Course Overview

Course Overview

Hello and welcome to my course, Getting Started with Apache Kafka. I am Ryan Plant, a software engineer, solutions architect and, in general, a distributed systems addict. Chances are, you have a profile on LinkedIn and frequent the site often. You're also likely to be a subscriber to Netflix and have interacted with its service on your smartphone, web browser or TV. Perhaps you've gotten a ride with Uber or stayed in an Airbnb rental. If any of these are the case, congratulations! Whether you knew it or not, you're a user of a system called Apache Kafka. Apache Kafka provides the messaging infrastructure of these and many more massive software as a service applications we use every day. Each of these services produce hundreds of billions of messages that amount to several 100 terabytes of data being moved around and consumed per day. That's truly big data. Whether you're a small or large enterprise in this day and age, you're likely drowning in data and struggling to figure out how to move it where it needs to be to enable an ever increasing number of use cases. It's not an easy problem, but this is why I produced this course to introduce you to Apache Kafka and provide you with an overview of how it can help you solve these problems today and for the future. In this course, you will get a thorough understanding of Apache KAFTA's architecture and how it has adopted proven distributed systems design principles that enable it to scale and perform reliably. We will break down this architecture into individual components and cover each in great detail. We will demonstrate the components in action with common scenarios and walkthroughs. How Apache Kafka Solutions can be developed in Java. By the end of the course, you will have an understanding and appreciation for why Apache Kafka is taking the industry by storm, but most importantly, you will come away with confidence and knowledge to build your own next generation big data solutions with Apache Kafka. I hope you will invest your valuable time and allow me to show you how to get started with Apache Kafka.

Getting Started with Apache Kafka

Enterprise Challenges with Data

Hello. Welcome to the course, Getting Started with Apache Kafka. My name is Ryan Plant and I am the course author. With this first module, my objective will be to answer the question. Why Apache Kafka? Like most things, to understand the why, we first need to learn the what. Apache Kafka is all about data and getting large amounts of it from one place to another rapidly, scalably, and reliably. Throughout this course, I'll refer to this as data movement or data logistics. In computing, a common term for transferring data is messaging, and that's how Apache Kafka would describe itself, as a messaging system. But unlike other messaging systems, Apache Kafka is tailored for high throughput use cases where vast amounts of data need to be moved in a scalable, fault tolerant way. So why is something like a Kafka needed? This is what a lot of enterprises look like. Okay, I admit it. It may be a little dramatic, but if you think about large companies, your own possibly, there are hundreds of applications all needing data to operate. Now, whether it be creating logs, records and databases, key value pairs, binary objects or messages all of these applications are creating data at an incredible rate. Oftentimes, that rate can strain existing data stores and require more stores to take on the load. When that happens, you have issues related to getting the data where it needs to be and enabling applications to find it. Furthermore, as businesses change, the variety of the data increases, making the types of applications and data stores change as well. Now, this obviously doesn't happen overnight, but it happens, and the result becomes a complex web of point-to-point data movements that are very hard to manage and work with. In this common enterprise scenario, there are a lot of tools and methods being used to make this complex distribution topology possible. Most of these have been in the technology toolbox for decades, and each comes with its fair share of trade offs. Let's take a look at them. Probably the most common is database replication and log shipping. Obviously, it could be useful, but this is limited to a certain kind of data movement between relational databases that support replication. And that's it. The way a database implements replication is very specific to the database and therefore doesn't work across vendors. So in a heterogeneous database environment, this becomes a limitation. As a point-to-point integration, there is a significant amount of coupling between the source and the target. Changes to the schema have a direct impact on replication, so as your requirements change, the ripple effect can introduce challenges to your replication architecture. For log shipping, performance could be a challenge, depending on how big the log is that you're trying to ship and the frequency. But overall, these methods, while somewhat functional, are cumbersome to manage and maintain and really only apply to a certain type of datastore. For integrating data between different sources and targets, ETL or extract, transform and load options are implemented, but not without its drawbacks. Enterprise-grade ETL is typically very costly and, in some cases, proprietary. In recent years, viable open source options have sprung up, but nonetheless, ETL tools and infrastructure usually requires a bit of training and ramp up time to use productively. Every ETL job that runs is a custom application written by a developer who specializes in ETL. As the data environment increases in complexity, so do the jobs and as most ETL systems centralize the execution of these jobs, the performance and scalability become strained as concurrent or sequential jobs compete for the limited resources, which may require multiple ETL environments to exist, which further increases the complexity.

Messaging Limitations and Challenges

The next two areas are where we'll spend a bit more time because, as I said, Kafka is a messaging system. So discussing how current messaging systems are applied to enterprise scenarios will help you understand how and why Kafka is a viable tool to consider in your modern day toolbox. Messaging makes a lot of sense because it establishes a fairly simple paradigm for moving data between applications and datastores. However, when it comes to a large scale implementation, traditional message systems can struggle, namely with scalability. The means to collect and distribute data as messages relies on the role of a messaging broker, which is oftentimes a bottleneck for reasons we'll cover shortly. Additionally, there are a lot of variables that determine the reliability and performance of a messaging system, a big one being message or data packet size. Larger messages can put severe strain on message brokers, and this is a challenge because you may not be able to control messages coming from some systems. Furthermore, a messaging environment is dependent on the ability for message consumers to actually consume at a reasonable rate. There is also the challenge of fault tolerance. Think about it, if a consumer pops something off the queue or reads it from a topic. It's probably gone. So if the consumer loses the message or processes it incorrectly, it is extremely difficult to get it back to

reprocess. Let's go into some of these issues a little further, since its key to understanding how Kafka provides a better messaging system. Under ideal circumstances, you have applications serving as publishers of messages and a broker that is like a mailbox whose job it is to deliver or make available to messages to consuming applications, which consume their messages at a reasonable rate. But under higher volumes and varieties of message sizes, the publishing applications can run amok and blast the broker with messages. If the applications have not implemented some sort of throttling, the broker can be put into a tough situation fast. Now, most messaging systems are implemented on a single node or host, which generally relies on a limited amount of local or quota storage. Generally, this isn't a problem as messaging systems are usually very efficient, provided they could turn over the messages they're receiving fast enough before the storage becomes limited. This happens when you have lazy, slow or unresponsive application consumers. For whatever reason, the result can be an outage of disastrous proportions. The brokers disks get full, the broker croaks, becomes unresponsive, and now you're publishing applications can't publish their
messages. And, depending on the error handling, can cause a complete denial of service of the application altogether. Another category of peril is with regard to application faults in the consuming applications. Faults can happen for any reason, but a common reason is a bug of some sort. Where this becomes a problem is when the bug incorrectly processes the message it is getting from the broker either via a queue or a topic. Why is this a problem? As we mentioned, the broker's job is to turn over the messages. It doesn't and can't keep them around for very long. So if a consumer consumes the message, processes it incorrectly and poisons data, it can't go back to retrieve the message again because it's not there. Of course, the consumer wouldn't have to do it if it stashed all of the messages somewhere, but that isn't always the case. Even so, the work to retrieve the message again and reprocess it once the bug has been smashed is a lot of work, and it may be too late. Technically, messaging systems are considered a form of middleware. In this case, I'm referring to more custom brokering solutions where you need to write complex logic to handle data movement between applications and data stores. This is where your code needs to have intimate knowledge of every datastore, and that knowledge will likely be specific to the datastore type and provider. Furthermore, you will likely be in the realm of dealing with
distributed coordination logic, multiphase commits and error handling to consistently manage data. Anyone who has lived this world will tell you it is extremely complex, and it never ends. With every application change, new datastore, new schema, you have to revisit this
code, which is deceiving because on a white board it sounds like a great solution, and it is tempting to pursue and may even work out under simple conditions. But when you have to handle multiple sources of data at different velocities and use cases, you run into challenges maintaining data consistency. The more distributed your system gets, the harder it is to enforce strict consistency. Writing your own middleware, maybe cheap at first, but when considering the overall total cost of ownership of a complex code base, the costs are high. An alternative is to employ a vendor's middleware solution, which may or may not work for all of your scenarios and in itself can become expensive. Let's take a look at two leading patterns for using middleware for data movement. The first pattern is a multi-write scenario where your application relies on code written somewhere to handle data flows transactionally on two or more different datastores. As I said before, this requires substantial care and maintenance for it to work reliably without data consistency issues. For example, if a target database is not available for whatever reason, and the transaction cannot commit, what should happen? There are many different approaches to this, but if not careful, it could lead to data inconsistency where the second transaction commits without the first. Performance and

scalability challenges come from this pattern as well, because transactional consistency requires all participants to commit that can cause holds on the part of the application and database. Furthermore, if one were to scale out to more resources to share the load it would mean that more and more would have to be party to the coordination logic. This can get out of hand fast. An alternative could be to leverage a messaging broker in the middle to coordinate application data movements to stores and vice versa. This would be in line with the messaging scenario we discussed earlier and therefore subject to the same issues of slow consumers or unavailable data stores. This pattern is also difficult to scale out as more consumers intending to share the load, may compete for access to messages and may not coordinate consistently across their peers.

LinkedIn's Search for a Better Solution

This is the typical enterprise challenge when it comes to handling growing data sets, moving faster and faster through systems. Surely there has to be a better way to move data cleanly without a complex web of different integration topologies, reliably as to reduce the impact of any one component slowness or availability on the system quickly as data movement and access is only getting faster for real-time use cases and finally, autonomously reducing the coupling between components so we can improve or change parts of the system without a cascading effect. Is there a better way? Well, it just so happens that in 2010 LinkedIn asked that same question. Now, most of you, if not all of you, know what LinkedIn is. It's a massive social network for professionals. Over the years since its founding in 2003 it has grown exponentially. They are the epitome of big, fast and varied data challenges. Here are but a few of the most recent stats to give you a sense of the scale that they work with on a day-to-day basis. LinkedIn's site offers many different features and capabilities, all provided by a portfolio of data creating and consuming applications. Their ability to direct, manage and utilize the data has been a direct enabler of their success. But this data handling ability wasn't always there. In fact, up until 2010 things were quite ugly and getting uglier. Remember this picture? Well, it more or less represented, LinkedIn before they invented Kafka. Under the pressure of hypergrowth, LinkedIn needed to find a better way to deal with data. I suspect many of you actively use LinkedIn and therefore will know how much functionality is available on their website and through their mobile apps. All of this functionality is in self-contained applications that autonomously produce and consume data from the data infrastructure. Over the years, as more applications were written, more users used the site. A lot of technology was used to get data where it needed to go. LinkedIn had several relational database types multiple node SQL stores, queueing systems, log processors, you name it. Generally, all that was at their disposal to handle their growing data problems were the tools and methods that we discussed earlier, which LinkedIn found woefully inadequate for solving their data problems. Given the growth, you could imagine how important it was for LinkedIn to come up with a better way to make all of this data available without slowing down, crashing or further limiting the system. And, of course, this is where a Kafka comes in, where it started as an internal project in 2009. Incidentally, you may be wondering why LinkedIn named their solution Kafka. It refers to the German language writer, Franz Kafka, whose work was so freakish and surreal it inspired an adjective based on his name. In the case of LinkedIn their data infrastructure and the ability to work with it had become so nightmarish and scary that they named their solution after the author, whose name would best describe the solution they were hoping to escape from.

Apache Kafka as a Viable Solution

To guide their design, LinkedIn set forth some goals that a solution must meet in order to be a viable tool. First and foremost, it needed to be able to handle large volumes of data in the terabytes and beyond. They knew that in order for this to be accomplished, it would need to be designed to scale out by adding machines to seamlessly share the load. They couldn't afford a lossy system. Data had to be reliably managed, transmitted and made durable in the case of failure. It was important for LinkedIn that all application producers and consumers be loosely coupled but engage in common data exchanges. It would be unacceptable, for one application's runtime conditions to affect another's. To enable this loosely coupled paradigm between producers and consumers, they wanted to embody common and simple messaging semantics of publish-subscribe. Independent data producing applications would send data on a topic, and any interested consuming application could listen in and receive data on that topic, which it could process and in turn, produce on a different topic for others to consume. At a high-level this is what the data architecture looked like once Kafka was in place. Now, the real details of how this works will be covered in the next few modules. But you can see what role Kafka plays with respect to different data producers and consumers. As a central broker of data, Kafka enables disparate applications and data stores to engage with one another in a loosely coupled fashion by conveying data through messaging topics, which Kafka manages at scale and reliably, regardless of the system, the vendor, the language or runtime all can integrate into this data fabric provided by none other than Apache Kafka. Because of Kafka, LinkedIn was able to successfully weather their storm of data and establish a foundation upon which to build their next generation of applications and services. It has been openly admitted by LinkedIn engineers that without Kafka, LinkedIn would not have been able to sustain their growth and achieve the valuation they have today. As we discussed, Kafka's development started in 2009, and its first deployment was in 2010. Within the next year, LinkedIn hardened Kafka to a point that they felt it could be released as an open-source project under the Apache Software Foundation. This they did in 2011. Very soon after its submission to the Apache incubator, it achieved top-level status and has become one of the most adopted tools in the Apache ecosystem. Today, Kafka is responsible for driving 1.1 trillion messages per day at LinkedIn alone. But there are many more big name companies that have adopted Kafka to solve the common problems we discussed at the onset. Here are just a few of the most noteworthy names. All in all, Since 2015 Apache Kafka's adoption rate has grown 700%. As the software development community contributes more and more capabilities to its code base, many of the more recent innovations that Kafka has produced will be touched on in the last module, and likely in additional courses. For now, let's get more into the guts of Apache Kafka and find out just how it can pull off such amazing data movement feats, not only for these enterprises listed here, but for your own. Before we move on to the next module, let's quickly summarize what we've covered in this module, as it self-describes on its website, Kafka is a distributed messaging system designed for high throughput. It was intended to address many of the shortcomings that traditional data handling tools and methodologies have, particularly when data is growing and it needs to move faster through more and more diverse systems. Which was the cause at LinkedIn in 2009 when it decided to invest R and D effort to solve its data problems. Luckily, LinkedIn did a great job and was kind enough to open source it in 2012 and make it available under the generous Apache license. Now, many enterprises and internet scale companies can take advantage of Kafka and many are, using it as LinkedIn does for addressing their complex data infrastructure and preparing themselves to take advantage of the future opportunities for analyzing large amounts of data.

