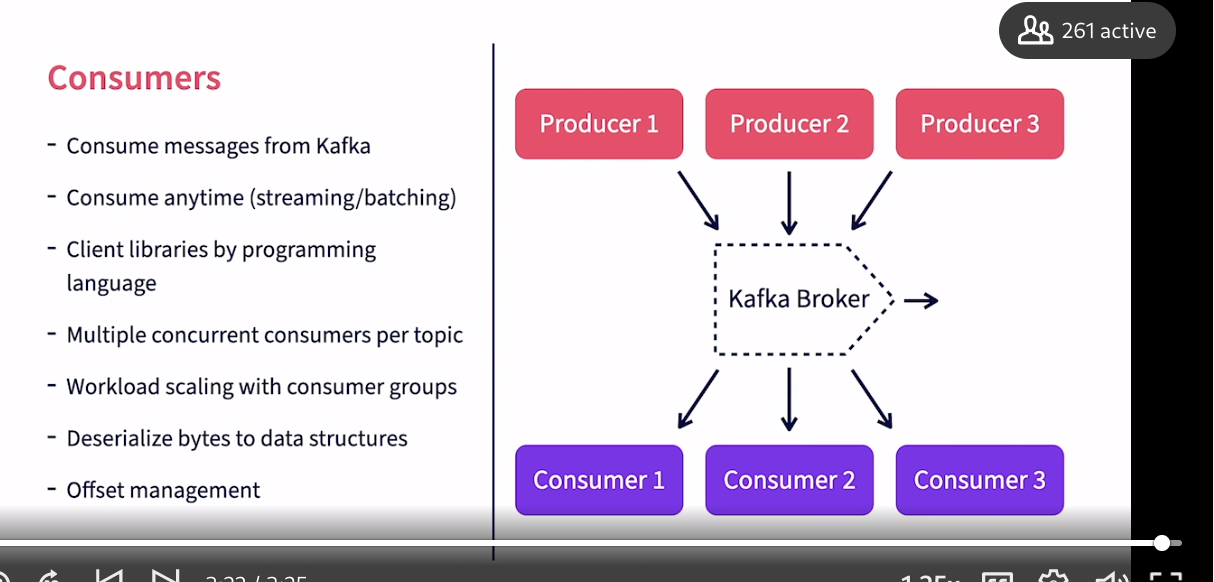
### 

### **Producers and consumers**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Producers and consumers from the ecosystem for Kafka, from feeding in data to consuming them. Let's discuss more about them in this video. A producer is any client that publishes data to Kafka. Some examples include web servers that publish web click events, a log scrubber pushing in log messages, and a sensor pushing in telemetric data. To build a producer, a developer needs to use a Kafka client library within their code and publish data. Such client libraries are available for multiple programming languages. Please check the Kafka website for more details. We will discuss Java client libraries in this course. There are multiple concurrent producers for each topic. These producers can be different physical processes or threats in the same process. It is the job of the producer to identify the key of the message. Typical keys would include customer IDs, product IDs or transaction IDs. Please note that the keys influence partition selection. Ideally, the string values for these keys should have equal distribution to ensure equal load across all partitions. The producers also need to serialize the message data to bytes. Anything can be serialized including strings, JSON, objects, or blobs. The only requirement is to make sure that the consumer can deserialize it to recreate the same data. There are synchronous and asynchronous options available for publishing to Kafka. Asynchronous options don't wait for an acknowledgement from the broker before proceeding with the next message and hence it's faster, but it also results in complex tracking of acknowledgements and republishing. A consumer on the other hand, is used to consume and use the messages from Kafka. Typical examples of consumers include a log filter, a real-time data, or a data archiver. Consumers can consume a message anytime as long as the message is stored in the log files and not pruned over. So in real time, the producer and the consumer are running at the same time and as each message is published it is immediately consumed. In batch processing, the producer can publish anytime, even in batches. The consumer process can start up at a much later time and consume all the pending records. The same client labored is used for producers will be used to consume the data by consumers. It will be a different set of function calls. There can be multiple concurrent consumers per topic and each consumer will get a complete set of messages from the topic. If scaling is needed beyond one consumer for a specific job, then consumers can be grouped into consumer groups and share the load. We will discuss consumer groups in detail later in the course. Consumers are responsible for deserializing the messages in byte format and recreating the original objects sent by producers. Consumers also can manage the offset for data that they would consume. They can consume from the start of the topic or from a specific offset. They also provide acknowledgement to the brokers once they have successfully consumed a message. Finally, let's explore the role of zookeepers in Kafka in the next video.





### **Role of ZooKeeper**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Every Kafka instance or cluster needs an instance of ZooKeeper deployed along with it. **Let's discuss the purpose of ZooKeeper for Kafka in this video. When we deploy Kafka, it needs an instance of ZooKeeper provision with it. ZooKeeper is a centralized service that is used for storing configuration information as well as helping with distributed synchronization. ZooKeeper serves as the central information store for Kafka. There are ongoing efforts to make Kafka independent of ZooKeeper in the future versions. ZooKeeper helps Kafka in broker management. When each Kafka broker starts up, they register themselves with ZooKeeper. They also discover about other Kafka brokers from the same ZooKeeper. One of the Kafka brokers, typically the first broker that starts up, registers itself as the active controller. It then controls and manages other brokers in the cluster.** **If the active controller fails, one of the other brokers will take up that role. This synchronization is handled through ZooKeeper. ZooKeeper is also used to manage topics. All topics and their partitions are registered with ZooKeeper. It is used to track and manage broker leaders for each of the partitions. Again, failure recovery is synchronized through ZooKeeper. It is recommended to install ZooKeeper as a cluster in production environments to provide fault tolerance.** Now let's start to use Kafka for creating topics and publishing data in the next chapter.

## **Question 1 of 6**

Which of the following is an important responsibility of a producer?

* topic management
* message serialization  
  Correct
* offset management
* message deserialization  
  Incorrect

## **Question 2 of 6**

How many active controllers are there in a Kafka Cluster with 10 brokers?

* 3
* 10
* 2
* 1  
  Correct

## **Question 3 of 6**

In order for a consumer to process the message correctly, it needs to \_\_\_\_\_.

* add a timestamp to the message
* know the format of the message beforehand  
  Correct
* convert the message to lowercase
* maintain a message cache

## **Question 4 of 6**

What is the relationship between Topics and partitions ?

