4. Understanding Topics and Brokers

Contents

[1. Introduction and Apache Kafka Setup Demo 1](#_Toc185727832)

[2. Apache Kafka Topics 3](#_Toc185727833)

[3. Demo: Starting Apache Kafka and Producing and Consuming Messages 5](#_Toc185727834)

[4. Module Summary 9](#_Toc185727835)

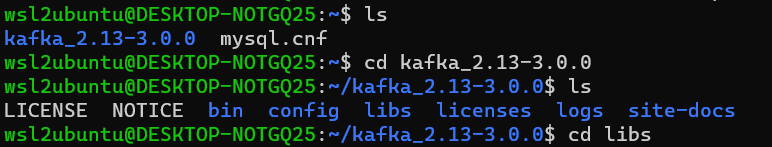
# Introduction and Apache Kafka Setup Demo

=>slides: Pg. 1

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.

=>slides: Pg. 2

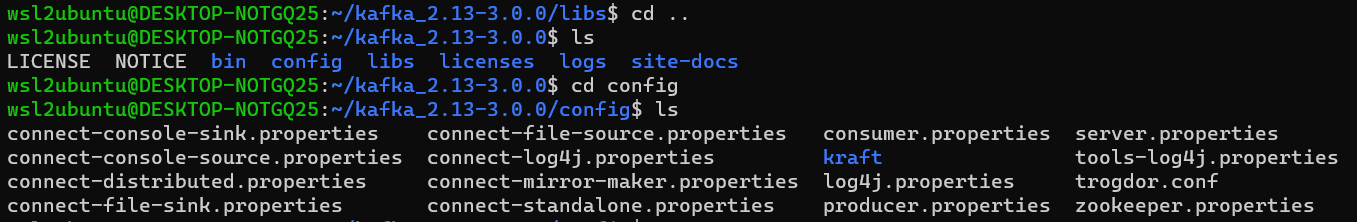
Before we go much further, I want to show you how to get Apache Kafka installed on a development machine like a virtual machine running Linux. We will not go into a lot of configuration details at this point. Thankfully, we don't need to. Kafka is ready to go with a basic installation, and that's what 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 or later 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. 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 browserWe can see that it's downloaded as an archive, so now we'll extract it using the tar command. Now we 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.



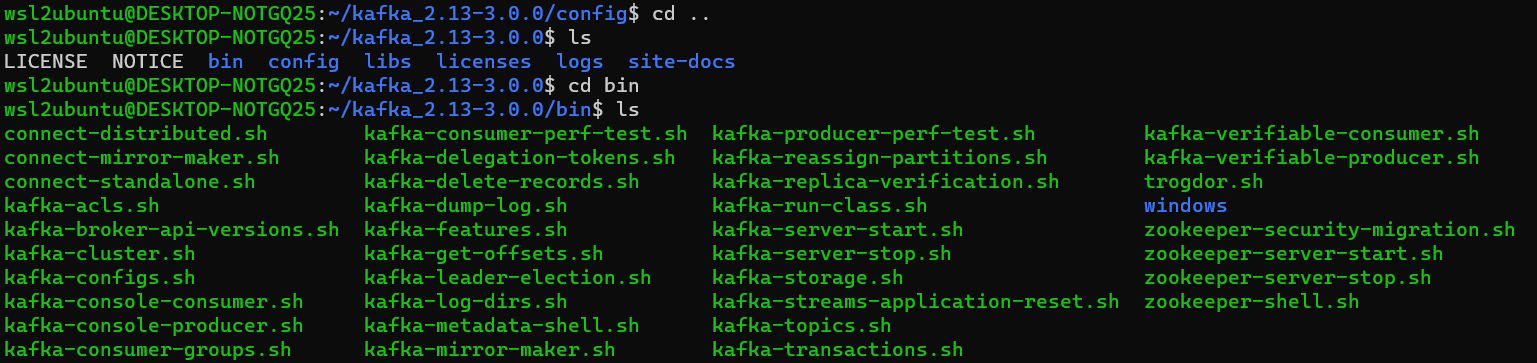
You'll notice there at the bottom these archive for ZooKeeper and its client library. This enables Kafka to be a self‑contained installation, not requiring ZooKeeper to be installed prior, which is convenient to get started quickly.