Getting to Know Apache Kafka's Architecture

Apache Kafka as a Messaging System

In the last module, I raved about Apache Kafka and why it is a breakthrough tool for managing data movement at scale. To support my claims, I cited many large scale companies that use it and even gave some impressive statistics that back up Kafka's ability to move data. That's all nice. But now we're going to start getting deeper into Apache Kafka and learn just how exactly it is able to deliver on such abilities. This module will start by examining the overall architecture of Apache Kafka. As I highlighted in the first module, Apache Kafka is truly a messaging system. More specifically, it is a publish subscribe messaging system in a pub subsystem, there are publishers of messages and subscribers of messages. A publisher creates some data and sends it to a specific location where an interested and authorized subscriber can retrieve the message and process it. In Kafka, we call these traditional publishers something slightly different - producers, and the subscribers we call - consumers. As we'll see in the upcoming modules, producers and consumers are simply applications that you write or use to implement the producing and consuming APIs. Now the producer sends its messages to what I said was a specific location. In Kafka, this location is referred to as a topic, which is really a collection or grouping of messages. Topics have a specific name that can be defined up front or on demand as long as producers know the topic name and have permission to send to it, messages can be sent to that specific location. The same goes for consumers. Consumers retrieve messages based on the topic it is interested in. This should be very familiar for those with experience using pub sub messaging. As these concepts were intentionally carried forward into Kafka due to the simplicity of the paradigm. The messages and their topics need to be kept somewhere, after all, they are physical containers of data. The place where Kafka keeps and maintains topics is called the broker, as it is in other messaging systems. We'll cover topics in greater detail later, but for now let's look closer at the Kafka broker. The Kafka broker is a software process also referred to as an executable or daemon service that runs on a machine, a physical machine or a virtual machine. A synonym for a broker is also a server, but I like to avoid using the term server, since it has a tendency to be overloaded. The broker has access to resources on the machine, such as the file system, which it uses to store messages, which it categorizes as topics. Like any executable, you can run more than one on a machine, but each must have unique settings so that they don't conflict with one another. We'll come back to this shortly. It is in the Kafka broker, where the differences between other messaging systems become apparent.

The Apache Kafka Cluster

How the Kafka broker handles messages in their topics is what gives Kafka its high throughput capabilities. Achieving high throughput is largely a function of how well a system can distribute its load and efficiently process it on multiple nodes in parallel. This is a hallmark of scalable design. With Apache Kafka, you can see scale out the number of brokers as much as needed to achieve the levels of throughput required. And all of this without affecting existing producer and consuming applications. In fact, LinkedIn as of April of 2016 has publicly stated that they have upwards of 1400 brokers processing over two petabytes per week. You simply can't find another messaging system out there with that capability. I am not satisfied yet, though, and I hope you're not either. Let's continue to dive deeper and understand what about Kafka's architecture enables it to scale so well and furthermore, how it can achieve such high levels of reliability. It's time to introduce another key concept within the Apache Kafka architecture and that is the cluster. A Kafka cluster is a grouping of multiple Kafka brokers. As I said, you can have a single broker or multiple brokers on a single machine or brokers on different machines. For example, if you had only a single broker on a machine and only one machine, it would be said that you have a cluster of one. If you have two brokers running on the same machine, it would be considered a cluster of two. This would be the same if each broker was running on its own machine, you still would have a cluster of two. The important thing to remember here is that a Kafka cluster is just a grouping of brokers. It doesn't matter if they're running on their own machines or not. What matters is how independent brokers are grouped to form a cluster. The grouping mechanism that determines a cluster's membership of brokers is an important part of Kafka's architecture, and what really enables its ability to scale to thousands upon thousands of brokers and be distributed in a fault tolerant way. For the sake of putting down a placeholder, this is where Apache ZooKeeper comes in

Principles of Distributed Systems

At this point, it is important to discuss a little theory about distributed systems. But along the way I will do my best to associate the principles to Apache Kafka. So you'll understand how brokers organize into clusters and how Kafka clusters achieve amazing feats of scalability and reliability. First, I'm going to give you my general definition of a distributed system for the purposes of describing how Kafka, as a distributed system, works without spending too much time discussing philosophy. Let's just say a system is a collection of resources that have instructions to achieve a specific goal or function. A distributed system is one that consists of multiple independent resources, also known as workers or nodes, sometimes even called worker nodes. Obviously, the reason there are multiple nodes is to spread the work around, presumably to get more done than what could otherwise be achieved with less. But in order to do that, there needs to be coordination amongst all of the available working nodes to ensure consistency and optimal progress towards the overall task or goal at hand. Without coordination, it would be difficult, if not outright chaotic, to divide up the work appropriately and track progress as to ensure the most optimal use of resources, for example, proper coordination can avoid duplication of effort or individual worker nodes, undermining each other's work without knowing it. Of course, coordination isn't possible without clear and thorough communication between all components within the system. In Kafka, these worker nodes are the Kafka brokers. Within a distributed system, there are different roles and responsibilities. And like any organization of able workers, there is generally a hierarchy that starts with a controller or supervisor. A controller is really just a worker node like any other. It just happened to be elected from amongst its peers to officiate in the administrative capacity of a controller. In fact, the worker node selected is the controller is commonly the one that's been around the longest. But other factors could be considered as well. I'll have to leave it at that, as this specific topic is beyond the scope of this course. Once selected, the controller has some critical responsibilities. First, to maintain an inventory of what workers are available to take on work. Second, to maintain a list of work items that has been committed to and assigned to workers, and third to maintain active status of the staff and their progress on assigned tasks. Once a controller is established and the workers are assigned and available, you could say a team is formed and work can now be distributed and executed. In Kafka, this team formation is the cluster, and its members are brokers that have assigned themselves to a designated controller within the cluster. When a task comes in as an example from a Kafka producer, the controller has to make a decision which workers should take it. There are a lot of factors at play here, many of which are beyond the scope of this module, but let's cover the most important ones.

Reliable Work Distribution

First, the controller needs to know who is available and in good health. And very importantly, the controller needs to know what risk policy should govern its assignment decisions. A great example for risk policy is the redundancy level, the thing that determines what level of replication to employ in case an assigned worker fails. In a distributed system that offers redundancy options, it has to ensure that if work is assigned to a worker and that worker becomes unavailable, the work assigned or the work already done is not lost. That means each task given to a worker must also be given to at least one of the worker's peers in the event of an unexpected catastrophe. But amongst a group of worker nodes, which one will get assigned? This is how it works. For an assignment, if the controller determines redundancy is required, it will promote a worker into a leader which will take direct ownership of the task assigned. It will be the leader's job to recruit two of its peers to take part in the replication. In Kafka, the risk policy to protect against loss is known as its replication factor, and we will cover this in more detail within the next module. Once peers have committed to the leader, a quorum is formed, and these committed peers now take on a new role in relation to a leader, a follower. If, for whatever reason a leader cannot get a quorum, the controller will reassign tasks to leaders that can. Up until this point in covering the principles of distributed systems and how Apache Kafka applies them, we have been using the term work somewhat generally. In Apache Kafka, the work that the cluster of brokers performs is receiving messages, categorizing them into topics and reliably persisting them for eventual retrieval. As we have already discussed, the components that drive this work are producing applications called producers that send the messages to the cluster and consuming applications, called consumers that retrieve and process messages. Comparatively speaking, the effort required to handle messages from the producers is substantially less than the effort required by consumers. We will talk about this more in modules four and five. I just wanted to bring this up to highlight that both add load to the cluster, but in very different ways

Distributed Consensus with Apache Zookeeper

Virtually every component within a distributed system has to keep some form of communication going between themselves. In distributed computing terms, this communication is referred to as a consensus or gossip protocol, and without it distributed system simply cannot operate. Besides the obvious data payloads being transferred as messages There are other types of network communications happening that keep the cluster operating normally. For example, events related to brokers becoming available and requesting cluster membership or broker name lookups. Retrieving Bootstrap configuration settings and being notified of new settings consistently and in a timely fashion. Events related to controller and leader election and health status updates like heartbeat events It is time to talk about Apache ZooKeeper. I have to give a big disclaimer, though Apache ZooKeeper could use its own course, so we can't spend a lot of time on it here. It is used in a variety of distributed systems. For all the reasons we have been discussing in the context of Apache Kafka, specifically, ZooKeeper serves as a centralized service for metadata about vast clusters of distributed nodes needing Bootstrap and runtime configuration information, health and synchronization status, and cluster and quorum group membership, including the roles of elected nodes. Some rather large distributed systems depend on ZooKeeper, like Apache Hadoop HBase, Mesos, Solr, as well as Redis and Neo4j database is they all use it to enable their scale out fault tolerant capabilities. ZooKeeper itself is a distributed system, and for it to run reliably has to have multiple nodes, which form what is called a ZooKeeper ensemble. An ensemble is like saying a cluster for ZooKeeper. In the case of Kafka, because of the type of work a ZooKeeper ensemble performs, it is generally not needed to have more than one ensemble to power one or many Kafka clusters. Let's bring it all together in a logical view. At the heart of Apache Kafka, you have a cluster which, as we discussed, consists of possibly hundreds of independent brokers closely associated with the Kafka cluster. You have a ZooKeeper environment which provides the brokers within a cluster, the metadata it needs to operate at scale and reliably. As this metadata is constantly changing, connectivity and chatter between the cluster members and ZooKeeper is required. Of course, the cluster doesn't do much unless if you put it to work and that's where Kafka producers and consumer applications come in. Each of these components can scale out to take on more demand and increase levels of reliability and availability. I have given some incredible statistics about LinkedIn's Kafka environment, but Netflix is even more impressive. Over 4000 brokers across 36 clusters, processing over 700 billion messages per day. I hope with this foundation established, you will start to understand how. But we're not finished. We still have to dive deeper into the messaging internals, which happen inside the broker. And with that, learn more about Kafka topics and partitions. In this module, we covered a lot of ground. We started by discussing how Kafka is a publish subscribe messaging system with common concepts as other messaging systems. But we started to differentiate Kafka with slightly different terminology, especially around producers and consumers that publish and receive messages respectively, and independent Kafka brokers and how they are grouped to form a cluster. We also introduced some fundamental characteristics of distributed systems, and we started to describe the worker node roles of controllers, leaders, peers and followers. Also, we discussed how distributed systems enforce and provide flexibility and reliability through replication policies. We rounded it off by highlighting the importance of consensus or gossip-oriented communication within a distributed system and the critical role that ZooKeeper plays in that, and in enabling Kafka and other distributed systems to function scalably and reliably.

Understanding Topics, Partitions, and Brokers

Introduction and Apache Kafka Setup Demo

At this point, I am hoping you now have a fundamental understanding of the architecture of Apache Kafka, at least from the standpoint of how Kafka organizes its brokers into clusters and distributes work redundantly. We will build upon this foundation and now discuss the central concepts of Kafka, message topics, partitions, and how brokers manage them in line with the distributed systems principles we just covered. Before we go much further, I want to show you how to get Apache Kafka installed on a development machine like a virtual machine running Linux. We will not go into a lot of configuration details at this point. Thankfully, we don't need to. Kafka is ready to go with a basic installation, and that's what we'll start with because it's the easiest. We'll download the binary package from the Kafka Apache site, extract the archive into a directory, and finally, we'll take a look inside that directory. In this demo and throughout the rest of the course, I'm going to assume a set of prerequisites. First, that you're somewhat familiar with the Linux operating system as that is the recommended operating system for running Apache Kafka. We'll be using the terminal and a Bash shell mostly for this. The Java 8 Development Kit needs to be installed and configured. It doesn't matter which JDK you use. It could be the OpenJDK or the Oracle JDK. It's just important that you have one installed and configured. Finally, you have to have Scala installed. Since Apache Kafka was mostly written in Scala, you'll need it's runtime. For this course, we're using the latest version, which is 2.11. Let's get started. Let's grab the binary package from one of the official Apache Kafka mirrors. We'll use wget for this, but you can also download it using a web browser. In this case, we're downloading the version of Kafka that corresponds to the version of Scala installed on the system. We can see that it's downloaded as an archive, so now we'll extract it using the tar command. Now I took the liberty to create a Kafka program files directory in usr/local/bin and copied the extracted archive contents there. This step isn't required as you can run Kafka from wherever you would like, even from the same location that you extracted it from the archive. But let's explore the Kafka installation directory contents. The site-docs folder just contains an archive containing all of the documentation that you'll find online. Let's go into the libs folder. This folder contains all of the dependencies Kafka has in order to run. You'll notice there at the bottom these archive for ZooKeeper and its client library. This enables Kafka to be a self-contained installation, not requiring ZooKeeper to be installed prior, which is convenient to get started quickly. Next is the config folder. This is an important folder because here you'll find all of the files you'll need to configure all of the components of Apache Kafka. Some of these files are out of scope for the current course, but the ones related to the broker, the producers, and consumers, we'll get into those in due time. This file, server.properties, is the configuration file for the Kafka broker itself. By taking a quick peek into it, you'll notice that it's self-describing and straightforward as far as knowing where to add custom settings. Finally, the bin folder. This folder contains all of the programs to get Kafka up and running in a variety of capacities. You'll notice the windows folder as well. It contains batch files that more or less do the same job as the shell scripts you see here. Like with the configuration files, many of these scripts will be outside of the scope of this course, and we'll stick to the most pertinent ones.