* many-to-many
* many-to-one
* one-to-many  
  Correct
* one-to-one

## **Question 5 of 6**

Which of the following is NOT a function of a Kafka broker?

* data processing  
  Correct
* subscription management  
  Incorrect
* topics management
* replication  
  Incorrect

## **Question 6 of 6**

Messages that are stored in Kafka logs are \_\_\_\_\_.

* pruned periodically  
  Correct
* never pruned
* truncated for minimal space
* deleted instantly

### **Kafka client scripts**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter, we will explore the command line options in Kafka for managing topics. As for less publishing and subscribing. In this video, we will explore various client shell scripts that are available in Kafka. To begin with, we need to log into the Kafka container using the Kafka exec command. We do so by using the docker exec -it, Kafka-broker /bin/ bash command. This will take us into the Kafka container. Now let's navigate to the root directory for Kafka here. Kafka is installed under op bitnami Kafka. Let's explore the contents here. The config directory contains configuration that can be modified to suit the specific setup. Any configuration changes would require a restart of Kafka. Do note that these files are inside the container and restarting the container would reset the files to the original value. If you need to modify and use these files, use host volumes. The logs directory contains log files generated by Kafka and are useful for troubleshooting. The bin directory contains a number of shell scripts for Kafka management, publishing, and subscribing. Let's explore the directory in detail. The bin directory contains a number of shell scripts that can be used to interact with Kafka. The kafka-server-start.sh, and kafka-server-stop.sh scripts can be used to start and stop Kafka. We will explore other scripts later in the course. Now we can start creating some topics in the next video.

### **Creating a topic**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, we will create two topics in Kafka. I will first explore the topic creation command first. It's available in the chapter-3-commands.txt file inside the Kafka getting started project. The script to use for creating topics, is the Kafkatopics.sh file. What are the parameters for the script? First, we need to provide the link to Zookeeper. Topics are stored and managed in Zookeeper, so the creation command needs that information. Then comes the action, which in this case is create. We need to provide the name of the topic, with the topic parameter. While the name can be any string, it's recommended to use a pattern of qualified names for better management. In this case it's a tweets topic called Kafka.learning.tweets. We then specify two mandatory parameters. The number of partitions and the replication factor. We keep a partition size of one. We will discuss partitions later in the course. Replication is addressed in detail in the next Kafka course, Apache Kafka essentials, building scalable applications. For now, we keep the replication to one, since we only have one Kafka broker running. We now proceed to create the topic. We first need to log in to the Kafka container. Then we need to navigate to the bin directory. Let's run the creation command now. There is usually a warning to not use underscores and periods together. The topic is successfully created. Let's now proceed to create another topic in a similar fashion called Kafka.learning.alerts. This is also successfully created. Now, we proceed to explore the topics in the next video.

### **Exploring topics**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, let's list and explore the topics that we have already created. How do we get a list of topics in a Kafka cluster? We used the list action. This is similar to the create action. We will provide the zookeeper location as the parameter and use the list command. If we need further details around topics, we can use the describe command. Let's run these commands and see the results. Here is the list command. It only lists the topics that got created in the system. Do note that you will also get to see additional internal Kafka topics that start with an underscore. Let's execute the describe command now. Here, we get to see a lot more details about each topic. Specifically, it provides a partition count and the replication factor. For each partition in the topic, it also shows the partition ID, which in this case is zero. Partitions are usually numbered zero to N. It then shows the broker ID for the leader of the partition. The broker ID is set for each Kafka instance in the silver.properties file under the conflict directory. We will explore more of these later in the course. Now let's start publishing some data to Kafka in the next video.

### **Publishing messages to topics**

Selecting transcript lines in this section will navigate to timestamp in the video

In this video, I will demonstrate publishing of messages to Kafka using the command line publisher. For this, we will use the Kafka console producer.sh shell script. In order to publish, we need to provide the bootstrap server parameter, which can point to a list of Kafka brokers, we provide localhost 29092 as the value. As explained in the earlier setup video, we need to pay attention to the URL used. We are running this command from inside the container, which is within the Docker network. So we will need to use an internal host and port as the host of the same container. We can use localhost, the port should be the internal port, which is 29092, we need to specify the topic to which we are publishing data. There are other optional parameters available, which you can explore by simply using the shell script without any parameters. let's publish some data now. Let me just provide the shell script name, Kafka console producer daughter search. (typing) We get to see a list of parameters that are supported by the command. Now let's start the producer for Kafka dot learning dot tweets. We get a prompt for us to provide any message here. let's publish our first message. This is my first tweet. on pressing Enter This message is immediately sent to Kafka. We can now add more messages and keep publishing them. The programme will keep running until we press Ctrl C to abort it. This is a quick way to test if the Kafka topic has been created successfully and it is working fine. Messages once produced will be stored in Kafka until they are pruned. They can be consumed anytime. In the next video, let's consume the data produced.

### **Using console consumer**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] How do I consume messages from Kafka? We can use the Kafka-console.consumer.sh shell script for that. The command takes the mandatory bootstrap server parameter. The same considerations we discussed in the earlier video apply here, too. We need to provide a list of topics to consume data from. We have the option to either consume all the messages by using the front beginning parameter or only consume newly created messages. Let's execute this now. For demo purposes, I have four windows open. There are two producers windows on the left, and two consumer windows on the right. As seen in the previous videos, we create separate shell sessions on the container using the same docker exec command and then navigate to the Kafka bin directory. In both the producer shells, we start the Kafka producer for the given topic. In the first consumer shell, let me start the consumer and consume data from the beginning. We see that all the messages we have published earlier in the topic show up. This command will continue to run and listen for any new messages published to the topic. On the second consumer shell, let's start the consumer and only consume new messages. We see that there are no messages published from history and it simply waits for any new messages. Now let's go to the first producer and add a new message. It's called testing two consumers message 1. We see that this message immediately shows up in both the consumers. Now, let me go to the second producer and add a new message. We again, see that this message shows up in both the consumers. This demonstrates how Kafka can receive messages from multiple producers and send them to multiple consumers at the same time. This is a simple yet powerful demonstration of Kafka's capabilities. In the next video, let's explore the Kafka topic operations.

### **Topics management**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] The Kafka topics.sh shell script can also be used to modify and delete topics. Running just the Kafka topics.sh script shows all the parameters available to modify. Things like partition count cannot be modified though, and need to be planned ahead. To demonstrate some of this functionality, we will use the shell script to delete the Kafka learning alerts topic. We need to again provide the zookeeper parameter name and the topic name. The delete command takes care of deletion operation. Let's run this command. Running the Delete command returns a topic marked for deletion message. Deletion is not synchronous since deletion involves deleting all physical copies of data. Running a list command after some time will show if the topic is already deleted or not. In the next video, let's explore some internals of Kafka.

