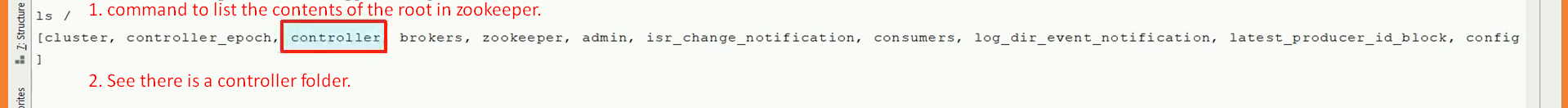
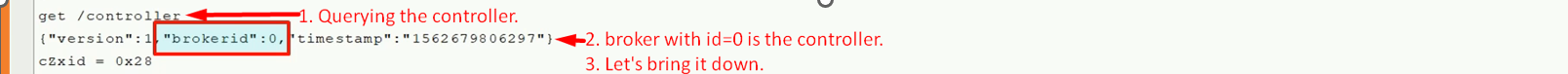
# Completed

1. 23

# 23. Kafka Cluster Controller

1. Cluster Architecture:
   1. Masterless Cluster.
   2. Not Master-Slave Architecture.
2. Administrative Activities such as monitoring the list of active brokers => Controller (**ephemeral controller** **node**)
   1. When other nodes try to be controller **exception 🡪 ‘node already exists’**
3. zookeeper-shell.bat localhost:2181 🡸 This is to ender zookeeper DB. **(**zookeeper-server-start.bat zookeeper.properties to start zookeeper).  
   Following controller entity will be created only when we start at least one broker.  
     
   

# 24. Partition Allocation and Fault Tolerance

1. Each partition is self-contained.
2. Partition Assignment -> **2 racks**, each rack with 3 brokers = **Total 6 Brokers**.   
   10 Partitions & 3 Replication Factor = Total 30 Partition Replicas (10 Leaders & 20 Followers)
3. Distribution Goals: 1. Evenly distribution 2. Follower Partition Replica should be on a different machine.
4. Process:
   1. Make ordered list of brokers. One random broker from each rack alternatively.
   2. Assign in **round-robin fashion**.
   3. Table

      Description automatically generatedA picture containing calendar

      Description automatically generated

# 25. Partition Leader vs Partition Follower

1. **Agenda**:
   1. Responsibility of Leader and Followers.
2. Leader Partition has leader Activity and Follower has follower Activity.
3. Let’s try to understand through example: A leader broker handles all the requests from Producer & consumer.  
   Producer querying for Meda Data and any broker can respond Metadata request. response = List of Leader brokers with Host + port.   
   Now producer can decide to which partition of a topic, a msg to be sent.

A screenshot of a computer

Description automatically generated with medium confidence

1. Follower Broker doesn’t entertain request. They just remain in-sync with their leader and are eligible for leader selection later on.

# 26. The ISR List - In Sync Replica

1. **ISR (In-Sync Replica)** Maintained by Leader Broker in Zookeeper: A set of replicas of a partition for a topic which are in sync with leader. Dynamic.
2. Reasons for not being in sync: **A.** Network Congestion **B.** Follower Broker Crashes and restarts.
3. Chart

   Description automatically generated with medium confidenceDiagram

   Description automatically generated  
   A close up of a text

   Description automatically generated

# 27. Committed Vs Un-Committed Records

1. Diagram, schematic

   Description automatically generatedDiagram

   Description automatically generated

# 28. Minimum ISR List

1. A msg committed if written to all ISRs.
2. **min-insync.replicas=2 Side-effect: Not Enough Replica Exception** in case when P0-ISR contains less than 2 replica and leader becomes read-only.

# 29. Introducing Kafka Producers

1. Sending msg to Kafka Cluster using **Kafka Producer API** (written in java but also available in other). Language Binding.