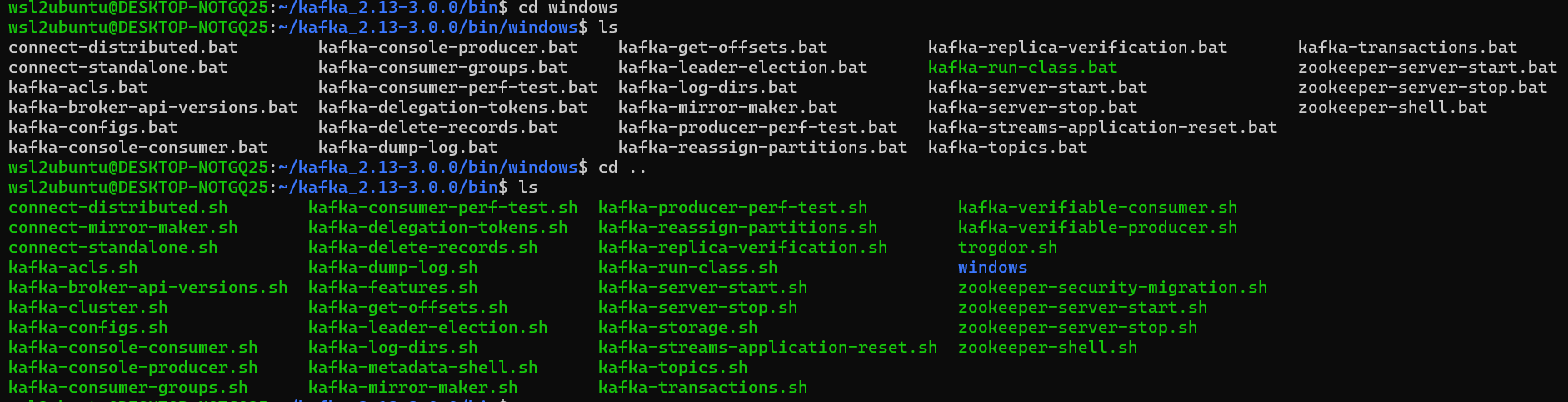
Next is the config folder. This is an important folder because here you'll find all of the files you'll need to configure all of the components of Apache Kafka. Some of these files are out of scope for the current course, but the ones related to the broker, the producers, and consumers, we'll get into 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

**=**>slides: Pg. 3

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.

**=**>slides: Pg. 4

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.

**=**>slides: Pg. 5

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.

**=**>slides: Pg. 8

**Kafka Replication Factor**

The replication factor in Apache Kafka defines the number of copies of a partition that are maintained across different brokers in the Kafka cluster. This replication ensures data durability and high availability.

**Key Points:**

* **Fault Tolerance**: If one broker fails, other brokers can serve the replicated data, ensuring no data loss.
* **Data Durability**: Even with broker failures, data remains safe and accessible.
* **Configuration**: The replication factor is set at the topic level, and a common practice is to use a factor of 3 for a good balance between durability and resource usage.

In summary, the replication factor is crucial for maintaining data integrity and availability in a Kafka cluster. 😊📊🚀

**=**>slides: Pg. 9

**What are Kafka Partitions?**

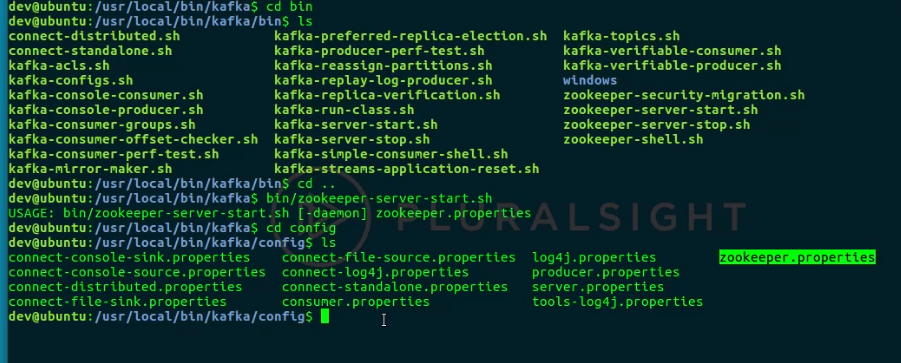
* **Unit of Parallelism**: Partitions allow Kafka to handle high throughput by dividing a topic into multiple parts. Each partition can be read and written independently.
* **Data Distribution**: When you produce data to a topic, it is written to one of its partitions. Kafka ensures even distribution of messages across partitions for load balancing.
* **Ordered Data**: Within a partition, Kafka maintains the order of records. Consumers read records in the order they were written.
* **Fault Tolerance**: Partitions are replicated across multiple brokers. The replication factor determines how many copies of a partition are kept.