### **Kafka setup**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, we will review the Kafka setup and see how things are organized inside Kafka. Let's first start from zookeeper. We will log in to the Docker instance of the zookeeper container using docker exec -it zookeeper /bin/bash. Once inside the zookeeper, let's navigate to the zookeeper bin directory. The bin directory is /opt/bitnami/zookeeper/bin. Once inside here, let us execute the zookeeper client. We are now logged in to the zookeeper client. Let us clear the screen for now. Kafka creates multiple zookeeper nodes to keep track of its configuration and status. Running ls / command shows all the various nodes available. Running ls /brokers/ids shows all the brokers that are registered with zookeeper. We only have one Kafka broker registered for now, which is 1001. Executing a get /brokers/ids/1001 shows further details about the specific broker. We see the endpoints and ports registered. This information is used by brokers to identify other brokers in the cluster. Running ls /brokers/topics shows the list of all the topics currently available. This is the central information that is used to track topics. Please note that the \_\_consumer\_offset topic is internally created and managed by Kafka. Running a get command with /brokers/topics/ topic name shows further details about the topic. Let's now exit out of the zookeeper instance with the quit command. Let's exit out of this container. Now, let's get into the Kafka Docker instance. Executing a ps -ef |grep kafka command shows the Kafka process that is running on the system. It is a long command line, with the end of the command line showing the configuration file for Kafka. Let us look into the content of this configuration file. The key items to note here are as follows. The broker.id specifies the unique id for this instance. A value of -1 means that the id can be auto-generated by Kafka. In this case, 1001 is auto-generated. Next, we see the listeners configured based on the environment variables set using Docker Compose. The log.dirs parameter specifies the location where Kafka data logs are physically stored. Please explore, yourself, the other optional parameters further. Now, let us explore the Kafka data log directory as configured here. There are a number of \_\_consumer\_offset directories that manage consumption of messages by consumers. We will further discuss \_\_consumer\_offsets in the next chapter. Then for each partition for each topic, there is a data directory. For example, we have a data directory kafka.learning.tweets-0, where 0 is the partition id. Let's explore this directory further. We see the internal files in which data is stored. It's stored in an internal format. By doing a cat command on the log file, we can see bits of data that we have published to this topic. I hope this gives some overview on the internals of how Kafka works. In the next chapter, let's learn more about the partitions and consumer groups.

## **Question 1 of 7**

In which directory (under the Kafka root installation directory ) would you find the Kafka client scripts ?

* scripts
* tools
* bin  
  Correct
* lib

## **Question 2 of 7**

Which of the following is an optional parameter for creating a topic?

* retention.ms  
  Correct
* zookeeper
* topic  
  Incorrect
* partitions  
  Incorrect

## **Question 3 of 7**

If you have multiple producers and consumers, how can you route a message from a specific producer instance to a specific consumer instance?

* It is not possible.
* Use the producer-id parameter while consuming.  
  Incorrect
* Use the consumer-id parameter while publishing.  
  Incorrect
* Use the group-id parameter while publishing.  
  Incorrect



Replay

Review this video

Using console consumer

2m 26s

## **Question 4 of 7**

How many Leaders can a single partition have?

* 0 or 1
* 1  
  Correct
* Configurable
* 2

## **Question 5 of 7**

Which of the configuration parameters in the server.properties file is used to specify the physical log file location?

* log.dirs  
  Correct
* kafka\_log\_dir
* archive.folder
* retention.folder

## **Question 6 of 7**

Partitions in Kafka are numbered \_\_\_\_\_.

* 0 to n  
  Correct
* 1 to n
* randomly
* 1001 to 100n

## **Question 7 of 7**

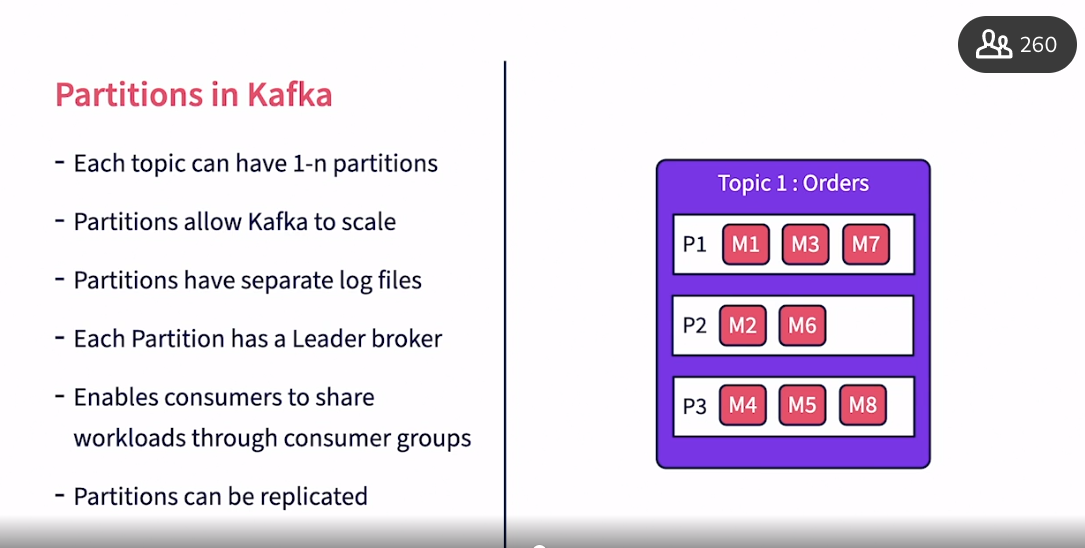
The bootstrap-server parameter is used to provide \_\_\_\_\_ to the shell command.