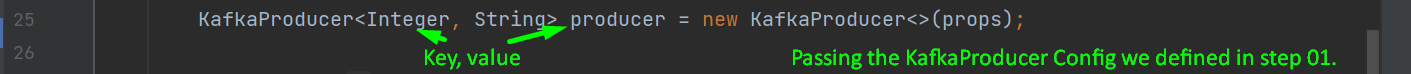
# 30. Creating Your First Kafka Producer

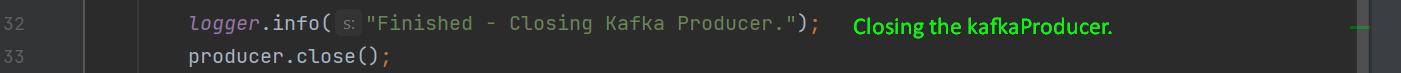
1. **Producer configurations** in Java Properties Class as Kafka is highly configurable.
2. In this first example, we will add 4 basic configurations. These are the bare minimum configurations for producer to work.

|  |  |  |
| --- | --- | --- |
| **Key** | **Value** | **Purpose** |
| **ProducerConfig**.**CLIENT\_ID\_CONFIG** | “HelloProducer” | 1. Simple String that is passed to Kafka Server. 2. Purpose is to track the source of the msg. |
| **ProducerConfig**.**BOOTSTRAP\_SERVERS\_CONFIG** | "localhost:9092,localhost:9093" | 1. List of host/port pairs separated by comma. 2. Producer will use this info to set up the initial connection with the Kafka Cluster. 3. The bootstrap configuration is used only for the initial connection. Once connected, the Kafka Producer will automatically query for the metadata and discover the full list of Kafka Brokers in the cluster. No matter if it is follower or leader broker IP. As after getting full list of broker Ips, producer will send msg only to leader It means you don’t need to send the complete list of Kafka Brokers as Bootstrap configuration because each broker leader or follower has complete broker list to send back to Producer. However, it is recommended to provide two to three Broker addresses of a multi-node cluster. Doing so will help the producer to check for the second or third Broker in case the first Broker in the list is down. |
| **ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG** | IntegerSerializer.class.getName() | 1. As key travels over network and so needs to be serialized. |
| **ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG** | StringSerializer.class.getName() | 1. As value travels over network and so needs to be serialized. |

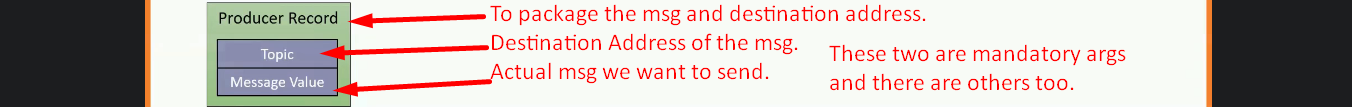
1. A picture containing text

   Description automatically generated
2. Kafka Producer API comes ready-to-use with Serializer classes for key/value in MSg.
3. **Step 01**: **Kafka Producer Configuration**  
   A picture containing text

   Description automatically generated  
   **Step 02**: **Create Producer**.  
     
   **Step 03**: **Sending Msgs**.  
   

**Step 04: Closing Producer**

# 31. Producer Record

1. producer.send(new ProducerRecord<>(AppConfigs.topicName, key, msg);  
   A red text on a white background

   Description automatically generated

# 32. Producer Serializer

1. When we hand over a msg to Producer, the msgs goes through Serialization (String, Int, Avro, Thrift), Partition and eventually buffering.  
   A screenshot of a computer

   Description automatically generated

# 33. Producer Partitioner

1. As a topic is divided into partitions, so Producer needs to decide which Partition.
   1. Setting the partition# in the ProducerRecord.
   2. You can supply your own partitioner that implements your desired partitioning strategy and assigns **partition# to each msg at runtime**.  
      
   3. **Default Partitioner**:
      1. **Hash Key Partitioning**: If key present, hashing function on Key to number then the number is module total partitions to reduce hash# to partition# and also ensuring that all msgs with same key, go to same partition# but if the # of partitions changes, the default partitioner will start giving different partition# for same key. If key=null, then round-robin to achieve equal distribution.
      2. **Round-Robin Partitioning**: When key=null.