Apache Kafka Topics in Detail

All right, now that we have Kafka installed, let's get back to the course content. At the beginning of the last module, we briefly mentioned the concept of a messaging topic. It is the primary abstraction of Kafka because it is central to its entire purpose. Kafka topics are really just a named feed or category of messages. One way to think of it would be to consider a mailbox. It's an addressable, referenceable collection point for messages that producers send messages to and consumers retrieve messages from. In Kafka, a topic is a logical entity, something that virtually spans across the entire cluster of brokers. Producers and consumers alike don't really know or care about where and how the messages are kept. They just care about the topic to work with. However, behind the scenes, for each topic, the Kafka cluster is maintaining one or more physical log files. We'll go into that last statement soon, but before we do, let's get clear on what a topic is from a logical viewpoint. Topics can span an entire cluster of brokers for the benefit of scalability and fault tolerance. With the abstraction of a topic, a producer simply needs to publish messages to that topic. How it's maintained and managed over the multiple brokers is not its concern. Similarly, consumers simply want to consume from a topic, regardless of where it is. Let's zoom in a bit and look at what's happening within any given topic. When a producer sends a message to a Kafka topic, the messages are appended to a time-ordered sequential stream. Each message represents an event, or fact, that from the perspective of the producer is worthwhile to preserve and make available to potential consumers. These events are immutable. Once they are received into a topic, they cannot be changed. Consequently, if a producer happens to send a message that is incorrect or represent a fact that is no longer valid, its only recourse is to follow up that previous message with a newer, more correct one. It would be the job of the consumer to reconcile between the messages when it reads them and processes them. This style of maintaining data as events is an architectural style known as event sourcing and is becoming widely used as a means to manage independent caches of data in a reliable, flexible, and distributable manner. You'll recall I just said that a Kafka topic stores a time-ordered sequence of messages that share the same category. Let's look at a Kafka message from a logical standpoint and discuss what it contains. At a high level, a Kafka message has a timestamp that it's set when the message is received by a Kafka broker. Furthermore, a message received gets a unique identifier. The combination of the timestamp and its identifier form its placement in the sequence of messages received within a topic. These identifier itself is referenceable by the consumers in order for them to operate autonomously and efficiently, as you will shortly see. Of course, the message itself has a binary payload of data, which is what the producers and consumers really care about. We'll get into the binary data in detail in the next module when we talk about producers. From the consumer's perspective, they simply read messages from a topic. One of the truly remarkable things about Kafka and a primary tenant, if you recall, of its design goals was to make message consumption possible from a theoretically unlimited number of independent and autonomous consumers. Again, there may be several consumers that are all interested in receiving the same messages at the same or different times. Furthermore, if a consumer happens to erroneously process some messages, that fault should not have any impact on any other consumer or the producers, for that matter. Each should maintain its own exclusive operational boundary from one another. Even a complete crash should not keep others from operating. So, how do consumers do this? We'll get into the guts of this in module five. But since our current discussion is still from a logical point of view, let me touch on that.

The Consumer Offset and Message Retention Policy

So back to the consumers. How do they maintain their autonomy as far as message consumption from a common topic? It's called the message offset. The message offset is a critical concept to understand because it is how consumers can do read messages at their own pace and process them independently. Essentially, the offset is a placeholder. You can think of it like a bookmark that maintains the last read position. In the case of a Kafka topic, it is the last read message. The offset is entirely established and maintained by the consumer. Since the consumer is responsible for reading the messages and processing them on its own, it stands to reason it should also keep track of what it has read and has not read. The offset itself refers to a message identifier. In fact, it is the same identifier I described a few slides ago. Let's look at how this works. When a consumer wishes to read from a topic, it must establish a connection with a broker. Upon the connection, the consumer will decide what messages it wants to consume. This is entirely at the consumer's discretion. If the consumer has not previously read from the topic, or if it has, but wants to start over, it will issue a statement to read from the beginning of the topic. Behind the scenes, this is essentially the consumer establishing that its message offset for the topic is 0. And as it reads through the sequence of messages, it will inevitably come to the last message in the topic and move its offset accordingly. Of course, another consumer is likely interested in the message from the topic as well, but another consumer could be at a different place in the topic. It could've already read the messages from the beginning and simply is waiting for more messages to arrive so it can read and process them. The key here is that it knows where it left off and can choose to advance its position, stay put, or go back and reread another previously read message, all without the producer, brokers, or other consumers needing to know or care. When other messages arrive, the connected consumer will receive an event indicating there is a new message, and it can advance its position once it retrieves the new message. When the last message in the topic has been read and processed, the consumer can set its offset and at that point is caught up. Kafka is immune to one of the challenges of most messaging systems, slow consumers. We discussed this in the previous module. The reason why Kafka is immune is because it has the means to retain messages over a long period of time. The time it can retain messages is configurable and is known as the message retention policy. All published messages are retained by a Kafka cluster, regardless if a single consumer has consumed a message. The length of time in which messages are retained is configurable in hours. By default, the retention period is 168 hours, or 7 days. Beyond that, messages will start to fall off, starting with the oldest messages that have expired or fallen past the retention period to make room for the new messages. The retention period is set on a per-topic basis, which means that within a cluster you could have hundreds of topics, each with different retention periods. Besides the retention policy, the ability to retain messages is obviously going to correspond to the storage resources available.

Demo: Starting Apache Kafka and Producing and Consuming Messages

I think it's about time for a demo. In this one, we'll look at the basics of setting up a Kafka cluster with a single ZooKeeper server in a single Kafka broker and creating an Apache Kafka topic, sending some messages to it with a producer client and reading from that topic with some consumer clients. The demo will be very basic, as it's meant to illustrate the quickest way to get started and see Kafka actually working. As we go on, we'll get more detailed in terms of what's happening behind the scenes. I'm hoping you'll watch out for how we use the built-in Kafka producer and consumer client applications and the message order in which we'll be producing and consuming messages in a topic. There's a tendency to grasp onto every detail and want to understand the what and why for everything. For example, you may find yourself getting caught up on the command line parameters and options. Don't worry about that, because we're going to cover those things soon enough. The purpose for this is to give you a solid baseline of a working Kafka environment. The first thing we need to do in order to use Kafka is to start the main components of the cluster. Hopefully, you remember what those are, the ZooKeeper instance and at least one broker. Fortunately, Kafka makes this easy by giving us some shell programs. Remember the bin folder from the Kafka installation demo? That's where we're going. Notice the ZooKeeper shell programs. We'll use the zookeeper-server-start one. Notice the USAGE hint. It is expecting a configuration file to know how ZooKeeper should behave once started. You can examine and modify this file as needed, as it is found in the config folder with the other Kafka configuration files. We'll just use the installation defaults this time. When you run the shell program, you'll see a bunch of info messages, but this signals that we've successively started ZooKeeper, and now it is sitting there waiting for processes to connect to it. To test that a ZooKeeper environment is running as expected, we can connect to it via telnet and issue a ZooKeeper "four-letter command", such a stat. This gives us the status of the ZooKeeper server. You may notice we're running in standalone mode. That is to say there is only a single instance running for testing and development purposes. With ZooKeeper started, we can now start a single Kafka broker. The process is very similar. We simply use another shell program. Again, notice the USAGE hint. Like ZooKeeper it is expecting a configuration file to represent a specific broker instance. For this single broker demo, we'll use the defaults, but we'll come back to this soon. Once the Kafka servers start command is executed, you'll see a bunch of info messages whirling by the terminal again. Notice the last few. It will say Registered broker 0 at path. That's saying that the broker has registered itself with the ZooKeeper server and is available to do work, so let's give it some work to do. Now with the server started, we'll create a topic. We will use the handy shell programs to do this for us. The one we'll use is kafka-topics.sh. When we execute this, the USAGE hint here is quite a bit more involved. There are a lot of commands and actions that can be taken for this process of creating a topic. I encourage you to study these options later. We'll use a few of them as we go. Take a look at the command I just typed in to create a topic. Yes, that's quite a few things needed to successfully create a topic. But let's talk a little bit about why. aside from the obvious create and topic commands, you'll notice we needed to pass in the ZooKeeper server. This is because there could be multiple ZooKeeper instances, each managing their own independent clusters. By specifying the ZooKeeper server here, you're basically saying, I want this topic to be created for this specific ZooKeeper managed cluster. Remember, it is the ZooKeeper component that is responsible for assigning a broker to be responsible for the topic. Another important thing to call out is the flags regarding replication factor and partitions. We'll talk about partitions and replication factors in a lot of detail later. When the topic was created, some interesting things happened behind the scenes. First, ZooKeeper scanned its registry of brokers and made a decision to assign a broker as the leader for the topic, my_topic. Second, on the broker there is a logs directory, and in there a new directory was created called my_topic,0. Within this directory, there are two files, an index file and the log file. We'll get into this as we talk next about partitions. Another useful function of the kafka-topics shell program is an option that enables us to inquire about the topics that are available on the cluster. You do this with the kafka-topics shell program with the option of list, and then, of course, you have to pass the

ZooKeeper server. Now that we have a topic, let's produce and consume some basic messages. First thing is to instantiate a producer. By now, you can predict how we'll do this. That's right, a handy shell program called kafka-console-producer.sh, another case of a lot of usage hints. In the next module, we'll cover the majority of these, but for now, we'll just go with the minimum to produce some basic messages. Once you've executed this command, you can keep the terminal window open most of the time, or as long as you like, and type whatever you want here. Everything you type and follow with Enter, you will cause the producer to send the message to the broker, which will then commit it to its log, waiting for a consumer to consume them. So let's get the other side working. Like the producer shell program, there is a consumer shell program, kafka-console-consumer.sh. This is how you execute it. Immediately, you'll see it pulling the messages from the broker and displaying them on the terminal window. Let's keep the consumer terminal window open and move over to the producer terminal window and write some more messages. Lastly, let's take a look now at the messages in the log. Warning, they won't make a lot of sense because most of the content is binary, but you'll recognize the text portion to prove that the messages were actually received and persisted in the log.

Apache Kafka as a Distributed Commit Log

Up to this point, we've discussed, at a logical level, the critical concept of a messaging topic in Apache Kafka. Throughout this description, you may have drawn parallels with other types of data processing systems, for sure other messaging systems that are also centered around the concept of a messaging topic. But given what we have described as to how messages are represented as immutable events that are appended and persisted in a time-ordered sequence, does this sound familiar to you? Think of the internals of other data processing systems that you've learned about or even worked with in your career. Can you see some analogies? The reason we have spent our initial focus in this module on the logical abstract concept of the topic is to impress upon you this simple yet powerful basis upon which Apache Kafka's architecture was built, the commit log. If you think about the internals of the database, you'll see the corollary I am suggesting. A database transaction or commit log is the primary record of what happens. It continually appends events in the order in which they are received. Log entries, new or otherwise, are maintained in the physical log file maintained by the database. From there, any number of derivative data structures can be formed to represent the contents of the log. In relational databases, for example, these structures can be things like indexes, tables, and views. They are just projections from that authoritative source of truth we're calling a log. This concept is useful for many reasons. For example, when a database crash occurs, it is the commit or transaction log that serves as the source from which restoration is possible. Because when you read from the log, you're essentially able to replay all of the things that happened in the order in which they happened. Additionally, it is also the basis for which replication and distribution can occur for redundancy, fault tolerance, and even scalability. If you go to Apache Kafka's project site at kafka.apache.org, which I encourage everybody to do by the way, you'll notice the very first statement made to describe what Apache Kafka is and how it works. This is that statement. I emphasize the last three words based on the discussion we have had up to this point. Yes, Apache Kafka is a messaging system. We've already discussed this. Yes, it uses publish and subscribe semantics as the basis for its data movement. But what it really is, at its heart, is a distributed commit log. In a nutshell, if you were to strip off all of the higher abstraction layers from a database and be left with its commit log, and if you were to leverage the principles we discussed in the last module as the basis for distributing the contents of that log, you would have Kafka. In a way, Kafka is really just a highly distributed raw database that brokers reads and writes using publish and subscribe semantics. Simple and really cool.

Apache Kafka Partitions in Detail

Now we're ready to cover the next most important concept in Apache Kafka, partitions. This is where we start to cover how the logical concept of a topic is implemented in the physical world and how, within this physical realm, Kafka is able to operate and deliver upon its amazing promise. The topic, as a logical concept, is represented by one or more physical log files called partitions. The number of partitions in a topic depends on the circumstances in which Apache Kafka is intended to be used. Nonetheless, the number of partitions per topic is entirely configurable. The partition itself is central to how Apache Kafka achieves its amazing nonfunctional capabilities, such as its ability to scale out, providing for greater levels of fault tolerance, and achieving higher levels of overall throughput. As such, each partition is maintained on at least one or more brokers. Let's take a closer look. As you will recall from the demo, we created a topic called my_topic, and as part of that process, we were required to indicate the number of partitions it would have, as well as the topic's replication factor. For the simple case of the demo, we chose a single partition. Obviously, we could've set any reasonable number we wanted, but I chose a single one to illustrate the basics of how partitions work and why they are a critical element of the Kafka architecture. At a minimum, each topic has to have a single partition because that partition, as I mentioned, is the physical representation of the topic as a commit log stored on one or more brokers. Referencing the demo, that log was maintained on the broker's file system in the directory tmp/kafka-logs. For that one topic we created, there was a subfolder called my_topic-0, which contained the log for that single partition. There are some legitimate reasons for choosing a single partition topic, even in production. Those reasons we'll cover and module five. But like any design choice, there are tradeoffs. In this case, the tradeoffs can limit scalability and throughput. Why? As you know, a physical node, upon which the broker and the partition log resides, is limited by a finite amount of computational resources, such as CPU, memory, disk space, and network. While it is possible to keep adding more faster and bigger resources, eventually, under the strain of high use, your only real option is to scale out, especially considering each partition you have must fit on one machine. You cannot split a single partition across multiple machines. Therefore, if you only have one partition for a large and growing topic, you will be limited by the one broker node's ability to capture and retain messages being published to that topic, not to mention the possible I/O constraints it will run into. In Kafka, that means you'll need more brokers in the cluster and topics that can leverage those brokers by partitioning into multiple partitions. It is the biggest way to scale Apache Kafka. So let's suppose we have created the same topic, but with three partitions instead of one. How is this different? In that scenario, with three partitions, we're causing a single topic to be split across three different log files, ideally managed by three different broker nodes. Each partition is mutually exclusive from one another in that they receive unique messages from a Kafka producer producing on the same topic. This enables each partition to share the burden of the message load across multiple different broker nodes and increase the parallelism of certain operations like message consumption. Despite each partition getting different messages, the manner in which they are managed is the same. It's a time-ordered sequence of events that are appended to buy the Kafka producer. It's the same. I hope you're wondering how the Kafka producer splits the messages across the different partitions. That is a good question and one we will discuss in the next module. But to tease the answer a little bit, it is based on a partitioning scheme that can be established by the producer. In this illustration, a specific partitioning scheme is not used, so the producer is just doing it round robin. It's a very important topic because it has implications on how balanced the partitions are for any given topic.