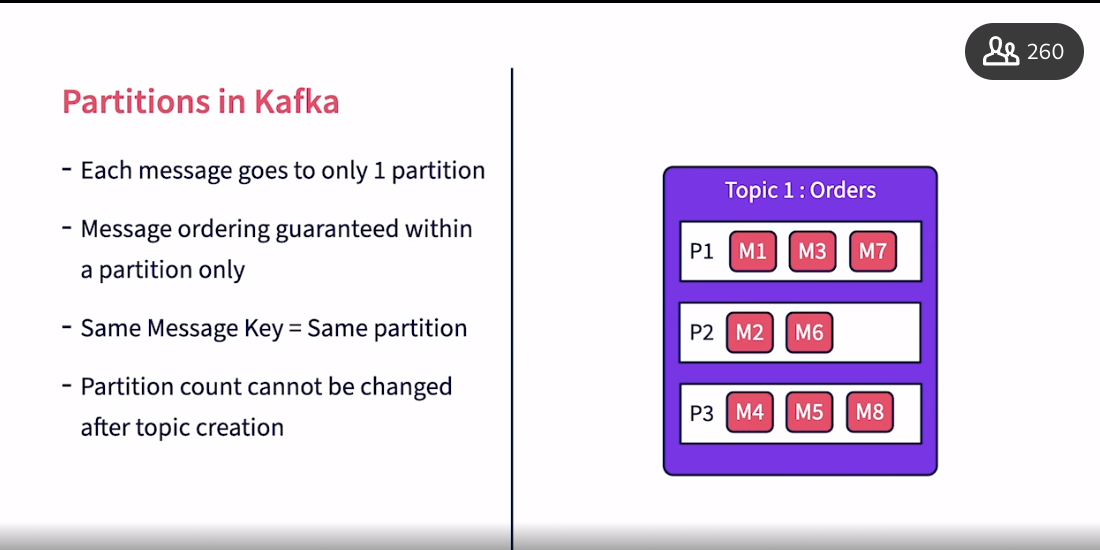
* a list of Kafka brokers  
  Correct
* a single Zookeeper
* a single Kafka broker
* a Zookeeper cluster

### **Intro to partitions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this chapter, we will deep dive into partitions and how they control the ingestion, storage, and consumption of messages. As discussed previously, each topic in Kafka can contain multiple partitions. A topic can have 1-n partitions. The number of partitions are set up during topic creation. The maximum number of partitions per cluster and per topic varies by the specific version of Kafka. Partitions allow Kafka to scale by parallelizing ingestion, storage, and consumption of messages. It provides horizontal scalability. However, creating too many partitions may result in increased memory usage and file handles. Each partition has separate physical log files which of course will rollover as they reach configured sizes. A given message in Kafka is taught in only 1 partition. Each partition is assigned a broker process, known as its Leader broker. In order to write to a specific partition, the message needs to be sent to its corresponding leader. The leader takes care of updating its log file as well as replicating that partition to other copies. The leader will also send data to the subscribers of the partition, but multiple partitions for a topic. Consumers can share workloads by partitions using consumer groups. Partitions can also be replicated for fault tolerance purposes. There are a few things to note about partitions. These significantly impact the performance and latency of pipelines. Each published message gets stored in only 1 partition. If the partition is replicated, each replicated copy will also get an instant of this message. Message ordering is guaranteed only within a partition. So in the example provided, messages are pushed to Kafka in the order from M1 to M8. M1, M3, and M7 are guaranteed to be delivered to the consumer in the same sequence as they belong to a single partition. But on the other hand, there is no guarantee of ordering between M1 and M2 as they belong to different partitions. It is possible for M2 to be delivered before M1. The partition for a message is determined by its message key. Kafka uses a hashing function to allocate a partition based on the message key. Messages with the same key will always end up in the same partition. It is important to choose keys that have an equal distribution of its values. Otherwise some partitions may get overloaded while others will be used minimally. The number of partitions cannot be modified after the topic is created. Hence, care should be taken to set the right size for partitions during creation time. Now let's look at creating topics with multiple partitions and then publishing messages to these partitions.





### **Creating topics with partitions**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Lecturer] Having looked into how partitions help Kafka scale, let's now create a topic with partitions. The commands for this chapter are available in the file, chapter-4-commands.txt, under the resources directory. In order to create multiple partitions, we simply need to specify the number of partitions needed in the partitions parameter. In this case, we are creating a topic called kafka.learning.orders, with a partition count of three. Let us execute this command in the Kafka container. To recollect, we need to use the docker exec command to log into the shell as shown in the previous chapters. The topic has been successfully created now. Let's explore this topic with the describe command. The describe command also takes the topic name to show only details for this specific topic. The output now shows one lane for each partition in the topic. There is a partition ID shown which is numbered from zero to two. Each partition also has the broker ID of its leader shown. The leader is responsible for managing the partition, including receiving messages, storing them in local logs and pushing them to subscribers. Given that we only have one broker in our setup, all partitions are assigned to the same broker. The controller in the Kafka cluster is responsible for assigning partitions to the brokers. In the next video, let us push some data with keys to the topic we have created.

### **Publishing with keys**

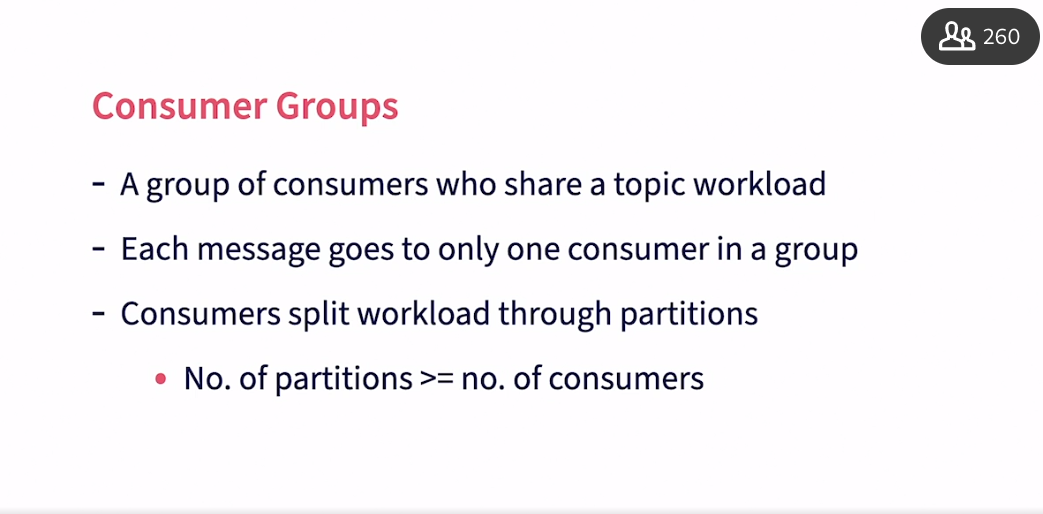
Selecting transcript lines in this section will navigate to timestamp in the video

