Kafka Tutorial

Klaus Haller

24.1.2020

Part 1 "First Steps": Overview & Architecture

In this tutorial, I provide a broad overview on the Kafka technology for development and operations as well covering the following steps. In the first tutorial, you have:

- A basic understanding of the Kafka architecture
- A single node Kafka installation up and running
- Using the command line interface, you can start a producer and send information
- Using the command line interface, you can start a consumer and receive information

Please note that we do not discuss the scenarios when to use Kafka. Also, the tutorial is based on Windows 10. If you work with Linux, some of the commands in section 1 might differ.

1.1 Kafka Architecture

From the perspective of developers, Kafka is a pub/sub (publish and subscribe) solution enabling various applications to talk with each other. The senders (or "**producers**" in Kafka terminology) do not have to know who might be interested in the messages or events they share. They publish information related to certain topics. The receivers (or "**consumers**" in Kafka) also do not have to know who exactly creates events and message they are interested in. They just have to subscribe to a topic and get all the information.

From an administrator perspective, a Kafka installation consists of a **Zookeeper** application as a kind of orchestrator and one or more **brokers** that provide the actual functionality for producers and consumers.

This first tutorial focuses on a simple installation: one Zookeeper instance and one broker as illustrated in the figure below. Load balancing and fault-tolerance are discussed in Part 2 *Kafka Fault Tolerance using Replica* and Part 3 *Kafka Throughput Optimization using Partitions and Consumer Groups* of my Tutorial. Security topics are discussed in Part 4: Kafka Security Basics.

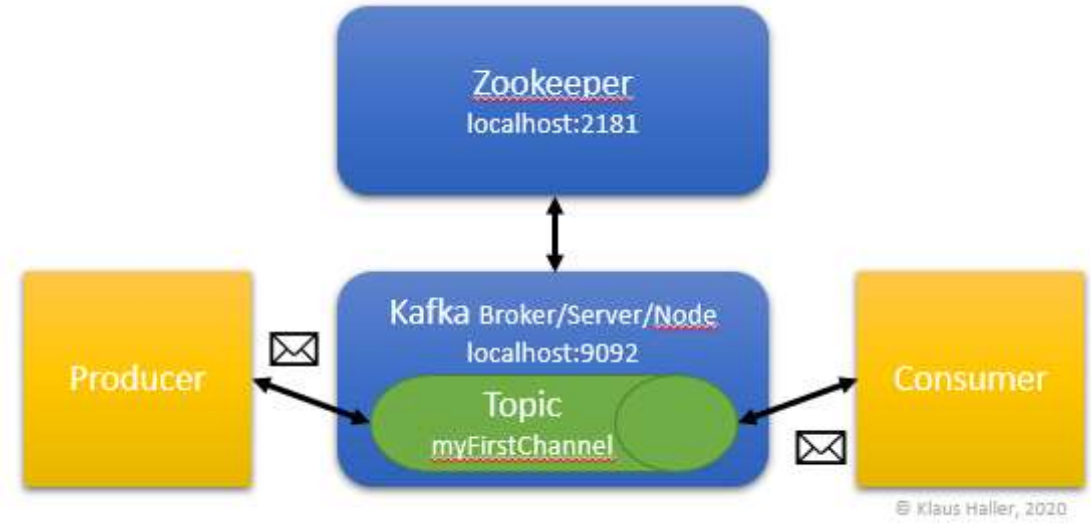

Figure 1: Simple Kafka architecture with one Kafka node and Zookeeper instance as used in the first part of the tutorial.

1.2 Prerequisite Java Run Time Environment

Kafka requires a running Java runtime environment. You can type in "Java -version" in a command shell to verify this. As a result, you get the installed version number. This tutorial bases on java version "13.0.1" 2019-10-15. If the system returns the error message

```
'Java' is not recognized as an internal or external command,
operable program or batch file.
```

you have to install a Java Runtime Environment or finish the installation by configuring path variables.

1.3 Download and Install Kafka Files

You can download Kafka from this webpage: https://kafka.apache.org/downloads. This tutorial bases on version 2.4.0 and binary build 2.13 from December 16th, 2010 and the name of the downloaded file is `kafka_2.13-2.4.0.tgz`.