Distributed Partition Management in Apache Kafka

Let's start to bring this together into a consolidated big picture. From a work distribution standpoint, remember in the last module we talked about leaders being elected to take ownership of work. This is how partitions get distributed to the available brokers in the cluster. For example, when a command to create a topic with three partitions has issued, it is handled by ZooKeeper, who is maintaining metadata regarding the cluster. At this stage, ZooKeeper is specifically going to look at the available brokers and decide which brokers will be made the responsible leaders for managing a single partition within a topic. When that assignment is made, each unique Kafka broker will create a log for the newly assigned partition. This log will be precisely what we saw in the demo, and it's hosting directory will be named by the topic and the partition in which it represents. Additionally, as partition assignments are broadcast, each individual broker maintains a subset of the metadata that ZooKeeper does, particularly the mapping of what partitions are being managed by what brokers in the cluster. This enables any individual broker to direct a producer client to the appropriate broker for producing messages to a specific partition. Along the way, status is being sent by each broker to ZooKeeper so that proper consensus can be maintained in the cluster. When a producer is ready to publish messages to a topic, it must have knowledge of at least one broker in the cluster so it can find the leaders of the topic's partitions. Each broker knows which partitions are owned by which leader. The metadata related to the topic is sent back to the producer so it can begin to send messages to the individual brokers participating in managing the topic, or I should say, the partitions in that topic. On the surface, consumers operate much the same way a producer does, except it leverages ZooKeeper more for some very important ways that I'll have to defer the explanation of which until module five. But one way or another, when consuming messages from the cluster, the consumer inquires of ZooKeeper which brokers own which partitions and gets additional metadata that affects the consumer's consumption behavior, particularly in scenarios where there are large groups of consumers sharing the consumption workload. Once the consumer knows the brokers with the partitions that make up the topic, it will pull the messages from the brokers based on the message offset per partition. Because messages are produced to multiple partitions and at potentially different times, consumers working with multiple partitions are likely going to consume messages in different orders and will therefore be responsible for handling the order if it is required. We've established that having multiple partitions is a must in order to effectively scale out a Kafka cluster's ability to handle large volumes of messages and increase the degree of parallelism in which consumers can consume those messages. However, there are drawbacks to having too many partitions. Like any architectural decision, the number of Kafka partitions will have its tradeoffs, and therefore, a right balance needs to be found based on use case needs and resource availability. As you've seen, ZooKeeper plays a big role in making sure all of the brokers in the cluster are working in concert. The more partitions there are, the more entry ZooKeeper has to make to keep track of them. And since ZooKeeper works on this registry in memory, the resources on ZooKeeper can become constrained. Now this can be mitigated by ensuring your ZooKeeper ensemble is properly provisioned, commensurate with the growth of topics and their partitions. Every message in each partition is totally ordered, as we discussed earlier, in the sequence in which it is received. However, as I mentioned, a topic may consist of many partitions, so there will not be a global order to messages in a topic across all partitions. This can be complex if your consuming application needs to have a global messaging order in the topic across all partitions. To get a global order without the consumers having to manage the ordering process, you may need to consider a single partition for a topic. The tradeoff there is obvious. You'll be limited in terms of scalability beyond the single broker managing that single partition. Alternatively, you can intelligently use consumers and consumer groups to consume messages from the topic partitions and handle the ordering process there. We'll cover these scenarios more in module five. Lastly, when you have a higher number of partitions, the process of a leader falling over to another can start to get time consuming. Now the failover process is handled very fast, in the low milliseconds, but in larger clusters with a large number of partitions, this can start to add up, which is why in big, big implementations you'll see multiple clusters on their own. These are a few major considerations, but not the only ones. We'll discuss more of these as we talk about producers in module four and consumers in Module five.

Achieving Reliability with Apache Kafka Replication

So we've covered partitions fairly well. We know what they are, why they are. But you remember this slide. Based on what we've discussed thus far, what are we missing that we should cover in this module? Yes, we need to discuss how partitions enable work not only to be distributed, but reliably distributed, as per the last module's distributed computing principles. There's a lot of different types of faults that can happen in a distributed system, exponentially more than in a non-distributed system. For example, what if a broker fails and becomes unresponsive? What if there is a network issue that makes a broker unreachable? Since the data is stored by the broker, what happens if the disk fails or the data is otherwise inaccessible? Let's suppose we have a multi-broker cluster, and each broker is the leader for one or more partitions. What would happen if one of those brokers were to fail or become unresponsive? This could be a potential catastrophe, because it could mean the loss of data. Due to the responsibility of ZooKeeper, when it determines that a broker is down, it will find another broker to take its place, and the metadata used for work distribution for either producers or consumers will get updated, and the system will go on. But without some redundancy between brokers, there could be unrecoverable data loss because the data being managed by the failed broker, unless replicated to another cluster, is now inaccessible. Once ZooKeeper and the brokers have handled the reassignment and the producers and consumers in the system are updated, they can continue to publish and consume messages to and from the topic partitions, but the previous messages that were lost in the partition that failed will still be inaccessible. This is why the designers of Kafka created a facility enabled through a configuration property to ensure redundancy of data, so in the inevitable event of an individual broker failure or fault, there isn't any data loss. This configuration property is one we've already seen. It's called the replication factor. You'll recall in the demo we've miserably set that to 1, which means that the topic's partitions will only have a single replica at any given time. We can excuse ourselves though because it's just a demo running on a virtual machine, but it's an important configuration property not to be for gotten, so let's discuss it further. Obviously, it's a good idea to leverage the replication facility designed in Kafka for the reasons we discussed. It's a critical safeguard that enables reliable work distribution, which for a distributed system is a must, as failure of some sort is virtually guaranteed. This ensures that messages are stored redundantly across more than one broker, which makes the overall system more resilient and fault tolerant when a broker failure happens, all for the purposes of mitigating the risk of data loss. By setting the replication factor to N, you virtually guarantee yourself up to N-1 server or broker failures. Therefore, it is generally recommended that you set the replication factor accordingly. In fact, a minimum of two to three so that failures or machine maintenance will not interrupt the cluster's operation. Finally, it is important to note that the replication factor can vary from topic to topic because it is set at the topic level. Let's walk through how Kafka operates when the replication factor is set. This should look familiar. Here we're using the built-in shell program for creating a topic, as I showed a few slides ago. In this particular scenario, let's simulate the same thing, but this time set the replication-factor to 3. This is the same create topic scenario as before. The only difference is we'll walk through what happens when we set the replication-factor to 3 instead of 1. With a replication-factor of 3 set, it is the leader's job to get peer brokers to participate in a quorum for the purposes of replicating the log to achieve the intended redundancy level. When the leader of a partition has a quorum, it will engage its peers and start copying the partition log. When all members of the replication quorum are caught up and a full synchronized replica set is in place, it is reported throughout the cluster that the number of in-sync replicas, or ISRs, is equal to the replication factor for that topic and each partition within it. Obviously, this is an important metric. When the ISR is equal to the replication factor, the topic and each partition within that topic is considered to be in a healthy state. If for any reason a quorum cannot be established and/or the number of ISRs fall below the configured replication factor for the topic, intervention may be required. There could be legitimate planned and unplanned reasons a broker is not able to be replicated to. And because of that, Kafka doesn't automatically go out and search for a new following peer to replace the quorum member. Despite how resilient Kafka is, vigilant monitoring and compensating actions are needed to eventually replace or tune a lagging or missing-in-action member of the quorum. In the last demo, you may have noticed I used this useful command in the kafka-topics shell program. It was a command for describe. This useful command enables you to see what's going on within a topic, including all of the partitions, the leaders it's assigned to, the replicas, and the ISRs.

Demo: Fault-tolerance and Resiliency in Apache Kafka

Let's go through one last demonstration that will cover as much as we have up to this point, which is a lot. In this demo, we will set up a Kafka cluster with multiple brokers. We'll go with an odd number of three. And for simplicity, we'll configure it with a topic with a single partition that has a replication factor of 3. We'll go with this configuration to illustrate what details are provided when using the describe command, and then we'll simulate a failure scenario and witness how Kafka is resilient and enables operation to continue transparently to both publishers and consumers. Setting up a multi-broker cluster on a single machine is relatively simple. Remember those server. property files that we showed at the beginning of the module? Well, you just basically need to create a separate server.properties configuration file for each broker you want to instantiate. So here I already have ZooKeeper running, so let's just start up the three Kafka brokers in their own terminal windows. You'll recall that we can do that by executing the shell program for kafka-server-start and passing the configuration properties. And each one, of course, as I just said, is going to have their own properties file, one per broker. Now let's create a topic with a replication-factor of 3 and a single partition. We'll call this topic replicated_topic. Let's run that. Yep, and as you can see, we successfully created the topic, replicated_topic. Now we can check some of the details about the topic we just created by using the handy-dandy describe command. Let's run that and take a look. All right. Okay, let's expand this a little bit more so we can see everything on one line. So it shows us that we have only one partition with a replication factor of 3, just like we asked. Further in the details, it lists the information by partition. If this were a topic with more than one partition, there would be multiple lines here, one for each partition. For this single partition, partition 0, it shows the leader host, in this case, node 1. Additionally, it indicates the replicas in place for this partition. We see 1, 0, and 2, meaning there is a replica on node 1, obviously because its leader, node 0, and node 2. Furthermore, we see the in-sync replicas in the same order, 1, 0, and 2. Since the number of ISRs is equal to the replicas, we can safely say the partition and the quorum managing it are in a healthy state. Now let's create a simple producer and produce some messages to the new topic. Okay, so here's a producer terminal window. You'll notice it's the same as before, but now you understand a little bit more about brokers and how they work together, and so you'll notice that we're passing a broker-list parameter. I'll explain why this is needed in the next module. So now let's add a few messages, and now let's quickly create a consumer to retrieve the messages. Okay, here's a terminal window for the consumer, and you'll notice there's not really anything new here. The messages were retrieved, and now the consumer is waiting for more. At this point, we're going to simulate a broker fault, and after we'll use the describe command to see how Kafka handled the broker going down as far as the topic management goes. Here I have the terminal window that I used to launch the first broker, or broker 1. So why 1? Well, because in the topic details, it says the leader for the topic, and its single partition is node 1. So the idea here is to kill it and see what happens. All right, let's kill this broker. Okay, so that did it. Now, you can see here in the INFO message that the node 1 shut down. All right, so now let's go to the window where we had the details up and pull up the details again. Okay, so here you can see the same topic with the same partition count, same replication factor, but now the leader has changed. The leader is now node 0. Whereas if we look at the previous details, the leader was node 1. Now it's again node 0. So if we look at the replicas at this point, we still see that there's three replicas. There's 1, 0, and 2. But if you look at the in-sync replicas, there's only two in-sync replicas. One is gone because we killed broker 1. It is no longer available. So, as an administrator, if you look at this and you see that there's two in-sync replicas and there's a replication factor of 3, what does that tell you? Well, that tells you that your quorum is unhealthy, that there is a missing replica, and it needs to be replaced. Now, if there were another broker available, Kafka would've already added it to the quorum and started replicating it to take the place of the lost broker 1. But if you recall, we only had three brokers to start with, we killed one, and that was all there is. But now let's go back to the producer and the consumer and see how they handled things. So here's the producer terminal. It's just sitting there humming along. It's as if it doesn't even know anything happened. Well, behind the scenes it does, but it's still ready to publish messages. Whereas when we killed broker 1, something happened in the consumer. If we scroll through the messages printed to the terminal, you'll see a lot of exceptions and stack trace information with warnings indicating that it attempted to pull or fetch records from the broker, and it was unsuccessful. But notice this didn't cause the consumer to fail. It's still there waiting for new messages to arrive. In module five, we're going to talk more about the consumer and what causes these kinds of errors to happen when there is a change in the brokers that are providing the messages. So to prove this, let's go to the producer terminal window and produce another message. We'll call it My Message 4. So in the consumer terminal window, you'll see that it was able to retrieve the message just fine. Just because a broker falls out of the quorum doesn't mean that any data was lost, and that's thanks to the replication factor. Sure, the cluster could use another broker to be a complete, healthy quorum, but things are still working. Okay, I hadn't planned on this, but just for fun, let's get broker 1 back into the ring. Let's start him back up. Now that he started, he's joined himself back into the cluster. Now, if we go back up here and do a describe on the topics, we're going to get some information about the new membership. So you'll see that it's the same topic, and the leader is still 0, as it was before, because that didn't change. Nothing changed about node 0. The replicas are the same. We still have 1, 0, and 2. But the difference now is is that our in-sync replicas have gone from two, when we killed the first broker, now it's back to three, and it 0, 2, and 1. So I hope that can illustrate how resilient a Kafka cluster can be. Not only the cluster itself, but the producers and the consumers. I hope this module's explanation of the brokers, the partitions, topics, and everything has given you an understanding as to why it can be that resilient at the broker side. The next two modules on producers and consumers will talk about what happens internally to enable them to be just as resilient.