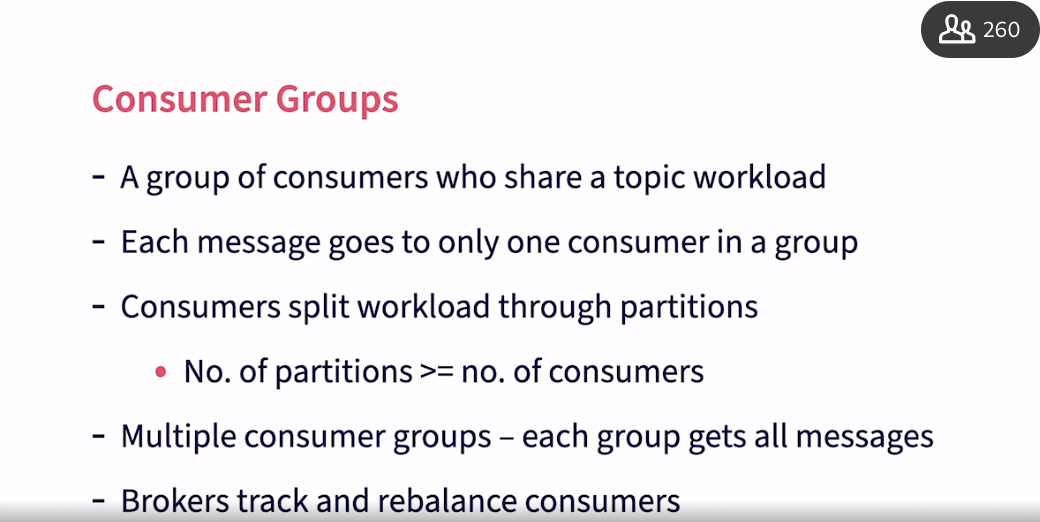
- [Instructor] In this video, let's use the console producer to publish some messages with keys for the calfcard.learning.orders topic. The command for publishing is the same as we discussed in chapter three, except for the topic name. In order to publish data with keys, we will use the colon operator to separate the key and the value. To specify this, we use two properties. The parse.key=true property tells the producer to read the key also. Key.separator is used to specify the separator character between the key and the value. Let's execute this producer and add some data. For the orders, we will use the order ID as the key and the product name and price as values. So we first provide the order ID as the key followed by the colon symbol and then provide a CSP string with the product name and value. Let's publish a second record with key as 1002, product as keyboard and value as 10. Kafka will now distribute these messages to various partitions. There is no command line available to see which partition each message ended up with. One way to explore this, is to use the data directory of Kafka. Let's navigate to the data directory. Let's look at the directories for this topic. We see that there are three directories created by each partition ID and we have files under each of them. Let's inspect these files under these partitions. Let's start with the first file on partition zero. In this, we see both the mouse and the keyboard records have landed up here. Let's inspect the next partition. There is nothing in this partition. There is nothing in the third partition also. How Kafka distributes data to these partitions is based on running a hash function on the key. As we keep publishing data into these partitions, the records will get distributed across all the three files. In the next video, let's explore consumer groups and how they consume partitions.

### **Consumer groups**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's discuss Consumer Groups in Kafka in this video. What is a consumer group? A consumer group is a group of consumers that share a topic workload. A topic may be generating thousands of messages in a short amount of time. It may not be possible for one single consumer process to keep up with processing these messages. For scalability, multiple consumer processes can be started and the messages can be distributed among them. For load balancing, a consumer group is a logical group of consumers that Kafka users for such low distribution. Each message will be sent to only one consumer within a consumer group, that consumer is then responsible for processing the message and acknowledging back to Kafka. Consumers split workload among themselves using petitions. Kafka, keep tracks of the active number of consumers for a given topic, it then distributes the messages evenly between these consumers. Kafka only considers the number of partitions for distribution, not the number of messages expected in each partition. It is expected that the number of partitions are equal to are higher than the number of consumers in a group. If there are more consumers than partitions, then there will be consumers with no work as the lowest granularity of work distribution is a partition. We can create multiple consumer groups each with a different set of consumers. Each group will get a full copy of all the messages, but each message will be sent only to one consumer within each consumer group. When new consumers come up or existing consumers go down, Kafka takes care of rebalancing the load by reassigning partitions among live consumers. Kafka users heartbeats with consumers to keep track of their health. Let's look at an example for consumer groups here. We have a topic called orders, it has three partitions P1 want to P3. There are eight messages numbered M1 to M8. There are two consumer group creator. The first consumer group is an analytics consumer group that has three consumers. The second is an audit consumer group that has two consumers. For the analytics consumer group as a number of partitions and consumers or equal, Kafka assigns one partition per consumer. For the audit consumer group as the number of consumers are less than the number of partitions, Kafka assigns one partition to the first consumer and two partitions to the second consumer, but both consumer groups get a copy of all the messages. In the next video, let us explore how Kafka keep tracks of what messages are consumed and if it needs to recent data towards consumers.