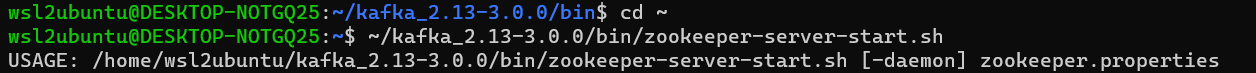
**Key Points:**

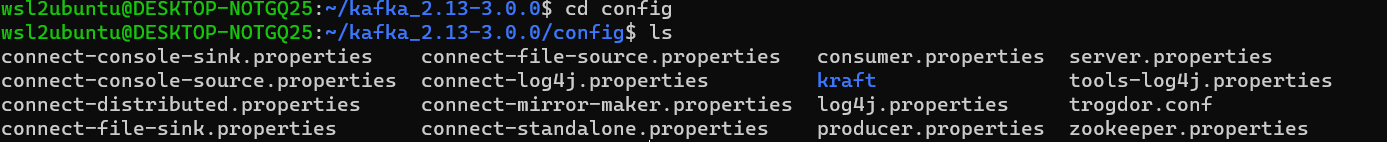
* **Producers**: Can decide which partition a message goes to, typically using a key. If no key is provided, Kafka uses a round-robin approach.
* **Consumers**: Consumers in a consumer group are assigned partitions, ensuring that each partition is consumed by only one consumer in the group at a time.
* **Scaling**: Increasing the number of partitions allows for greater parallelism and throughput. However, changing partition numbers can impact data distribution and consumer offsets.

# Demo: Starting Apache Kafka and Producing and Consuming Messages

**=**>slides: Pg. 10

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 srvr. This gives us extended details on 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.

Next, let’s start the Apache Kafka server:

Start Apache Kafka

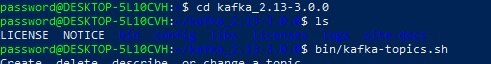
=====================

From the root of Apache Kafka to start Apache Kafka.

~/kafka\_2.13-3.0.0/bin/kafka-server-start.sh ~/kafka\_2.13-3.0.0/config/server.properties

Now with the server started, we'll create a topic.



bin

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.

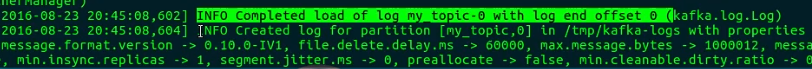


Take a look at the command I just typed in to create a topic. Yes, that's quite a few things needed to successfully create a topic. But let's talk a little bit about why. aside from the obvious create and topic commands, you'll notice we needed to pass in the ZooKeeper server. This is because there could 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.



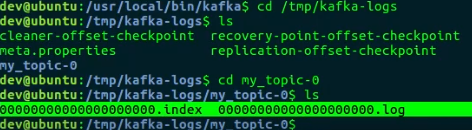


When the topic was created, some interesting things happened behind the scenes. First, ZooKeeper scanned its registry of brokers and made a decision to assign a broker as the leader for the topic, my\_topic.





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.

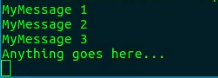


You do this with the kafka‑topics shell program with the option of list, and then, of course, you have to pass the ZooKeeper server. Now that we have a topic, let's produce and consume some basic messages. First thing is to instantiate a producer. By now, you can predict how we'll do this.

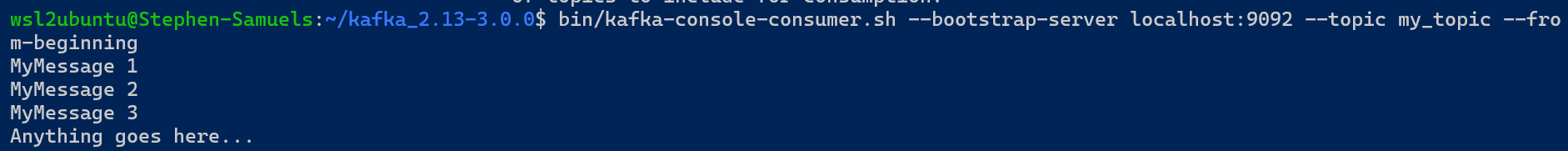


That's right, a handy shell program called kafka‑console‑producer.sh, another case of a lot of usage hints. In the next module, we'll cover the majority of these, but

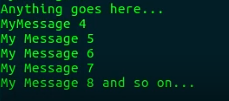
for now, we'll just go with the minimum to produce some basic messages. Once you've executed this command, you can keep the terminal window open most of the time, or as long as you like,



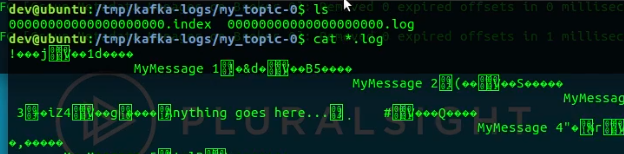
and type whatever you want here. Everything you type and follow with Enter, you will cause the producer to send the message to the broker, which will then commit it to its log, waiting for a consumer to consume them. So let's get the other side working. Like the producer shell program, there is a consumer 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.

# Module Summary

**=**>slides: Pg. 11

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.

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