Module 3 Summary

Wow! We covered a lot in this module. We needed to because there are a lot of moving parts in Kafka, and it's important to understand more about it before we start exploring what it means to build producing and consuming applications with it. In this module, we spent a bit of time discussing topics as logical concepts and partitions as the physical commit log that stores the topic's messages. We dived deeper into the role of brokers in a Kafka cluster, particularly as it pertains to partition management and behavior. As we went through these things, I tried my best to continually map it to the previous module where we discussed distributed systems and how Kafka embodies distributed systems principles. For example, we covered how brokers become leaders to own and manage partitions. We also looked at work distribution and failover. We saw Kafka in action. We did three demos, and I hope they were illustrative enough of the concepts. They were intended to motivate you to explore more on your own and to continue to go deeper and further into this course. There are a lot of things to Kafka, but I think we covered a good amount of detail that is relevant to understanding how it all works. Of course, we'll spend more focused time on this in the few modules remaining. All in all, I think we laid down a solid foundation upon which we can build upon, and I look forward to discussing more about producers and consumers in the next few modules.

Producing Messages with Kafka Producers

Introduction and Setting up an Apache Kafka Development Environment

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. 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. 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. 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. 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.

Basics of Creating an Apache Kafka Producer

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. 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. 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. 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.

Creating and Preparing Apache Kafka Producer Records

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. 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. 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. 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. 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.

Apache Kafka Producer Record Properties

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. 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. 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. 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.

The Process of Sending Messages, Part One

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. 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. 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.

The Process of Sending Messages, Part Two

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. 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. 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. 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.

Message Buffering and Micro-batching

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. 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.

Message Delivery and Ordering Guarantees

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. 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.

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

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.

Advanced Topics and Module 4 Summary

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. 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.

Consuming Messages with Kafka Consumers and Consumer Groups

Introduction and Apache Kafka Consumer Overview

We've arrived at the third and final major component of Apache Kafka, the consumer. As consumers share a lot in common with their producer sibling, we will cover consumers with a very similar approach. We will take a look at the internals and use that as a basis for understanding how consumers work and behave, while throughout, we'll look at samples and demos to help you understand how to create your own Kafka consumers. Finally, towards the end, we will cover consumer groups and how they enable Apache Kafka to scale on the message consumption side. Up to this point, when we've discussed the consumer, it has mostly been from an external superficial perspective related to its role in relationship to other Kafka components. In this module, we're going to look deep into the consumer internals. Similar to how we started with the producer, we'll use this diagram as a map to help illustrate and guide for the remainder of the course. Don't try to understand everything you see here now, because we'll cover each component in detail in its due time. This is almost exactly the same as the producer, because the function these initial required property serve is nearly identical. Remember, in the consumer's world, you are reading messages and therefore need to specify a deserializer class for both the key and value. This of course, must correspond to the type specification of the producer message that was serialized with the same serializer classes. As with the producer, there are a lot of configuration settings available. I encourage you to use the link above and familiarize yourself with the various consumer configuration settings and options. Of course, we'll be covering the most relevant ones in this module. When it comes to creating a consumer, you'll once again notice the similarities with creating a producer. This is by no coincidence, as the designers of the Kafka client APIs wanted to make working with Kafka familiar and productive. Here, we are using a standard main entry point because we'll be using a console-based Java application in our consumer samples. Of course, the difference here is that you're creating an instance of a Kafka consumer, but passing the required configuration is needed. With the Kafka consumer object in place, we now need to give it something to subscribe to so it can start working.

Subscribing and Unsubscribing to Topics

Subscribing to topics programmatically is extremely easy and somewhat straightforward. To subscribe to a topic, you call a specific method. Want to take a guess what it's named? That's right. The subscribe method. The method signature for subscribe takes in a collection of strings, which represent a list of topics. A single consumer can subscribe to any number of topics from one to theoretically infinity. Seriously, I don't know if there is a limit to the number of topics that can be subscribed to. In this case, we're just using a list of one. Alternatively, you could subscribe by passing a regular expression as the parameter, which is a useful overload for the subscribe method. There is a noteworthy nuance to the subscribe method, and it is evident when you want to add another topic to the subscription list. Let's look at this code example. Initially, we subscribe to my-topic by passing the subscribe method, a single item in a list of strings. Now we want to add another new topic using the same technique. By doing it this way, you would think that it would work and that you would now have two topics to subscribe to, right? Wrong! Calls to subscribe are not incremental, meaning that any subsequent call to subscribe will overwrite whatever it had in there before. Therefore, the best approach would be to maintain the topics of interest into a separate structure, manage them there, and pass in the reference to the topic list like this. While we're here, we may as well cover the opposite of subscribe, and that is unsubscribe. It's as basic as it gets. Notice, there isn't a parameter list here. This suggests that you don't really unsubscribe from individual topics. You're basically unsubscribing from all topics. Here's another option for unsubscribing. You just pass an empty list to the subscribe method.

Comparing Subscribe and Assign APIs

There are some important points about Kafka consumers that I want to teach you at this point. We just covered the basics of creating a list of topics we want a single consumer to subscribe to. It's as simple is calling the subscribe() method and passing it a list. By calling this method, you are asking for automatic or dynamic partition assignment. That is to say that you're enlisting the single consumer instance to eventually pull from every partition within that topic, which can be at least one, but likely many. When adding multiple topics to the list, you're enlisting the consumer instance to pull from every partition within every topic, which is guaranteed to be many. This has very important implications, which shouldn't be taken lightly for reasons we'll cover shortly. Besides subscribing to topics, there's another option, subscribing to individual partitions. This is done through the assign() method. The assign() method is only valid for subscribing to a list containing the class topic partition, as we'll see next. But first, let me explain a key difference between subscribe() and assign() methods. There's a reason why the API designers decided to call this operation out as a separate method as opposed to overloading the subscribe method. By asking for specific partitions, you're basically taking on the responsibility of assigning yourself specific partitions. More specifically, assigning specific partitions to a single consumer instance. Once you've assigned yourself a list of partitions, the consumer will then start pulling these individual partitions, regardless of the topic those partitions are part of. Both have one thing in common, they take lists, and they cannot be added to incrementally, as we covered earlier. As we'll cover later with consumer groups, this assignment responsibility is generally managed for you and for a good reason. I suppose you can say using the assign() method is an advanced case and therefore needs to be treated with respect for reasons we'll discuss later. To manually assign partitions to a Kafka consumer, you will first need to create a list containing your manual partition assignments. This is done by instantiating topic partition classes with the appropriate information. You'll see the topic partition class referenced in a lot of places in the Kafka API. It simply provides a type safe data structure to represent individual partitions within a topic. Second, you're going to invoke the assign() method, passing at the list of topic partitions just created. That's about it. I'm sure you'll notice the general similarities with the subscribe() method.

Single Consumer Subscriptions and Assignments

This is a good time to teach you about how individual Kafka consumer instances interact with their subscribe topics. When a single consumer subscribes to a topic using the subscribe method, it will constantly pull any and all partitions within the topic for new messages to consume. This is the case for all of the topics for which the consumer is subscribed. Depending on the number of topics and the number of partitions within each of those topics, that could be a lot of message polling by a single consumer. We will discuss the challenges of this approach soon, but I wanted to teach you how this works first. The benefit to using the subscribe method to retrieve data is that partition management is entirely managed for you. For example, suppose there is a new partition added to an existing topic, presumably because the administrators wanted to increase the scalability of the topic. When that happens, the metadata about the cluster will have changed, and it will be sent to the consumer. Since the consumer maintains an internal object that manages its subscriptions, called SubscriptionState, it will know if the change has affected its subscriptions. In this case, it has, so it will know to automatically add the new partition to the topic list, which the consumer will start polling for messages. We'll cover how this happens towards the end of this module. Pretty convenient, isn't it? Just remember, this capability is only available through the subscribe method. In slight contrast, a single consumer instance may want complete control over what partitions it wants to poll messages from. There are legitimate reasons this may be called for, but they are advanced use cases that we won't have a lot of time to cover in this course. By specifying a list of topic partition objects, the consumer is assigning itself to specific partitions. This is a lot like hard coding a list of specific partition IDs in a watch list. At this point, the fact that a partition participates as part of a topic is less relevant, because as far as the consumer is concerned, it doesn't know or really care. It knows what topic each partition is in, but it doesn't really do anything with that information once it has assigned itself a partition. So if a partition is added to a topic, the consumer instance may be notified of it, as per the protocol of retrieving metadata from the cluster, but it doesn't really care, and why would it? If it had good reason to assign itself specific partitions, why would it care what happens in the topic? If it was interested in what happens at the topic level, it would have used the subscribe method instead, right?

The Poll Loop

We've talked about subscribing to topics and assigning partitions, which is really important to understand how the Kafka consumer works. You may have noticed that several times in that discourse, I used the term poll or polling. This was intentional because now it is the time to understand what polling means within the consumer context. It would be natural to think that by invoking the subscribe or assign methods that we just talked about, you would be actually kicking off the consumer to start receiving messages. That's not how it works. Nothing happens until you start the most critical process in the entire consumer component, which is the poll loop. The poll loop is the heart and soul of the Kafka consumer, and it is what enables the consumer to realize its purpose, and that is to continuously and reliably poll the brokers in the cluster for messages. It's a single and simple method, but don't let that fool you. From that simple method, all of the complex interactions between the consumer and the broker are kicked off and coordinated. We'll get to the details of this soon, but you will see how it goes way beyond just receiving messages. Let's take a look how to start the polling process. First of all, the pole loop can't be a loop without a loop to run it in. As funny as that sounds, it's true. A Kafka consumer is a long-running application, or at least it should be, whose job is to always be looking for new messages and process them from the Kafka cluster. There should be very few reasons you would stop polling once you've started. So think of the loop as an infinite loop that we will only be interrupting for valid reasons, whether intentional or unintentional. You start the loop by calling the poll method on the Kafka consumer object, passing it a long typed number representing a very important value that we'll cover shortly. You'll notice the direct output of the poll method is to return an object of type ConsumerRecords, which contains any records the consumer was able to retrieve from the broker. From there, what you do with the ConsumerRecords is entirely up to you, and this is where the diversity of Kafka consumer applications come in. Otherwise, they're fairly generic as far as what it takes to get it polling for messages. You'll notice that I enclosed the call to poll within a try block. Since the Kafka consumer is idle until the poll method is invoked, there really isn't anything that can throw a runtime exception to be handled. But because the invocation of the poll methods starts any and all network activity with the cluster, it is a good idea to enclose it with the means to do structured exception handling. As the poll operation opens network resources, it is always a good idea to make sure it closes in the end.