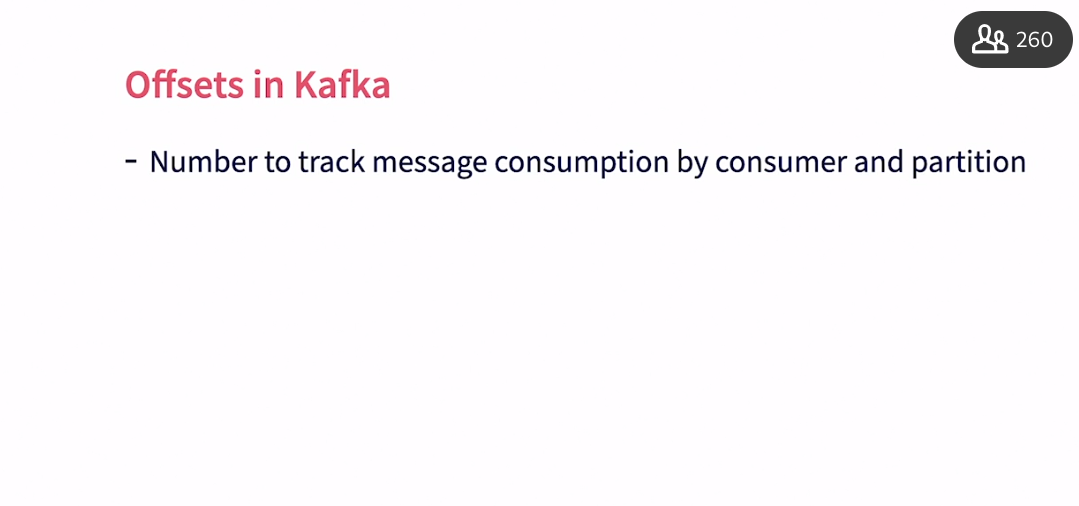


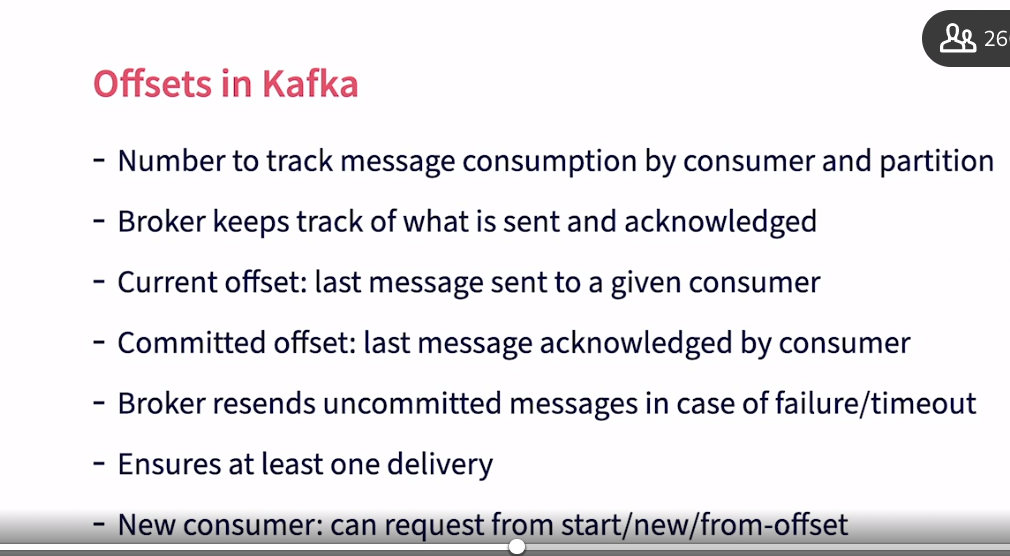


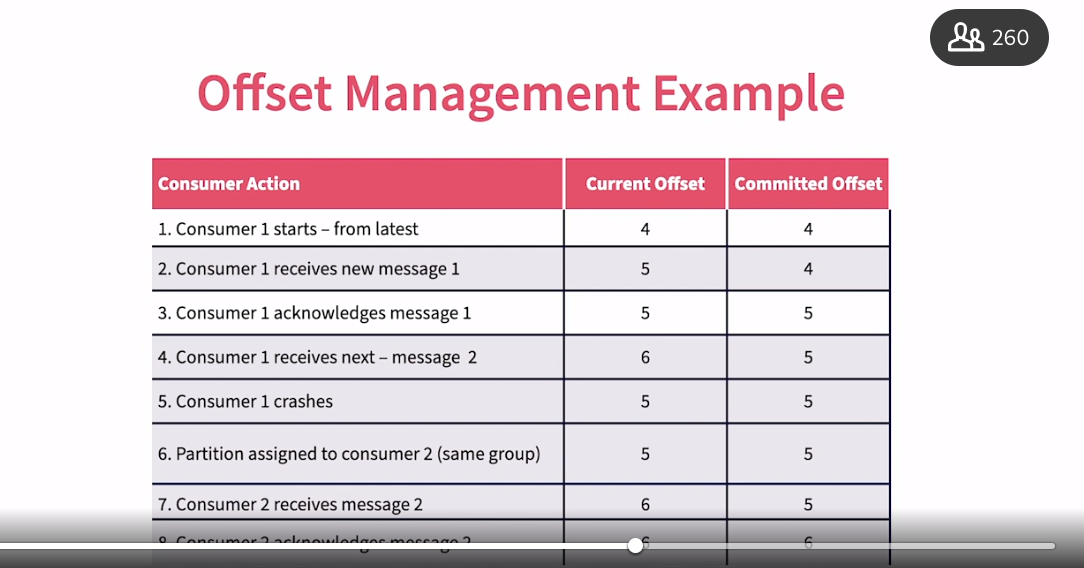
### **Consumer offset management**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] When Kafka pushes messages to consumers it also needs to ensure reliable delivery. Kafka ensures at least once guaranteed delivery using consumer offsets. What is a consumer offset? It is a number to track message consumption by each consumer and partition. As each message is received by Kafka, it allocates a message ID to the message. Kafka then maintains the message ID offset on a by consumer and by partition basis to track consumption. Kafka brokers keep track of both what is sent to the consumer and what is acknowledged by the consumer by using two offset values. The Current offset value tracks the last message ID that was sent to the consumer. The committed offset value tracks the last message that is acknowledged by the consumer. As consumers receive a message, they have the option of acknowledging immediately or after making sure that all required processing is done. This helps consumers to manage transactions and not lose messages in case of failures. By default, Kafka consumers auto acknowledge on receipt, but this can be changed by the consumer. When Kafka brokers do not receive acknowledgement within a set timeout they will resend the message to the consumer. This ensures at least once delivery of each message to a consumer group. A message can be delivered multiple times if acknowledgement does not happen within a timeout but each message will be delivered at least once. When a consumer group starts up, it has the option of requesting messages either from the start, only the latest, or from given offset. This allows the consumers to process messages based on their use case requirements. Let us look at an example of Offset management by Kafka. Let's say we have a partition being consumed by a consumer called Consumer One. Consumer one has set up his start from the latest message. So, in step one here, Kafka will set the Current and Committed Offset default which is the latest message at the time consumer one started. Step two, a new message, message one, comes into the Kafka partition which is then immediately sent to consumer one. The Current Offset is incremented by one to a value of five. The Committed Offset stays at four. Step three, consumer one acknowledges message one. Committed Offset is also incremented to five. Step four, under the message, message two, comes into the partition and is sent to consumer one. Current Offset increases to six. Step five, consumer one crashes. As a result, the brokers had it were the consumer's broken, so the broker resets the Current Offset back to five. Step six, a rebalancing step happens and the partition gets reallocated to another consumer, consumer two, which is within the same consumer group. Consumer two will start receiving messages from where Consumer one left off. Step seven, Consumer two will receive the message two that was not acknowledged by Consumer one, so the current Offset goes up to six. Step eight, Consumer two acknowledges message two. Committed Offset is updated to six. This process ensures that no message is left off and all are delivered to the consumers at least once. In the next video, let's execute an example of consumer groups and consumer offsets.