# 34. Message Timestamp

1. ProducerRecord**.timestamp**: optional. By default, set by Kafka.
2. **Kafka allows you to implement one of** these two types of msg timestamping mechanisms.
   1. **Create Time**: When msg is produced.
   2. **Log Append Time**: When msg is received at kafka Server side.

**NOTE**: We can’t use both. While creating the topic, our app should decide b/w the two timestamping methods.

1. Set **message.timestamp.type** = 0 which is topic configuration for Create time. Either you set else Producer will set.  
   Set **message.timestamp.type** = 1 for using Log Append Time. Default is 1.

# 35. Producer Message Buffer

1. Partition-wise Buffer space by Producer.
   1. **Asynchronous Call**: Producer stores the msg and returns immediately.
   2. **Network Optimization**: I/o Thread run by Producer will collect all the msgs from the buffer for better throughput.
2. If Buffer is full, Producer waits till TimeoutException.
3. Producer Buffer memory = 32 MB 🡺 **Kafka\_Configuration** **buffer.memory=32**

# 36. Producer IO Thread and Retries

1. Retry is performed by Producer I/O Threads in the background.
2. Retry is performed when not receiving acknowledgement or receiving error.   
   **Kafka\_Configuration** 🡺 properties.setProperty(**ProducerConfig**.**RETRIES\_CONFIG**, Integer.toString(Integer.MAX\_**VALUE**));