Demo: Simple Kafka Consumer

Let's go through a demo of a custom consumer application written in Java. The development environment will be the same as before. The only difference is that we'll have a consumer app class to run the consumer console application. The demo cluster configuration will consist of a single broker with two topics, each with three partitions. We'll only do a single replication factor for this demo. Look for the use of the handy kafka-producer-perf-test shell program to generate messages. Even though we're not planning on blasting our Kafka environment for performance tests, I want to use this as an opportunity to show you how to use this tool. We'll demonstrate two different consumers, one using the subscribe method for retrieving messages and the other using assign. We'll observe the output from each consumer and note the differences, and then we'll add a new partition to a topic. With this configuration, we'll look at the output of both consumers. You'll notice the differences between the assign consumer and subscribe consumer. I've already started a single Kafka broker and created two topics with three partitions each, and you can see this with the results of issuing a describe command against the cluster. Here's the topic, my-other-topic, again, PartitionCount of 3, single ReplicationFactor, and there they are with their leaders. And then we have the second topic, my-topic, with the same configuration. Before we do anything, let's take a look at our sample KafkaConsumer Java applications that we've got so far. So let's open up our IDE here. We have two applications actually. We have one for subscribing to topics and the other for getting specific partitions assigned to it. As we go through this, you'll notice that our code is aligned with what we've covered in the slides up to this point. For example, here we've established the required properties for the consumer and passed them into an instance of the KafkaConsumer class, thus creating myConsumer object. And here, for the subscribe consumer, we need to create a list of topics that we're interested in watching. These topics are simply my-topic and my-other-topic, as you saw in the terminal. And to add these topics to the consumer, we simply invoke the subscribe method here, passing in the list of topics. Next is where the action happens, starting responsibly with the try block and setting up a loop for which we can enter into the consumer poll loop. Here you'll see that the poll method has been set with a timeout value of 10 ms. Now, we'll get back to what this parameter means shortly. And within this loop, we're going to be taking each record that we get from the poll method, and we're going to be iterating over it and processing it minimally. In this case, we're just taking the values that are present and formatting them and outputting them to the console. And finally, you'll see that we literally have a finally block, so that when it exits we can responsibly close the consumer and free up the resources we need to. The assign consumer is virtually identical. The only difference is how we construct a list of specific topics that we want to assign to our particular consumer, and here you'll see those. We have to create specific TopicPartition objects, one representing each of the specific partitions within a specific topic that we want to assign. Here we are going to have to, we're going to have the first partition, Partition 0 from the my-topic partition, excuse me, my-topic topic, and we will have another representing the second partition, or I should say the third partition from my-other-topic. And we will be adding them to our list of partitions, and instead of calling the subscribe method, obviously we're invoking the assign method and passing in that list of partitions. Everything else, as far as how we're going to be processing the records that are retrieved from those partitions, is identical. Let's go ahead and run these two consumer programs so that they can be listening for messages to arrive in their respective subscriptions or assignments, and then after that, we'll start producing some messages. So let's start with the SubscribeApp. All we have to do is go in here and hit Run, and we'll do the same thing for the ConsumerApp. Jumping back out to a terminal, I wanted to show you a handy tool for creating lots of messages. It's called the kafka-producer-perf-test shell program, and this is roughly how you get it to work. So, as it says, this tool is used to verify the producer performance, so you can really just pump a bunch of messages in here. In order to get it to work, you have to pass in a topic. You give it the number of records that you want it to produce, the size of those records, and you can also determine what the throughput you want it to be. So in this case, you would be setting a value to represent how many messages per second. And finally, you would be passing to it a list of properties. Now, at the bare minimum, it would be the minimum required properties, such as bootstrap.servers, and then the, in this case, since we're producing, it would be the key.serializer and the value.serializer classes. So if I have this other window up here, it's a little busy, so you can see that I'm ready to start sending some messages to here to my-other-topic. I've said that I want to send 50 records with just 1 byte each, so very small, with a throughput of 10 per second. You can't see that, it's a little cut off, but it's going to be 10 per second, so it'll take a total of 5 seconds to send all of these messages. And, of course, I passed in the required properties, the bootstrap.servers and the string key-value serializers. I'm also doing this for the other topic, so both topics will have the same configuration. Alright, let's run our producers. Let's start off with the producer, the test producer for my-topic, and we'll run the test producer for my-other-topic. And here you're going to see a bunch of stuff getting output to the Subscriber app. And if we go over here, we'll see a lot of output to the Consumer app. Okay, so let's look at the results here, and then I'm going to, I want you to look for certain things. Okay, so when we ran the first performance tool, you'll notice that on the SubscribeApp, you'll remember now that it's subscribing to all partitions by topic. So, within this, you'll see for my-topic it has a mix of messages it's getting from all partitions. And then we ran the other perf tool that started publishing messages to the other topic, and just even with this, you'll start to see that it was looking at all of the partitions within that topic, as you would expect. Now the output is just based upon different producers formatting the results differently based upon what values were in the messages. Now, if we go over here to KafkaConsumer, where the assign method was used, we'll see some different results. For my-topic, all we see is Partition 0, because, if you remember, that's all we told this consumer to look for for that topic. In addition, if we see my-other-topic, which was the other topic whose partition number 2 we assigned to the consumer, that's all that it would notice, so it's doing exactly what we would expect. Now, we're going to keep this open, just as is, we're not going to stop it at all, and what I'm going to do is go over to another terminal window and I'm going to create another partition within one of those topics, and we're going to see what happens when we start producing data to those, to that new partition. Okay, here in this terminal window, you'll notice that I'm using the kafka-topics shell program, and we've seen this in action before. There's a little bit of a difference here. Yes, I'm passing in the zookeeper reference, but here I'm using the command to alter, because I'm basically saying I want to alter this topic, which is just the my-topic topic, and I'm saying I want it to have four partitions instead of three. So let's run this, and as you can see it says Adding partitions succeeded. Now let's go into here real quick and look at our describe just to make sure that it took. Basically, what you're going to see now, despite all of that whizzing by, we'll get to that later, is basically you'll see that for my-topic we now have a PartitionCount of 4, so 0 through 3. Okay, so with that, go back to our producer test tools, and let's produce a new round of messages and see what happens in our consumers. Okay, so if we go back here to our applications, we more less left off where we had them, and that is the SubscribeApp and the ConsumerApp, they're just waiting for new messages. Now, remember, we added a new partition to the topic my-topic, so in this window we are going to create more messages that go to my-topic. We don't need to add more messages to the other topic, since that was not changed, so let's do that here. So we're producing more messages to the KafkaConsumerSubscribeApp, and as you can see, it went through and here within the my-topic we have now an extra partitions-worth of messages that it's looking at. It has 0, 1, 2, and 3, and it successfully subscribed to all of those new partitions. It became aware that there was a new partition, it added it to its subscription, and then actively started listening for it without really any intervention on our part other than just creating the new topic. Now, let's compare that to the ConsumerAssignApp. Nothing happened. When it got more messages, it definitely received them, but it only got the messages for Partition 0. You'll notice there's nothing there from Partition 1, nothing there for Partition 2 or 3. All it knows or cares about is Partition 0; it has no knowledge that there is other partitions other than this. So that hopefully illustrates the differences a bit between the assign and the subscribe methods when polling for messages in a consumer.

Walkthrough: Consumer Polling

When the subscriber assign method is invoked, the content of the collections they were passed to are used to set fields within the SubscriptionState object. This object serves as the source of truth for any and all details related to the topics and partitions this consumer instance is subscribed or assigned to. A lot of what happens within the consumer invariably crosses paths with this object. This object also plays a very important role with the consumer coordinator in managing the offsets, a topic we covered briefly in module three and we'll spend a bit more time on it later in the module. When poll is invoked, consumer settings, particularly those referring to the bootstrap servers, is used to request the metadata about the cluster. This kicks off everything within the consumer. The fetcher serves as the responsible object for most of the communication between the consumer and the cluster. Within it, there are several fetch-related operations that are executed to initiate communication with the cluster, but the fetcher itself doesn't actually communicate with the cluster, that is the job of the consumer network client. With the client open and sending TCP packets, the consumers start sending heartbeats, which enable the cluster to know what consumers are still connected. Additionally, the initial request for metadata is sent and received. The response is used to instantiate its internal metadata object, which will keep up to date, while the poll method runs, getting periodic updates from the cluster when cluster details change. With metadata available, other major elements become more involved. With information about the cluster, the consumer coordinator can now take responsibility to coordinate between the consumer. This object has two main duties. First, being aware of automatic or dynamic partition reassignment and notification of assignment changes to the subscription state object, and second, for committing offsets to the cluster, the confirmation of which will cause the update of the subscription state so it can always be aware of the status of topics and partitions. To actually start retrieving messages, the fetcher needs to know what topics or individual partitions it should be asking for. It gets this information from the subscription state object and with it, starts requesting messages. Here is where I'll explain what that value that is being passed to the poll method means. It is a timeout setting, representing the number of milliseconds the network client is to spend pulling the cluster for messages before returning. This is an important setting because it establishes the minimum amount of time each message retrieval cycle will take. I'll cover this shortly. When the timeout expires, a batch of records are returned and added to an in-memory buffer where they are parsed, deserialized, and grouped into consumer records by topic and partition. Once the fetcher finishes this process, it returns the objects for processing.

Walkthrough: Message Processing

An important thing to understand about Kafka consumers is that they are essentially single-threaded. There is one poll loop per Kafka consumer, and you can only have a single thread per Kafka consumer. With all of the responsibilities that stem from the poll method, this may be surprising to you, if not downright troubling. The Kafka consumer was designed this way mainly to keep its operations simple and to force parallelism of message consumption in another, more scalable way that we'll see shortly. Again, knowing this is important for you and your approach to designing Kafka consumer applications because the reality of only a single thread available for record processing will have implications on how much you can reasonably expect a single Kafka consumer to do. Let's discuss this further. Let's continue our walkthrough of the consumer internals by discussing what happens after the poll method has returned messages for processing. Since the return type of the poll method is a collection of consumer records, we will need to iterate through them to process them individually. Now, what logic to apply to each individual record is entirely up to the developers working on the consumers. But careful consideration should be made to how each record should be processed. Remember, when calling the poll method, you can only do so much within a single thread. If you were to spend too much time in processing records, it could have big implications on the environment in which the consumer application process is running. Thankfully, because of Kafka's architecture, a slow consumer doesn't have an impact on the cluster, producers, or other consumers. Nonetheless, it's important to remember that any one consumer can subscribe to any number of topics and partitions. The more the consumer signs up for, the more it has to process and all within a single polling loop. Given the possible load that a Kafka cluster can be required to handle, having a single consumer may not be a feasible or rational idea. Let's get into some more details about the consumer so we can explore options for developing and configuring consumer applications at scale.

The Consumer OFfset in Detail

It's been a few modules since we discussed the all-important offset. If you recall, the offset is the critical value that enables consumers to operate independently by representing the last read position the consumer has read from a partition within a topic. When you think about the business of consuming messages, you realize just how important the offset is and, more importantly, whether it is accurate. How Kafka manages the consumer offset is one of the more important things to understand, and that's why we're going to spend a bit of time on it right now. First, there is some important terminology to learn about the offset. There are different categories of offsets, with each representing the various stage they are in. When an individual is reading from a partition, it obviously needs to establish what it has and hasn't read. This definitive answer is called the last committed offset, and it represents the last record that the consumer has confirmed to have processed. We'll get into this confirmation process shortly, but this is the starting point for a consumer within any given partition, depending on the configured offset/reset behavior, which we'll also cover later. You will notice we're really looking at it from a partition viewpoint, and that is because each partition is mutually exclusive with regard to consumer offsets. So for any given topic, a consumer may have multiple offsets it's tracking, one for each partition within a topic. As the consumer reads records from the last committed offset, it tracks its current position. As we illustrated in module 3, this position advances as the consumer advances in the log towards the last record in the partition, which is known as the log end offset. There is a notable difference, however, between the current position and the last committed offset, and it represents potentially uncommitted offsets. The success of robust and scalable message consumption in Apache Kafka largely depends on your understanding of what creates this gap and what can be done to narrow it. Every application has different processing requirements, functional and nonfunctional. It is the job of the application designer and developer to find the appropriate trade-offs that work. Next, I will walk through a scenario that illustrates this gap. There are two very important configuration properties that govern the default behavior of the consumer offset. These properties are optional because their defaults are sufficient for getting up and running. The first is enable.auto.commit, which is basically giving Kafka the responsibility to manage when current position offsets are upgraded to full committed offsets. This is a fairly blind setting because Kafka isn't going to know under what logical circumstances a record should be considered a committed record. The only thing it can do is establish an interval of time between commit actions that faithfully commit based on a frequency. That frequency is established by the auto commit interval property, and, by default, it is set to 5,000 milliseconds or 5 seconds. Now for high-throughput scenarios, 5 seconds is an eternity and likely sufficient. But let's consider the biggest variable here for a moment, and that is your processing logic. When a record is in processing scope, let's say it has a current offset position of 4 because the last successfully committed record was 3. Let's also suppose that for whatever reason, the processing of the current record takes longer than 5,000 milliseconds or whatever that interval is set to. Faithfully. Kafka is going to commit that record's offset regardless if it is finished processing or not because unless If you tell it explicitly when it's done, how is it supposed to know? Now this may be fine, but it's not entirely consistent. Generally, large-scale systems operate within eventually consistent boundaries. Now that's okay most of the time provided there is something else that's very important present, and that is reliability. Sorry, but I have to stand on my little soapbox for a minute. The gap between what is considered committed and what is actually committed isn't entirely bad. As I said, many large-scale distributed systems aren't 100% consistent. They are eventually consistent and Kafka. And consumers don't have to be an exception to that. But the extent in which you can tolerate eventual consistency is based on your application's functional requirements, of course, but also on the degree in which you can ensure reliability. If you can't provide reliability and robustness assurances, then an eventually consistent ideal becomes a never-consistent reality, which can be a disaster. So to continue with the offset gap illustration, suppose an are occurs that causes the message processing to fail for whatever reason. Now what? Depending on how far behind the consumer was when it failed, it may be very hard to know where you may need to go back to to start processing again because, according to Kafka, the records were committed. Knowing the current position at the time will be a start, but it could be messy to recover from. The impact varies largely based on your consumer topology. So far, we've only discussed a single consumer. The issues for a single consumer are different for a topology where multiple consumers exist within what is called a consumer group. I keep pushing this down, but we'll talk about this soon enough.

Offset Behavior and Management

So to recap and summarize for now on offset behavior. Remember, just because something is red, doesn't mean it's committed. A lot of things determine this and it is very subjective depending on the offset management mode you're operating in. The offset management mode is determined by the offset configuration properties. First and foremost is whether you want Kafka to manage your commits for you. The default is true because it is very convenient from a development standpoint, but as we saw, depending on the situation, it can be operationally inconvenient if there is an issue. It's a lot like garbage collection in modern programming languages, it's very convenient until it is inconvenient. The challenge is generally to have some sort of control to govern when it is tolerable to be inconvenient. Fortunately, in Kafka, you can adjust the commit frequency to be in line with your particular consumer application. This is the commit interval we discussed earlier. Lengthening this interval will provide an upper bound in which you can ensure your record processing will be finished, but it could also create an offset gap in the opposite direction where the commits are lagging behind your processing positions. As long as there is a gap, there is some risk exposure to failure and the possible inconsistent state you may be left with to clean up, not to mention the possible duplication of records when reprocessing. Another property we haven't covered yet, but will, is the strategy to use when a consumer starts reading from a new partition. The default is to start reading from the latest known committed offset. In contrast, this could also be set to the earliest. There is also a setting for none, which basically you're asking Kafka to throw an exception to the consumer and let you decide what to do with it. The offset behavior and the issues related to it vary depending on whether you're in a single consumer or a consumer group topology. All this time we've been talking about offsets, and I haven't taught you how and where they are stored in Kafka. The only thing I've said at this point is that consumers track the offset in terms of what it has or has not read, but where does it actually store them? Any guesses? Think about how Kafka stores data, period. If you guessed a topic, you would win a prize. Kafka stores the committed offsets in a special topic called __consumer_offsets. If you were to issue a describe command to the cluster asking it to show you all of the topics and their partitions, you would notice this consumer offsets topic and it would have 50 partitions. Yeah, that's a lot of partitions for a single topic. Now, why they chose the default of 50 is beyond me, but that's what it is. Okay, using the demo from the last time where we already have a couple of topics in place and some data in them, I wanted to show you how to take a look at the __consumer_offsets topic, which again, is these designated topic to store all of the consumer offsets throughout the entire Kafka cluster. Now, in this scenario, I only have one single broker running, but let's take a look at what this offset describe would do. So here, you're going to see it listing all of the partitions that are in this particular topic, which again, is __consumer_offsets, it has a partition count of 50, and it only has a single replication factor, which is a little bit dangerous, but if we tried to set that higher, at this point, with only one broker running, we would get an error. So it's probably only doing a replication factor based on the number of nodes available to it, at this point, by default. So we now know the committed offsets are stored in a topic on the cluster, but how does the committed offset values get produced into the topic? Remember the class consumer coordinator we touched upon earlier? This is the responsible object for communicating to the cluster and ensuring the committed offsets are produced into the topic. This means that a consumer is also a producer of sorts. We've covered quite a bit more about offsets, but there are a few more points I want you to add to your growing Kafka encyclopedia and the offset mode I mentioned a couple of slides back. There are effectively two modes, automatic and manual. Automatic being the default. To switch to manual mode, you simply set enable.auto.commit property to false. Of course, by doing this, the property for auto commit interval is irrelevant, and, therefore ignored. When you do this, you are taking full control of when you want Kafka to consider a record to be fully processed. This is a fairly advanced, but not uncommon scenario. We won't get into it in depth in this course, but I will give you a high-level overview of why, how, and what it means to use it. The API for manual offset management consists of two methods, commitSync and commitAsync.