### **Consuming partitioned data**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, let's consume the data published into the kafka.learning.orders topic using consumer groups, and then track offsets. We will be opening four shell windows each of which is started by executing a Docker exit command into the kafka container and then navigating to the bender decree as shown in the previous chapters. In the first window on the left we start a publisher to publish into the orders topic. In the window on the top right, we start the first consumer to specify a consumer group. We use the group parameter and specify the group string. In this case, the group name is best consumer group. We specify the printer key property to print the key to the key.separator specifies the character used to separate the key and the value. We will use the equal to character for this. We will be subscribing from the beginning. Let's run this consumer. We immediately see all the messages we have published so far to this topic here. Both the key and the value are printed. Let us explore the consumer group status in the bottom left window we use the kafka consumer groups.sh shell script to check for status. We provide the bootstrap server list and request the described command for all the consumer groups. The output of the comment shows how the consumers are set up. The first column is the consumer group name. The next column is the topic name. The third is the partition ID. We will see one line showing up here for each partition. The current offset shows the latest offset of the message sent to the consumer. It is already updated for all the messages it does send to the first consumer. The log and offset is the offset for the last message received in the partition. This shows a value of two because there is only two messages sent so far. Lag is the difference between the login offset and the current offset. It is a key metric to track if messages are consumed immediately by the consumers or if there's a delay. Consumer ID is an auto-generated ID for the consumer. This shows which consumer is currently consuming this specific partition. Currently we only see one consumer. So it is consuming all the partitions. Now let's start a second consumer in the bottom right window with the same group name. When a second consumer starts, kafka immediately re balances and reallocates partitions. We do not see any messages in this window even though we have request it from the beginning. This is because all the messages are already sent to the first consumer in the consumer group. Now let's read on the consumer group command again in the bottom left window. Now we see the consumer ID for partition zero and one has been changed to the new consumer. Now let's start producing some new data in the producer and see how the data gets distributed to the two consumers. Let's start an order ID 1003:monitor with the value of 15.00. This gets delivered to the second consumer. You can see that it does not update it the first consumer though. So only one consumer in the consumer group always gets the message. Let's try another message. This goes to the second consumer too. Another new message, this goes to the first consumer. So as you can see, kafka keeps distributing the messages that are coming in to the two consumers that are available. Let's go back to the consumer group status and execute the command again. We now see that the offsets have been updated for all the partitions. Partitions zero and one have gotten two messages each whereas partition two has gotten only one message. This again, shows how Kafka distributes data among its partitions. In the next chapter, let's build producers and consumers using Java.

## **Question 1 of 6**

Which of the following is NOT true about partitions?

* A broker can be the leader for multiple partitions of the same topic.
* A message will only be stored in 1 partition.
* A partition can belong to multiple topics.  
  Correct
* A partition can belong to only one topic.

## **Question 2 of 6**

Kafka guarantees message ordering within a given \_\_\_\_\_.

* Broker
* Producer
* Kafka Cluster
* Partition  
  Correct

## **Question 3 of 6**

What are the limitations for changing the number of partitions after topic creation?

* It's not possible to change at all.  
  Correct
* It is only allowed for replicated topics.
* Only increases are allowed.
* It is allowed if there are no active consumers.

## **Question 4 of 6**

Which of the following is NOT a benefit of Offsets in Kafka ?

* It ensures at least once delivery.  
  Incorrect
* It is used to allocate partitions to consumers in a consumer group.
* Consumers need not be up all the time. They can wake up periodically and process data..  
  Incorrect
* It allows consumers to request messages from the beginning, only the latest or from an offset.  
  Incorrect



Replay

Review this video

Consumer offset management

3m 33s

## **Question 5 of 6**

In the output of kafka-consumer-groups.sh, the value of "lag" will keep increasing when \_\_\_\_\_.

* number of active producers increases
* consumers are not keeping up with the speed of publishers.  
  Correct
* number of consumers in a consumer group increases
* consumers have no pending data to consumer

## **Question 6 of 6**

What happens if the number of consumers in a consumer group is higher than the number of partitions?

* A given message will be delivered to more than one consumer.
* Some of the consumers will be starved, with no partitions allocated to them.  
  Correct
* Kafka will shutdown extra consumers.
* Kafka splits a partition and allocates to multiple consumers.

### **Kafka client libraries**

Selecting transcript lines in this section will navigate to timestamp in the video

[Tutor] - While building real world data pipelines, command line interfaces are hardly used to publish to and consume from Kafka. Kafka producers and consumers are written in various programming languages. Fortunately, Kafka has client libraries for a long list of programming languages. This website docs.confluent.io/platform/current/clients/index.html provides a list of client libraries for various programming languages. From here you can access documentation for various clients and also download these libraries. In this course, we will build sample Kafka clients using Java. For a Maven project, it is easy to add a Kafka client. In your pom.xml simply include the dependency for Kafka and it will download and set up Kafka for your project. Now let's build a simple producer for Kafka in Java in the next video.

### **Creating a producer in Java**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's explore the Java code for a simple KafkaProducer in this video. The code is available in the package, com.learning.kafkagettingstarted.chapter5. The classified list, KafkaSimpleProducer.Java. Let's explore the code now. To connect to Kafka, we need to create a properties object to specify the Kafka connection properties. There are three mandatory parameters required here. First, we need to use the bootstrap server property to specify a list of Kafka brokers. Even if we specify one broker, the Kafka client will discover other brokers in the system. It is, however, recommended to specify at least two so even if the first broker is out of service the second one can be used for discovery. Then, we need to specify the key and value serializer\_classes. These classes are invoked by Kafka client to serialize the message and value into bytes. In this case, we are specifying a string serializer as both these values would be strings, in our case. Custom serializer's can be built, if needed. We create a KafkaProducer object using these properties. Now, we are going to publish 10 random messages to Kafka starting with the random key value. We do an iteration for 10 cycles. We then create a producer record in Kafka. The type belongs to the types of the key and value, both of which are strings. As parameters, we first specify the topic name, the key for the message and the value for the message. Value in place is the content of the message. We will be publishing to the kafka.learning.orders topic which we created earlier in the course. We will then print the message that we created. Then, we call the Send method on the KafkaProducer to push the record to Kafka. We will continue to publish after sleeping for 10 seconds. In this case, we will be doing 10 cycles and publishing 10 messages. In the next video let's run this code and review the results.