After the download is completed, unpack the file. You should now see a folder `kafka_2.13-2.4.0`.

1.4 Configure and Start Kafka

There are two configuration files, one for the Zookeeper instance and one for the/a Kafka server. The files are:

```
kafka_2.13-2.4.0\config\zookeeper.properties
```

and

```
kafka_2.13-2.4.0\config\server.properties
```

The only configuration you have to do is setting the path for all the log files in the `server.properties` file by defining the `log.dirs` property:

```
# A comma separated list of directories under which to store log
files

log.dirs=C:\Users\yourusername\kafka_2.13-2.4.0\logs_server0
```

To get a better understanding of the Kafka installation, you might be interested in checking the parameters for the connection from the Kafka server to the Zookeeper. In the Zookeeper's property file, there is a parameter that defines on which port the Zookeeper is listening for Kafka servers:

```
# the port at which the clients will connect
clientPort=2181
```

The Kafka properties file defines where to connect to a Zookeper instance:

```
zookeeper.connect=localhost:2181
```

Thus, in case of this simple installation, the Kafka server will look for a Zookeeper on the local machine.

1.5 Start the Kafka Installation

We are now ready to get our Kafka system up and running. This requires that we first start the Zookeeper. This can be done as follows for a Windows system:

1. Open a new command shell
2. Change to the Kafka bin directory for Windows

```
cd kafka_2.13-2.4.0\bin\windows
```

3. Start the Zookeeper

```
zookeeper-server-start.bat ..\..\config\zookeeper.properties
```

Note: If you work with Linux, you use the .sh scripts in the \bin folder instead.

It takes some seconds until Zookeeper is up and running. You should see the following text on the shell as Zookeeper output:

```
INFO Using checkIntervalMs=60000 maxPerMinute=10000
(org.apache.zookeeper.server.ContainerManager)
INFO Creating new log file: log.9a
(org.apache.zookeeper.server.persistence.FileTxnLog)
```

Note. The last line might only show up the first time you start Zookeeper.

With Zookeeper up and running, we can start now the Kafka server:

1. Open a new (!) command shell
2. Change to the Kafka bin directory for Windows

```
cd kafka_2.13-2.4.0\bin\windows
```

3. Start the Kafka broker

```
kafka-server-start.bat ..\..\config\server.properties
```

Now we just have to be sure that the server actually started. We should see the following last line in the command shell:

```
INFO [GroupMetadataManager brokerId=0] Removed 0 expired offsets in
0 milliseconds. (kafka.coordinator.group.GroupMetadataManager)
```

1.6 Sending a Hello Kafka World Message

In order to be able to send our first message or event using Kafka, we need a topic to which consumers can subscribe to and receive messages that producers send for this topic.

We open a new command shell in windows and run the following commands:

```
cd kafka_2.13-2.4.0\bin\windows
kafka-topics.bat --create --zookeeper localhost:2181 --replication-
factor 1 --partitions 1 --topic myFirstChannel
```

In the same command shell, we start now a consumer service:

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
myFirstChannel --from-beginning
```

The consumer is now ready and waits for a message or event. In order to send a message, we need a producer. Thus, we open another new command shell, the fourth one, and start a simple producer process:

```
cd kafka_2.13-2.4.0\bin\windows
kafka-console-producer.bat --broker-list localhost:9092 --topic
myFirstChannel
```

We now type in “Hello Kafka World!”. Once we hit the return button, we can see the message as well in the consumer window. We have a simple, running Kafka installation and send and received a simple message. We are done with section 1 of the tutorial: First Steps.

It is time to clean up before we begin with part 2 of the tutorial:

- Keep the Zookeeper and Kafka server command shell windows and keep the processes running.
- Stop the consumer and the producer applications and close the command shells.

Part 2: Kafka Fault Tolerance using Replica

If you replicate events on multiple Kafka servers, you still have all the data when one of the server crashes. The other server stores the data as well and is ready to deliver it to the consumers. This is the idea behind replica in Kafka. Note, however, that the overall platform is blocked if there is no Zookeeper up and running.

The concept is relatively simple for replica: There is one leader. It is the only one accepting new messages and events from producers and it is the only one delivering events and messages to consumers. Replica servers "just" get a copy of the events and messages from the leader.

To try out this functionality by ourselves, we extend our infrastructure by increasing the number of Kafka nodes from one to three and adding a new topic using the Kafka replica feature as shown in the figure below.

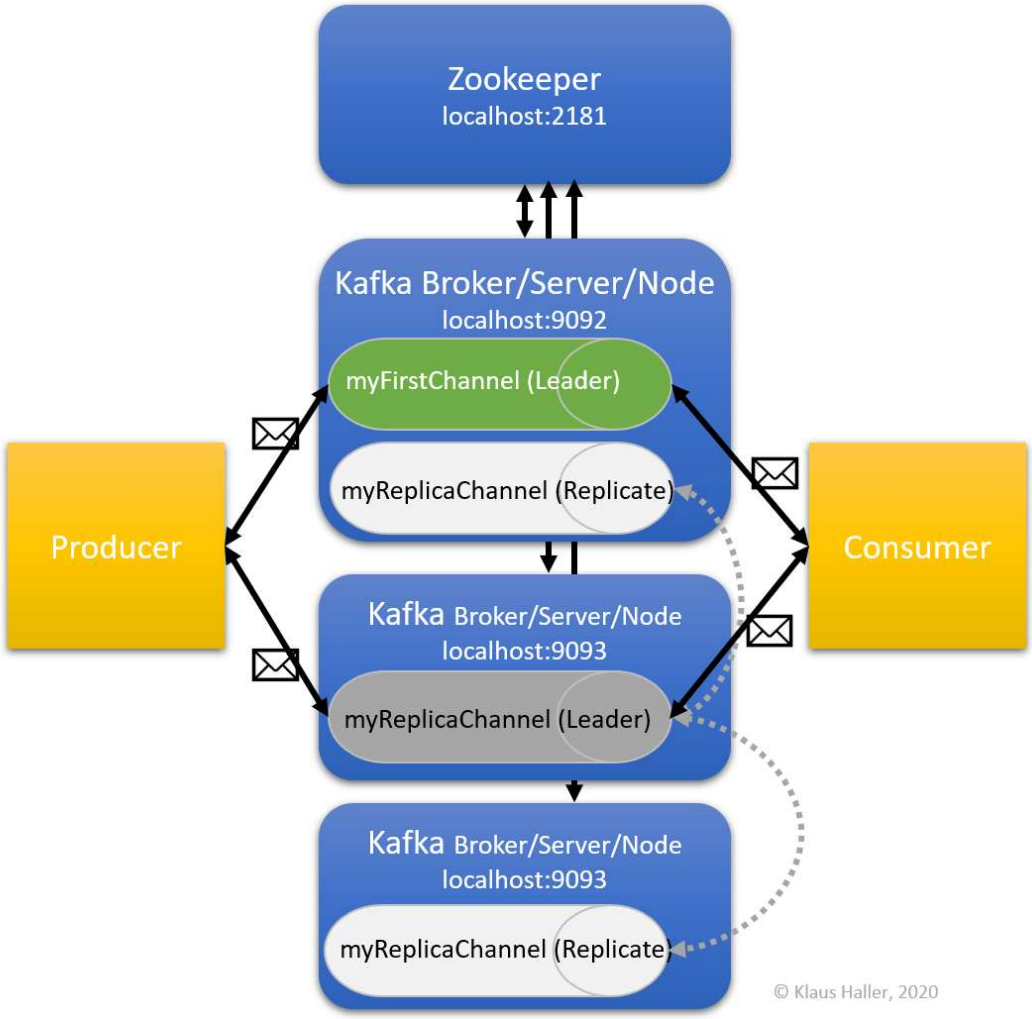

Figure 2: Kafka platform with topic replication

2.1 Configuring Additional Kafka Nodes

We copy the file `server.properties` twice to get two new files, `server1.properties` and `server2.properties`. In these files, we set the broker id parameter to 1 and 2. Also, we have

to configure new log file directories for each of them, since we run all of them on our local machine in this tutorial.

So, the file `server.properties` should contain:

```
# The id of the broker. This must be set to a unique integer for
each broker.
broker.id=0
[…]
# listeners=PLAINTEXT://:9092
[…]
# A comma separated list of directories under which to store log
files
log.dirs=C:\Users\post\Kafka\kafka_2.13-2.4.0\logs_server0
```

The file `server1.properties` should contain:

```
# The id of the broker. This must be set to a unique integer for
each broker.
broker.id=1
[…]
listeners=PLAINTEXT://:9093
[…]
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://localhost:9093
[…]
# A comma separated list of directories under which to store log
files
log.dirs=C:\Users\post\Kafka\kafka_2.13-2.4.0\logs_server1
```

The `file server2.properties` should contain:

```
# The id of the broker. This must be set to a unique integer for
each broker.
broker.id=2
[…]
listeners=PLAINTEXT://:9094
[…]
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://localhost:9094
[…]
# A comma separated list of directories under which to store log
files
log.dirs=C:\Users\post\Kafka\kafka_2.13-2.4.0\logs_server2
```

2.2 Start the Additional Kafka Nodes

The next steps are straight-forward:

1. Open a new command shell.
2. Start a Kafka server using configuration file server1.properties (see Part 1 of the tutorial if you cannot remember the command).
3. If everything went well, you should see the following:

```
INFO [KafkaServer id=1] started (kafka.server.KafkaServer)
```

4. Open another new command shell.
5. Start a Kafka server using configuration file server2.properties (see Part 1 of the tutorial if you cannot remember the command).
6. If everything went well, you should see the following:

```
INFO [KafkaServer id=2] started (kafka.server.KafkaServer)
```

At this moment, you should have four open command shell windows: one for the Zookeeper and three Kafka server nodes. All of them must be up and running to continue.

2.3 Create New Topic with Replication

Now the system is set-up so that we can create a new topic `myReplicaChannel`, this time using the replica features. We want to have one leader and two replica. We open another new command shell and execute the following command:

```
kafka-topics.bat --create --zookeeper localhost:2181 --replication-
factor 3 --partitions 1 --topic myReplicaChannel
```

Do you remember in which folder you have to execute this command?

Now, we create a new producer and consumer in two separate, new command shell windows using these two commands:

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
myReplicaChannel
```

and

```
kafka-console-producer.bat --broker-list localhost:9092 --topic
myReplicaChannel
```

In the producer command shell, we type in “Replication is great!” and press the return key. Directly afterwards, we see this sentence in the consumer command shell window. So, what does this mean?

The implication is simple: There is no change for users respectively producers and consumers if we use the replica feature.

2.4 Checking the Kafka System State

Using the command line interface, we get an overview of the overall system state (in a later part of the tutorial, we look on more production-like monitoring solutions). More precise, we can use the Kafka topics command with the list option:

```
kafka-topics.bat --list --zookeeper localhost:2181
```

As a result, we get a list with our two topics: `myFirstChannel` and `myReplicaChannel`.

Let’s assume we do not only have one or two topics and “just” three Kafka nodes, but 50 topics and 15 nodes. We might be interested in understanding which server is involved in which topic – and

how the actual configuration looks like. Did we specify 3 replica, or did we not consider this because we were experimenting with this topic for a while and forgot about this issue?

Here, we can use the topics command, this time with the "describe" option:

```
kafka-topics.bat --describe --zookeeper localhost:2181 --topic
myReplicaChannel
```

Before continue reading, you might want to compare the result for the `myReplicaChannel` with the one for the `myFirstChannel`. When we submit the command above, we get this result:

```
Topic: myReplicaChannel    PartitionCount: 1
ReplicationFactor: 3    Configs:
Topic: myReplicaChannel    Partition: 0    Leader: 0       Replicas:
0,2,1 Isr: 0,2,1
```

So, what does this mean?

- All information regarding partitions is out of our focus now. We discuss in the next part of this tutorial.
- The replication factor is three with the Kafka nodes with the broker ids 0, 2, 1.
- The leader, i.e., the node all the producers and consumers communicate with, is the node with broker id 0.
- All replica are up to date, because they are all listed after "Isr".

2.5 A Simple Failover Test

Now we want to check what is happening if we lose the leader node. Therefore, we terminate the process by closing the command shell. With our data above, the node with broker id 0 has to be terminated, i.e., the server started with the `server.properties` configuration. If we run the description command again, we will see that now a different broker became the leader. We will have a closer look on this topic later. First, we look on how Kafka enables parallel processing and, thus, a high throughput.