CommitSync and CommitAsync for Manual Offset Management

You would use the commitSync method when you want precise control over when to consider a record truly processed. This is common under circumstances where higher consistency and message processing fidelity is required, where you wouldn't want to retrieve and process new records until you're sure the ones you've currently processed are committed. It is suggested that you invoke this method after you have iterated and processed a batch of consumer records in the for loop, not during. I mean, you can invoke it after every single message, but that level of paranoia may not buy you anything extra other than added latency, because the call is, as the name suggests, synchronous, and will block the thread until it receives a response from the cluster. Hopefully, the response is a successful confirmation, because if it is an exception, there's not much you can do and you'll just have to start the process of recovery. The good news about commitSync is that it will automatically retry the commit until it succeeds, or again, if it were to receive an unrecoverable error. To control the retry attempt interval, you would work with the retry.backoff.ms setting, and it's similar to the setting found in the producer configuration as well. The default is 100ms, so it will retry a lot. With this manual offset management mode, you may be trading throughput and performance for control over the consistency. The synchronous blocking nature of the call can add a measure of latency to the overall polling process. Like the commitSync method, you would use it's asynchronous sibling to control when to consider your message as truly processed. The difference here is due to the asynchronous nature of the call, you may not know exactly when the commit succeeded or not. Because of this, the commitAsync method does not automatically retry when a commit doesn't happen. Retrying without knowing whether the first attempt succeeded or failed can lead to ordering issues and possible duplication of records; however, there is a useful option to pass in, and that is a callback. That callback will be triggered upon the commit response from the cluster. With this callback, you can determine the status of the commit and act accordingly. Since this is a non-blocking option, the throughput and overall performance is going to be better because you will not have to wait for a response to continue processing. However, I wouldn't recommend this option unless you register a callback and can handle the responses accordingly. Otherwise, you could end up in a worse situation altogether.

When to Manager Your Own Offsets Altogether

So we've nearly completed our journey through our Kafka consumer map. We've discussed at length the important process of managing offsets as part of the overall consumer's responsibility for reliably processing messages. The place where offset management occurs is after the poll method has timed out and presented records for processing. Whether this is an auto commit operation happening behind the scenes or an explicit call to one of the commit APIs, the commit process will take a batch of records, determine their offsets, and ask the consumer coordinator to commit them to the Kafka cluster via the consumer network client, which it does immediately. When the offsets have been confirmed to be committed, the consumer coordinator updates the subscription state object accordingly, so the fetcher can always know what offsets have been committed and what next records it should be retrieving. There are a lot of things Kafka can do for you out of the box. But for advanced scenarios, you may need to go outside the box entirely and leverage Kafka's APIs for complete offset self-management. We discussed many of the facilities for doing this, and I would encourage you to explore the APIs further. The question is what are some common reasons for taking control of the offsets? As we touched upon already, one of those big reasons is consistency control. Depending on your consumer application's purpose in the larger system, you may need finer-grain control over when a message is processed and considered ready to commit. If you leave it to the auto commit behavior, the only determination of done will be when the auto commit interval expires, and that may not be enough to ensure higher levels of consistency. Being able to treat the steps of message consumption and processing as a single atomic operation, that's a good reason. This is commonly understood and known in transaction processing systems as atomicity. It is an important attribute of highly consistent systems and may be required by your particular system. The main reason independent offset management becomes a common scenario with Kafka is the desire to achieve exactly once-semantics of message processing. Because of what can go on within a distributed system like Kafka, there is quite a bit of surface area to get messages out of order or have duplicates. This surface area is largely attributed to the scalable nature of how Kafka handles partitions and automatic partition reassignment and rebalancing, topics which we're going to cover next. But in order to get an exactly once system, you will likely need to manage offsets and the content of the message and/or the result of its processing in the same store where you can have full transactional control of the scope.

Scaling out with Consumer Groups

Up to this point, we've discussed a lot about consumers, mostly single consumers. In these discussions, we've been faced with a scary reality. A single consumer may be required to consume from dozens or possibly hundreds of topics, each with countless partitions. That's a lot for a single anything to manage, let alone having to do it with a single execution thread for both retrieving and processing messages. As I said, it simply isn't realistic to expect a single consumer application to take on the entire burden of message processing from a potentially large Kafka cluster environment. The solution is to be able to scale out the number of consumers consuming messages. But having a bunch of consumers independently consuming messages from topics and partitions won't alone solve this challenge of scalability, they have to work in concert with one another. Throughout this course, we've seen how each component of Apache Kafka has a solution for scaling out. If more message production is needed, the solution is to add more and more producers. If we need more message retention and redundancy, we add more and more brokers. If we need more metadata management facilities, we add more zookeeper members. But What about scaling the ability to read and process messages beyond a single consumer? Consumer groups is the answer. A consumer group really is a collection of individual independent consumer processes working together as a team. The only thing required to join a consumer to a consumer group is to use the group.id setting as a configuration property before starting the consumer. When a consumer is part of a consumer group, the task of processing the messages for an entire topic is distributed as evenly as possible amongst the number of consumers. Like any work distribution system, a consumer group can enable higher levels of overall throughput through multiple consumers working in parallel. It can increase the levels of redundancy as the failure or limitation of a single consumer is automatically handled and balanced by Kafka, and with an increased number of working consumers working in parallel, the overall performance can improve as far as the ability to process a large backlog of messages. Let's look at how this works. A consumer group is formed when individual consumers with a common group ID invoke the subscribe method and pass in a common topic list. Behind the scenes, a designated broker is elected to serve as a group coordinator, whose job it is to monitor and maintain a consumer groups membership. In addition, the group coordinator works with the cluster coordinator and zookeeper to assign and monitor specific partitions within a topic to individual consumers within a consumer group. From the second a consumer group is formed, each consumer is sending regular heartbeats at an interval defined in heartbeat.interval.ms property setting. The group coordinator relies on this heartbeat to determine whether an individual consumer is alive and able to participate in the group. The session.timeout setting is the amount of total time a group coordinator will wait after not receiving any heartbeats before it will consider the consumer failed and take corrective action. The group coordinator's main priority is to ensure that the purpose of the group is being met, and that purpose is sharing the load of a topic's messages amongst all of its consumer group members. If there is a consumer that isn't available to share in that load, the group coordinator will remove that consumer and reassign its partitions to another consumer in the group. This is called a consumer rebalance, and it is, as you can imagine, quite a process. If there aren't any additional consumers in the consumer group, the first consumer in the group will get the new assignment and in this case, end up taking on twice the load to compensate for the failed consumer. When this happens, the first consumer now has to figure out where the failed consumer left off and catch up, hopefully without processing duplicate records. This is why offset management can make or break the Kafka consumers because if it is not handled correctly, the ability for the consumer group to failover and rebalance itself can be compromised. Consider the case if the failed consumer processed messages but failed to commit them before it failed. The first consumer will likely reprocess the messages because it had no idea what records were actually committed or not. If and when a new consumer joins the group, another rebalance will occur and the same rebalance protocol will be followed. It's not just a consumer coming in and out of a consumer group that will cause a rebalance, it's also the addition of a new partition. Generally, a consumer group is planned for each application that requires message flow from one or more topics. For example, in this case, we have a consumer group called orders, and it could subscribe to any number of topics related to order management, because the application backend it is intended to serve is an order management system.

Consumer Group Coordinator

Let's spend some more time on what happens during a consumer group rebalance, specifically, when a new consumer in the group is assigned a partition that was previously assigned to another consumer. When the new consumer is assigned a partition, in this case partition 0, it needs to know what offset it should start from because it does not have a current position for this particular partition. Fortunately, the consumers subscription state object has cached the last committed offset from the previous consumer and can now instruct the new consumer that on its first poll on the new partition that it will start with offset 5 since the last committed offset was 4. This behavior of determining where the new consumer should reset its offset to is configured in the auto offset reset setting. Since the default is latest, the new consumer will start reading from the latest known committed position. Of course, this assumes that the committed offset was accurately and completely committed when the previous consumer was rebalanced. If the previous consumer was in the middle of processing records and didn't have the chance to commit its offsets when the rebalance happened, then there could be a chance that when the new consumer picks up, it could be reading from already processed records, thus creating duplicates. To finish up the discussion on consumer groups, let's highlight some of the important duties of the group coordinator, without which consumer groups wouldn't be possible. The primary purpose is to make sure each consumer in the consumer group is sharing the partition load across the group. Whenever it can, it will assign one consumer to one partition if there is an equal number of consumers and partitions. However, if there are more consumers in the group than there are partitions, the extra consumers will be idle, creating a consumer over-provisioning scenario that the group coordinator can't change unless partitions become available. When a partition does become available, the group coordinator will initiate the rebalancing protocol by engaging each consumer coordinator in the impacted consumers to start the process of rebalancing so the newly added partition can be assigned to an appropriate consumer. The rebalancing protocol is also initiated during a consumer failure, as we just illustrated.

Demo: Consumer Groups

In this final demo, I will extend the Java-based consumers we've already seen to take on teaming responsibilities within a consumer group. We will have three independent consumers, each sharing the same group ID and each participating in the task of processing messages from a single topic with three partitions. In this simple example, look for how each of the consumers are assigned a partition and are sharing the work of processing messages. Also, look for what happens when we add an additional consumer and when we add an additional partition. Finally, we'll observe what happens when a rebalance is forced. In this demo, we're going to launch three basic consumer applications, which contain identical polling and processing logic. Here it is. It's basically pretty simple. We're just taking the string that's coming through in the producer, and we're taking the value and upper-casing it, and that's it. And each one of these three is identical, and as far as producing, we have a loop running, and we'll produce 99 records of the alphabet in lowercase, and all the processing that it will do on the consumer side is just to turn those into uppercase. All right, so let's get these consumers running. We'll start one by one in each window, and they'll sit there and wait in a polling loop until our producer starts running. And we'll do that right now. Now, remember, each of these consumers are in a consumer group, all subscribing to the same topic, and the producer is going to be publishing to that topic. So here we go. First one, the second one, and the third one. So as we can see here is this particular consumer as part of the consumer group was getting all of the messages being sent to the first partition, and it took the lowercase alphabet and uppered it. The same thing could be said of the second consumer in the consumer group. It was taking partition 2, taking the value, and turning it to uppercase. And then consumer 03 from the consumer group was taking partition 0.
Okay, let's take a look at what happens when we add a fourth consumer to the consumer group. Now, remember, we have three consumers to the consumer group, and we have one

topic with three partitions. We have an over-provisioned consumer group. But let's see what happens. So we'll start each one of these consumers, let them sit and wait for messages, and then we will produce to them. So now we have four in the consumer group, and we can see it in here that basically there are four consumers in the consumer group test group. And as I was adding them, they added to them and registered to the group coordinator. So now let's produce some records. So we see that the first one receives some records. The second one receives some records. The third one did not receive any records, and I believe the fourth one will have received some records. So why didn't the third one receive records? Well, because there are only three consumers in the consumer group and there are three topics. So one of them is just sitting idle, and that one happened to be the third one. The first one here got partition 1. The second one got partition 2, and the fourth one got partition 0, and they did their job by taking the value and turning it to uppercase all the same. Okay, so now that we've added an additional partition, and now we have even numbers of consumers to partitions to consume, let's rerun the producer and see how it now distributes the messages across an even number of partitions because, basically, when we added the new partition, it forced a rebalance. So let's see what happens. So for App01 got messages, App02 got messages, App03 got messages, and App04 got messages. Four got partition 0, 03 got 1, 02 got 3, and 01 got 2. So it did rebalance. It did recognize the new partition, and it did assign the new consumer in the consumer group to that partition. All right, now that we have an even consumer group, even with the number of partitions, let's force a rebalance. Now the way you do that is by a basically killing a couple of consumers because if they are no longer present, the group coordinator will not find them. They won't get their heartbeat, and they will remove them from the consumer group. So let's kill, let's kill 04 and 03. So now they're sitting there, and let's go back over to here. Now, remember, it takes a little while for the rebalance to occur because, basically, at this point in time, the group coordinator is waiting around, waiting for some heartbeats from consumer 04 and 03, and so far it's not getting them. And then after a while, it's going to now say, Oh, my session's going to start timing out, and I'm going to remove the dead consumers from the consumer group. So these consumers are still running. So let's now produce some more records. So now you'll see that consumer 001 took some records. Consumer 02 took some records. And, of course, 03 and 04 did not because they were unresponsive. But what happened was 01 and 02 basically were reassigned the partitions that 03 and 04 had. In this case, it was partitions 1 and 0 that was reassigned to consumer 01, and 2 and 3 were reassigned to consumer 02.