### **Publishing messages in Java**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let us run the producer we created in the previous video and review the results. Let's run this code now. If you see this Edit configuration page, then go down to the classpath for module and select the KafkaGettingStarted module, then hit the Run button. We can see the producer record printed here as publishing goes on. Ignore the values for the partition and timestamp. Those will be populated when you create advanced producers by manually setting these values. Note that multiple producers return in different programming languages can publish to the same topic simultaneously. For Kafka, everything is a sequence of bytes. In the next video, we will build a consumer to consume this data.

### **Creating a consumer in Java**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, we will build a consumer in Java to consume messages from the kafka.learning.orders topic and print them to the console. The code for this is available in the KafkaSimpleConsumer.java file under the chapter five package. Setting up a consumer is similar to setting up a producer. We first provide a list of Kafka brokers with the bootstrap servers parameter. Then we need to provide deserializers for the messages. Messages received from Kafka are byte arrays, and the Kafka client will call the deserializer to convert the key and value to equivalent Java objects. In addition, we need to specify the consumer group name in the group ID parameter. We use Kafka Cava Consumer as the name. We use the auto offset reset config parameter to specify whether we want all the messages in the topic, or only new messages. In this case, we specify the earliest as we are requesting all messages from the topic. We then create a Kafka consumer with these values. We use the subscribe method on the consumer to specify a list of topics to subscribe to. We specify kafka.learning.orders here. Kafka consumers work by polling the Kafka brokers for any new data. To set up polling, we use a continuous loop. Inside this, we will pull Kafka for new messages with the 100 millisecond timeout. If there are any new messages that are delivered during that period, it would be populated in the messages object. We then iterate over this list and print the messages that are returned. In real world use cases, we would then proceed to process the message as required. It is usually recommended to use a separate thread for actual processing, so we do not block the consumer thread. In this case, the client automatically acknowledges the receipt of the message once it is received. If we need to wait until the message is successfully processed, there are additional setup parameters and methods available. In the next video, we will execute this code and review the results.

### **Consuming messages in Java**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's execute the Kafka consumer, we built in the previous video and review the results. Let's run the code now. We immediately see that all previously published messages are returned to the console. For each message, we see the topic name, the partition this message belongs to, and the offset for the message. We see the timestamp of the message as create time. The publish key and the value are also shown. We also see that the program continues to run, and wait for more data. Let's stop and restart the program again. Even though we specified it to publish from the earliest offset, it now does not print any of the old data. This is because based on the previous run, Kafka has incremented the committed offset for this consumer group, and will only provide new messages going forward. To force it to publish from the beginning, we need to request from a specific hard-coded offset of zero. Please check out the other consumer options available under the consumer conflict for this. Let's now run the producer also side by side. Now as the producer is publishing new data, they are immediately received and printed on the consumer. There can be multiple threads inside the same program, or multiple program that can consume from the same consumer group and share the workload. This completes our discussions around Kafka, and its basic capabilities. In the next chapter, you will test your learning by implementing your use case.

### **Kafka basics use case: Problem definition**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Kumaran] In this chapter you will implement a simple use case in Kafka to exercise the learnings in the course. Beyond this use case I also recommend you to try out some of your own use cases. We have three exercises to do in this course. First, create a topic called, Kafka.usecase.students. Assume that this topic would be used to publish student's course. It will have two partitions and an application factor of one. Second, use the console producer to publish messages to this topic. Publish both key and value. Use a console consumer to consumer suggest from this topic. Finally, build a producer and a consumer in Java for the same topic. Best of luck for the use case. Once you have completed the use case, you can also check the other videos in this chapter for the solutions.

### **Setting up topics**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Male Narrator] In this video, let's create a topic for the usecase, using the Kafka command line. The commands for this chapter are available under the resources directory in chapter-6-commands.txt. Let us go and create the topic now. In order to create the topic, we use the kafka-topics.sh command. We specify the topic name as kafka.usecase.students, and the partition counts as two, as expected from the usecase. Once we have created the topic, we can check if the topic has been successfully created using the descript command. We can see that the topic shows up here with a partition count of two and there are two lines shown here, one for each partition that has been created. Each partition has the same data rest 1001. Let's now proceed to publish and subscribe from this topic in the next video.

### **Producing and consuming data with scripts**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Continuing on the topic creation from the previous video. We will now publish messages and consume them using command line scripts for this use case. To publish data, we use the kafka-console-producer.sh. We specify the topic name as kafka.usecase.students. Since we are publishing with the key, we need to specify the parse.key parameter and the key.separated parameter. Let published some data now. Key would be 1001 and the values will be just a student name. Next, to consume messages from the topic we use the kafka console consumer. Again, we need to specify the printer key property and the key.separated property. We also can specify a group name to enable kafka to track consumption of data. On running this command, we see all the data that we have published so far to this topic, printer to the console. Now let's go and publish another record here. And we see that that record immediately shows up in the console also. Let's now proceed to implement producers and consumers using Java.

### **Producing and consuming data with Java**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, let's look at the final exercise for the course. Producing and consuming messages in kafka. The code for this exercise is available in the package com.learning.kafkagettingstarted.chapter6. It has two files. Kafka use case producer and kafka use case consumer. The code is simple and resembles the producer and consumer we built earlier in the course. Let's run the producer and consumer now. We see the results as expected. The base template for producers and consumers are the same. When you build production grade code though, you will need to change your implementation based on few things. On the producer side, this code currently blocks until the producer actually publishes to kafka. So this may not scale well. You may want to explore a synchronize publishing capabilities. On the consumer side, you may want to launch processing on a separate thread so the consumer is not blocked. You may also want to manually acknowledge receipt of the message after the message has been successfully processed. We will explore these advanced use cases in our next kafka course. Apache Kafka Essentials: building scalable applications.

### **Next steps**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Kumaran] Now that you have learned about the basics of Apache Kafka. You can take your learning even further. First, you can learn more about building scalable Kafka applications through the next course. Apache Kafka essentials, building scalable applications. Second, you can learn more about using Kafka for building stream processing biplanes with Apache Spark, Apache Flink or Kafka Streams. Finally, you can implement Kafka in your organization for one of your big data processing use cases. Data always intrigues me. I bet it intrigues you too. So, lets keep exploring it and find better ways of extracting knowledge out of it.