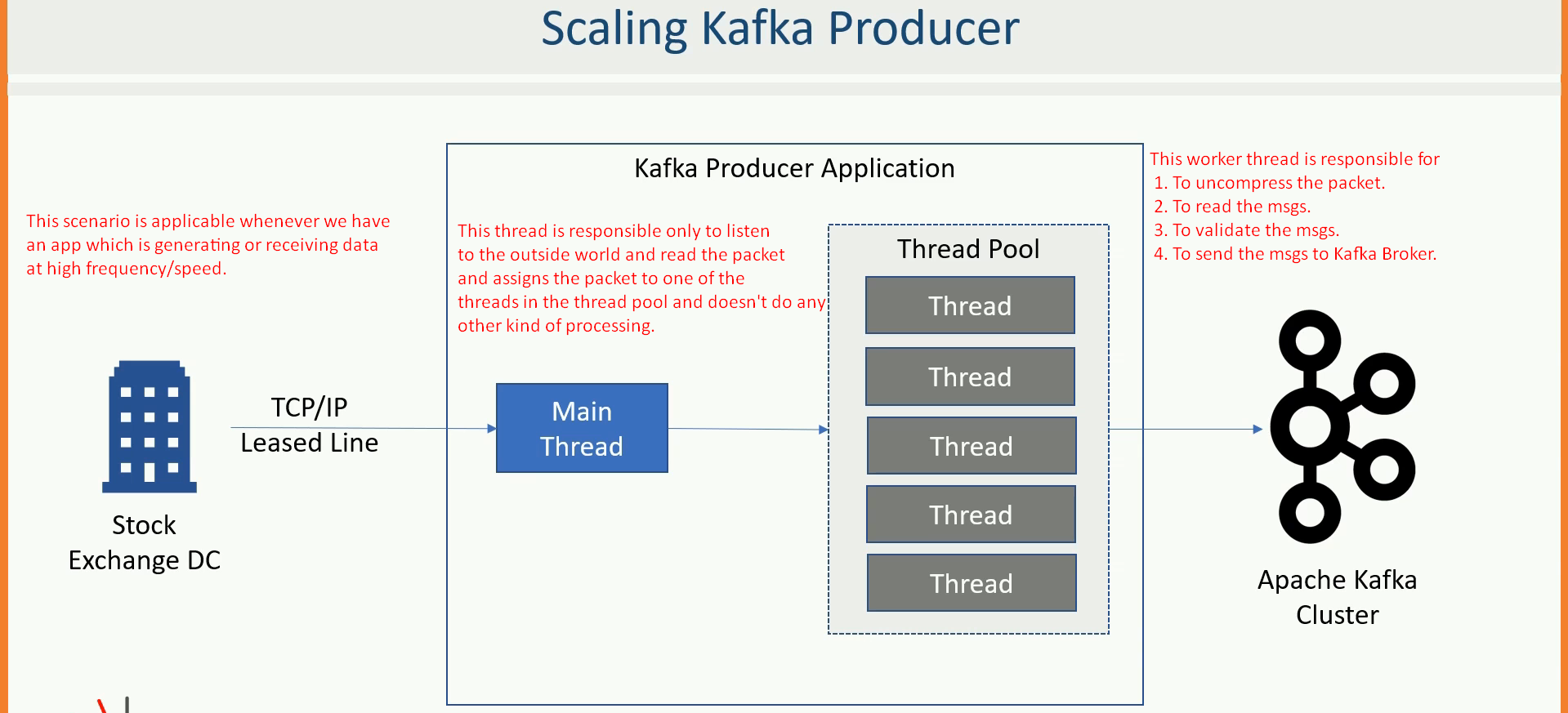
# 37. Summarizing Producer

1. Just Summary of this session.

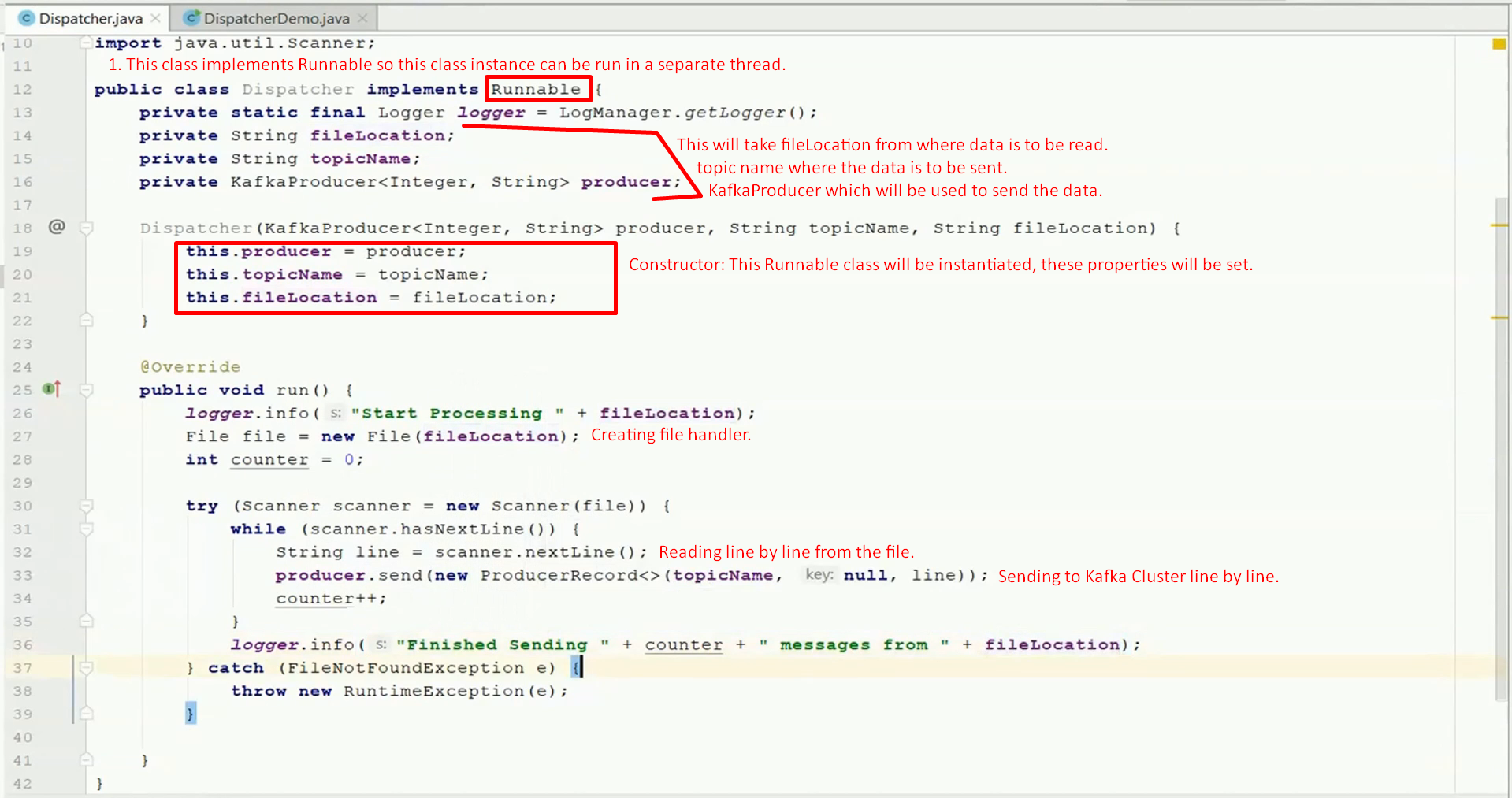
# 38. Horizontal Vs. Vertical Scalability