Configuration and Advanced Topics

We have covered a lot of consumer configuration properties in this module, but we haven't covered all of them due to time and focus. Since the performance and throughput of consumer processing can be affected by various settings and combinations of settings, I thought I would list a few of the more prominent settings here. My goal is to call some of these out so that you can spend some extra time studying them and experimenting with them to understand how the consumer behavior varies. These settings fall into a category I would call performance and overall efficiency. The fetch.min.bytes setting sets the minimum number of bytes that must be returned from the poll. This ensures that you don't have wasted cycles of processing if there aren't enough messages to process. This setting is analogous to the batch size setting on the producer. The max.fetch.wait.ms setting establishes the amount of time to wait. If there isn't enough data to meet the threshold set by the fetch.min.bytes setting. This is somewhat analogous to the linger.ms setting in the producer. To ensure that each poll isn't retrieving more data than your processing loop can handle safely, you can set the maximum number of bytes per partition that the poll can retrieve per cycle. Related to this is the setting to establish the maximum number of records allowed per poll cycle. These last two settings are useful to throttle the number and size of each incoming batch of records should your processing loop be such that a lot of time is spent in processing and you don't want to risk a session timeout. We covered a lot, but there are some things we just weren't able to cover in this introductory course. Each of these fall into the category of taking complete control of the consumer's behavior. You can specify how you want a consumer to read a partition's messages by using the consumer position control API. It comprises of three methods. First is the seek' method, allowing you to specify the specific offset you want to read in a given topic and partition. There's also seekToBeginning, which indicates that you want to start from the beginning of a group of a specific topics and partitions. Obviously, seekToEnd is the opposite of seekToBeginning. And then, there's the ability to literally control the flow of messages through pause and resume APIs. These allow you to determine which topics and partitions you may want to pause while focusing on other topics and partitions considered a higher priority. This is useful for situations where a single consumer has to read from multiple different topics and partitions. Finally, there are the rebalance listeners that you can leverage when subscribing to topics in a consumer group. These listeners will notify you when a rebalance event occurs so you can manage how you want to handle the offsets yourself.

Summary

Throughout this module, we focused on the internals of a Kafka consumer. We started with a high-level map and started to drill down component by component and in the process covered a lot of things, such as the required consumer properties and their internal consumer representation as the ConsumerConfig object, which was analogous to the ProducerConfig object in the producer, once again how the term message is really a reference to a ConsumerRecord, much the same way it was called the ProducerRecord from the point of view of the producer in the last module. We talked about how to subscribe to topics and how to assign yourself partitions. We discussed the important differences between the two and when it may be appropriate to use one over the other. We also talked about the end-to-end consumer polling process, complete with the poll method, the for loop for processing records, and all the internal consumer objects that enable to consumer to function. We also discussed the various different modes and options for managing offsets in the consumer. Additionally, we covered the way Kafka consumers can scale out through consumer groups and how using consumer groups can increase the overall throughput possible through parallel consumers, but also the degree in which consumers can be fault tolerant and robust amidst failure or cluster changes. We covered throughout the various configuration settings and how they control the behavior and non-functional outcomes of the consumer. And we had some demonstrations showing how to create and operate a Java-based consumer and consumer group. With the core components of Apache Kafka now covered, in this last module, we'll cover the broader ecosystem that Apache Kafka finds itself in, including its current challenges, opportunities, and the most recent areas of continued development and evolution.

Exploring the Kafka Ecosystem and Its Future

Apache Kafka's Success and Challenges

At this point, we've covered the major components of Apache Kafka. I hope you're feeling equipped with enough knowledge to really start exploring and building big data solutions using Kafka. This module is about taking a step back and surveying the landscape in which Apache Kafka exists. We will discuss the success it has enabled to continue challenges it faces and how it is evolving to meet those challenges head on. The main use cases for Apache Kafka today have more or less remained the same since it was first created by LinkedIn. It's hard to go anywhere and have discussions about the challenges of data management in modern day enterprises without the mention of Apache Kafka. It is generally regarded as a primary solution for connecting disparate sources of data. With its flexible client APIs, it is possible to write data connectors and syncs for practically any data source. Many of these have been shared and commercialized at this point, and we'll discuss more about them in the coming slides. Apache Kafka is becoming the de facto option for building data supply chains and pipelines that can displace long-standing, expensive, and fragile ETL environments. Within this context. Apache Kafka fits really well with other "Big Data" solutions like Hadoop and Spark, amongst others, because of its ability to integrate, move, and store data at massive scale. Essentially, reference architectures for data management has started to become established within the industry, and Kafka is a central piece to many of them. However, sometimes new solutions introduce new problems and reinforce old unsolved problems. Despite the vast utility that Kafka offers today's organizations, there are still a lot of gaps that the industry is being pressured to solve. For example, having the ability to unmask and manage more data actually makes it harder to govern data and manage its rapid evolution. The commoditization of technology and business specialization demands lower overhead and less investment. So regardless of how useful something in technology is, it will always be a challenge the more inconsistent or costly it is to wield. Data is becoming more and more of a strategic differentiator. In the last five years, there has been an arms race for anyone and anything that can manage more and more data. The next 5 years is going to be all about fast data, how to rapidly gain utility from it, particularly in predictive, deep learning contexts. Over the next few slides, I will use these three challenge areas to describe how Kafka is evolving to address these pressures.

Challenges and Solutions for Data Governance

I'll start off with Kafka's challenges with data governance and evolution. Let's consider the common case of a large and growing network of Kafka producers and consumers. As you know by now, each producer is defining its message contract to publish. You'll recall from module 4 that that contract is based on a fairly rigid type dependant serialization system. We didn't talk a lot about this or nearly in as much detail as I would have liked. But in advanced cases, it becomes infeasible to restrict message contracts solely based on the built-in serializer types. Eventually, as more data diversity is introduced from different systems, custom serializers come into play. Throughout the message lifecycle, there can be hundreds of different contract versions in motion, with each producer publishing massive amounts of data into Kafka. Of course, it takes consumers to derive any sort of value from the data being produced, but they have to be able to do it by reading the data first, which they're able to do through deserializing the message content. This means that with a growing diversity of producers and the data they're publishing, there is an increased complexity all around because consumers have to work with the data being produced and the specifications for each type of method it's consuming. The challenge with Kafka in this common scenario is the lack of some common means of cataloging, registering, and reconciling the disparate message specifications and compatibility mappings between the serializing producers and the deserializing consumers. Confluent is one of the biggest Apache Kafka ecosystem contributors, and they have recognized the challenge we just covered. Fortunately, they have started to take steps to address it by introducing the Kafka Schema Registry. This welcome addition to the Kafka family deserves its own course because of the richness of its functionality. But for now, let me introduce how it addresses the challenges we just covered. One of the more universal data serialization formats out there today is called Apache Avro. It was created to address the challenges with disparate data formats and serialization schemes that make integration and interoperability difficult. It is a self-describing version format that has broad industry adoption. With Avro, producers can serialize their messages in an Avro-versioned and self-describing format and expect them to be deserialized seamlessly by the consumers. As the name suggests, the schema used by both producers and consumers can be registered and version managed centrally within the Kafka cluster environment, allowing for easy, RESTful service-based discovery and version compatibility reconciliation. Now the great thing is the source is fully available on GitHub and available through the generous Apache version 2 license.

Challenges and Solutions for Consistency and Productivity

Let's consider a typical enterprise data environment. There are many sources and targets for data. Kafka has made quite a reputation for itself in being the conduit between these sources and targets. But the challenge has been a lot of duplication of effort in terms of writing producer and consumer applications that connect the sources and targets together. The crazy thing is, when you think about the work to integrate data stores, they're all more or less the same. I mean, look at relational database management systems, for example. They've been around forever, and there's only so many mainstream database vendors out there, yet, across the industry, it seems that within every company there's the same duplicated effort to write integrating producers and consumers for those very same data stores. Talk about reinventing the wheel. The same could be said about file systems, NoSQL databases, search engines and even Hadoop, amongst others not mentioned. The challenge with Kafka in this scenario has been the lack of consistency in providing a common framework for integrating data sources and targets. It was always left to the individual engineers to create their own solutions, using the generic producer and consumer client APIs. With each integration effort, there is cost not only to develop, but to maintain, and that isn't a very efficient or even productive use of time or effort to do something so common. Furthermore, not every company has the resources to develop and maintain these things, which are really becoming commodities at this point. With the 0.10 release of Apache Kafka, a new framework and marketplace was introduced to address this challenge head-on. It's called Kafka Connect and the Connector Hub. As with the case of the schema registry, this new innovation deserves its own course to give it justice. The Connect framework is an API for developers. It is intended to make the job of connecting data sources and targets easier and more consistent. The goal is to standardize on a common approach for integrating diverse data sources with standard producer and consumer applications. This is awesome because writing highly performant and reliable consumers, for example, can be really hard and complex, as we covered in the last module. So having a framework to simplify and standardize this is a huge step forward, Now currently, many of the developers using this framework are those that work for the leading technology data providers who have started to include a Kafka connector as part of their product roadmaps. Oracle and HP are some noteworthy examples of this. Currently, there are over 50 platform connectors available that are designed to connect to many different products and services, and that list is growing. Confluent itself has created many of these connectors, and they also provide an online portal they call the Connector Hub. They invite anyone and everyone to develop and contribute a Kafka connector using the API and that online portal for distribution. As adoption grows, this is bound to drive more consistency and greater productivity in Kafka-based data integration initiatives. Overall, it's going to get cheaper and faster than ever to get Kafka integrated in enterprises.

Challenges and Solutions for Fast Data

Within the last few years, there has been a lot of hype around predictive analytics, machine learning, real-time, stream-based, whatever buzzword of your choosing. There are multiple technology platforms that all propose to offer a unique ability to deliver upon this hype for real-time or stream-based analytics. Some of these platforms are legitimate, viable solutions such as Apache Storm, Hadoop, Cassandra, and Apache Spark. Again, Kafka is generally found in the middle. But the problem is each one of these technologies introduces a unique and mostly complex model for development and operation. Each have their own API and cluster-based management approach to distributed systems. Kafka, itself, as we've covered in this course, has its own API and vast cluster-based model. So if you have all of these technologies under the same roof, so to speak, that's a tremendous amount of technology to manage and maintain all for the same goal of achieving the ability to process and analyze data in real time. Touching on the last challenge, this introduces consistency and productivity challenges and integrating it all together. With Kafka generally being positioned between these technologies for integration, it would need an army of producers and consumers to keep the streaming pipes flowing. The challenge here is pretty obvious. Now I'm not saying all of these different platforms are present in each environment, but many are because they each have their own strengths and advantages that complement the weaknesses of the other. But regardless of whatever of these systems come and go, one thing is becoming more consistent. And that is the place Apache Kafka finds itself within these organizations. The 0.10 release of Apache Kafka was a huge one. In addition to Kafka Connect, a new client library for real-time, stream-based processing was introduced. This library is called Kafka Streams. The real value proposition of this is that for organizations that have already made an investment in Apache Kafka, they can now have streaming data capabilities without having to install, run, and maintain all of those different platforms. All they need is their existing Kafka environment. Given everything we've learned about Kafka in this course, I am sure you can see how adding this capability to Kafka wasn't that much of a stretch. I mean, consider what Kafka already does with data in motion and how it does it. This is significant because it doesn't require anything more. I mean, theoretically, Apache Kafka could be the only infrastructure solution required. But in reality, many enterprises have good reason to additionally invest in Apache, Hadoop, and Spark. So it may be that Kafka itself isn't the only big data system in place. But at the very least, it can be the only system needed for stream-based processing. Regardless, the potential to reduce and consolidate into fewer systems is now a very real possibility. Think of what that can do to lower the initial investment and overall total cost. As I said, Kafka Streams is a client library that works with the Kafka cluster. As you've learned in this course, that's exactly like Kafka producers and consumers. They are client libraries too. And just like we did with the producer and consumer client libraries, Kafka streams can be embedded within Java-based applications, making the barrier to adopt lower than any other platform offering stream-based processing. Think of it this way. If you already have producers and consumer applications, why not just extend them with the Kafka Streams library to provide stream-based processing capabilities all within the same place? This is an exciting area that I hope you'll continue to explore.

Apache Kafka's Ecosystem and Summary

Everything that Kafka is today, and what it will be tomorrow, is made possible through the growing and healthy ecosystem of adopters and source code contributors. These are but a few of the big names that not only have based significant parts of their business on Apache Kafka but also make generous contributions back, allowing all companies and organizations, big and small, to benefit. That's the beauty of the open source ecosystem in which Apache Kafka is firmly placed. In this module we covered the undeniable success that Apache Kafka has had since the beginning. It has enabled organizations to solve some of data management's biggest problems. But as I said, in the process, it has introduced new challenges. Many of the challenges facing Kafka and the data management industry in general stem from the rapidly growing and changing landscape. Data volumes, velocity and variety are increasing exponentially, and as a significant player in this landscape, Apache Kafka can't rest on its laurels. Luckily, with the vast and supporting ecosystem it has, Kafka has evolved to meet these challenges and establish reinforced foundations upon which to build further for many years to come. We discussed some of these recent innovations, like the Schema Registry, Kafka Connect and Kafka Streams. I hope you'll agree with me that there is a promising future ahead for Kafka and the many technology professionals and organizations that invest in it. We have come to the end of this course. I hope you learned a lot, at least enough to continue your journey in learning more. It's always hard to decide where to invest your limited amount of time. I personally faced this challenge, as the course author, in determining what details to focus on and what details to sacrifice, because I wanted you to get the most out of this course within a limited amount of time. I hope I succeeded, but it is hopefully just the beginning for you. I encourage you to continue learning about Apache Kafka and trying it out. It's a solid bet to make as a technology professional.