Before continuing to the next section of the tutorial, you can

Click here to go to the next part of this tutorial.

Part 3: Kafka Throughput Optimization using Partitions and Consumer Groups

Kafka is known for being able to process a heavy workload of messages and events. One way is to distribute the topics over various Kafka nodes. If you have 70 topics and 10 Kafka nodes, each node might be the leader for seven topics. Thus, Kafka can parallelize handling various topics while running a single Kafka cluster. In this section of the tutorial, we look on how to optimize the processing of the events of a single cluster using partitions and consumer groups.

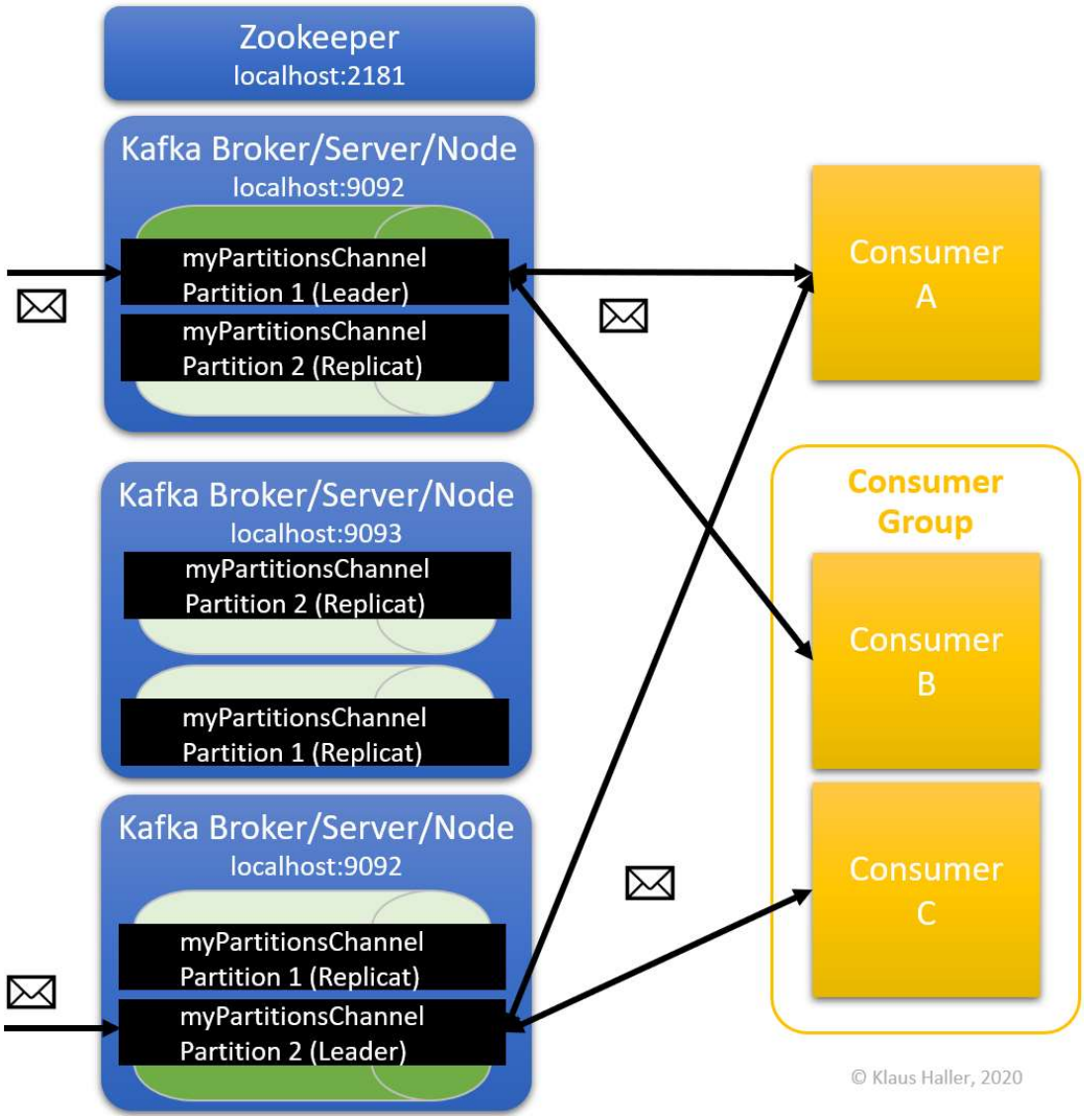

Figure 3: Kafka platform using partitions, consumer groups, and replica features

3.1 Kafka Partitions

Beside distributing topics over various Kafka nodes, Kafka can optimize the throughput also by distributing the workload of single topics over various Kafka nodes. This concept is called Kafka **partitions**. Each partition is a kind of event queue into which producers can write events and from which consumers can fetch events. Thus, events of a single topic can be handled, e.g., by 10 or 50

Kafka nodes. However, there is one aspect to be consider. All events of a partition are ordered, but there is no order between events of different partitions.

The number of partitions is defined on a per-topic level by setting the parameter during the topic creation. Before submitting the command, you might want to check whether your Zookeeper process and the three Kafka nodes/brokers are still running.

The command for the topic creation is the following:

```
kafka-topics.bat --create --zookeeper localhost:2181 --replication-
factor 3 --partitions 2 --topic myPartitionsChannel
```

Now let's have a look how our platform looks like using the topics describe command:

```
kafka-topics.bat --describe --zookeeper localhost:2181 --topic
myPartitionsChannel
```

The result should look similar to that that:

```
Topic: myPartitionsChannel      PartitionCount: 2
ReplicationFactor: 3    Configs:
        Topic: myPartitionsChannel      Partition: 0    Leader: 0
Replicas: 0,2,1 Isr: 0,2,1
        Topic: myPartitionsChannel      Partition: 1    Leader: 1
Replicas: 1,0,2 Isr: 1,0,2
```

We see that the Kafka broker, that is the leader, differs for partition 0 and 1, allowing for workload distribution if the brokers run on different virtual machines.

In a new command shell, we start a consumer listening at partition 0 for the new topic:

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
myPartitionsChannel --partition 0 --from-beginning
```

In another new command shell, we start a consumer listening at partition 1:

```
kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
myPartitionsChannel --partition 1 --from-beginning
```

Finally, we start – in yet another command shell – a new producer:

```
kafka-console-producer.bat --broker-list localhost:9092 --topic
myPartitionsChannel
```

If we type in "Hello1!" and press the return key, the message is displayed by only one of the consumers. What happens if you send more messages such as "Hello2!", "Hello3!" etc.?

3.2 Consumer Groups

Another concept, closely related to partitions, are consumer groups. They are an efficient way to distribute the work processing the events delivered by the Kafka infrastructure. Assume an image processing system. An "analyze image" service might need 2 seconds per image. If there are around 1000 images per second coming from Kafka, one process cannot handle all the load. Here, the concept of Kafka Consumer Groups helps.

Up to now, the rule was that each consumer reads all messages of the topic the consumer subscribed to. If there are two consumers, each consumer reads all message. Each message is read twice. If two brokers form a consumer group, this is different. A message is read only by one consumer of the group. This is perfect for the image processing example. You start 500 image processing processes, each one is Kafka consumer, and all together they form one consumer group. So, every time one of the image processing processes finishes the processing of one image, it contacts the Kafka broker and fetches the next image.

There is just one aspect to be considered: Each consumer of a consumer group can connect to one or more partitions. However, each partition delivers only events to one consumer of a consumer group. Thus, it does not make sense to have more consumers in a consumer group than there are partitions.

3.3 Summary: Partitions, Consumption Groups, and Replica in Kafka

To prevent any confusion, I want to point out the differences between partitions and replica:

- Replica help if a Kafka broker crashes. They ensure that no data is lost. Also, one of the replica nodes can take over the role as the leader interacting with consumers and producers of topics. Thus, such crashes do not impact consumers and providers at all.
- Partitions are measures for throughput of the Kafka platform. You distribute the events of a topic over various Kafka so that there is less load on a single machine. This helps if there are too many events to be handled by a single Kafka node.
- Consumption groups help when processes and applications outside of Kafka struggle to process all the events coming from the Kafka platform. Consumption groups ease distributing and parallelizing such work.

In the next part of the tutorial, we look at security configurations.