1. Two ways of scaling.
   1. By adding more Brokers.
   2. By adding more producer threads.

# 39. Producer Multi-Threading Scenario

1. 

# 40. Creating Multi-Threaded Kafka Producer

1. We have two files so two threads. Each reading from its own file and both threads will use same KafkaProducer and same topic.
2. 

# 41. Atleast Once Vs. At Most Once

1. **Remember: Retry is the reason for Duplicate msgs on the broker side.**
2. **At Least Once Semantics**: Due to not receiving the acknowledgement from the Leader broker (network issue or after storing the msg, leader broker goes down), the producer will retry. This is **default behavio**r.
3. **At Most Once Semantics**: By configuring Retries=0 but we may lose msg.

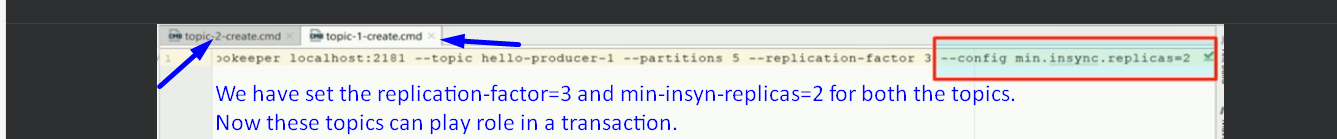
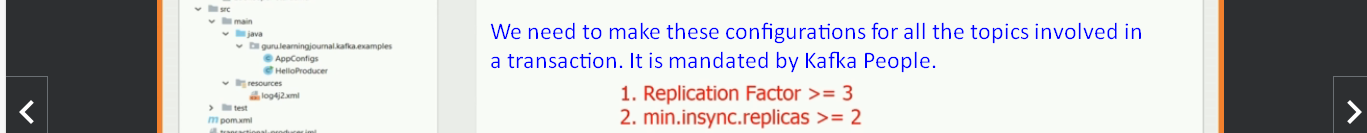
# 42. Exactly Once - Producer Idempotence

1. **Exactly Once Semantics:** No loss nor duplicate.
2. To meet **Exactly Once Requirement**, Kafka offers an **idempotent producer configuration**.
3. **Kafka\_Configuration 🡺 enable.idempotence=true**
4. **Remember: Producer generates a seq# within a specific topic partition.**A picture containing timeline

   Description automatically generated

# 43. Transactions in Kafka Producer

1. Transactional Producer goes one step ahead of **Idempotent Producer**.
2. That is an ability to write **to several partitions atomically**.
3. Diagram

   Description automatically generated
4. A screenshot of a computer code

   Description automatically generatedA close-up of a white background

   Description automatically generated
5. Steps
   1. 1: **KafkaProducer.initTransactions()**;Performs step to ensure if any previous transaction started by previous died instance is either completed or aborted also retrieves the Producer ID to send msgs as Transaction is by default idempotent.
   2. 2: Wrap all the send() API calls within the **KafkaProducer.beginTransaction() and KafkaProducer.commitTransaction()**;
6. **Complete code**:  
   Text

   Description automatically generated with medium confidence  
   Aborting:   
   Graphical user interface, text, application

   Description automatically generated