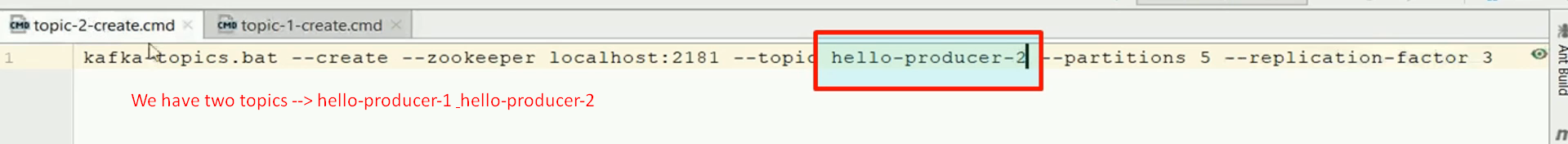
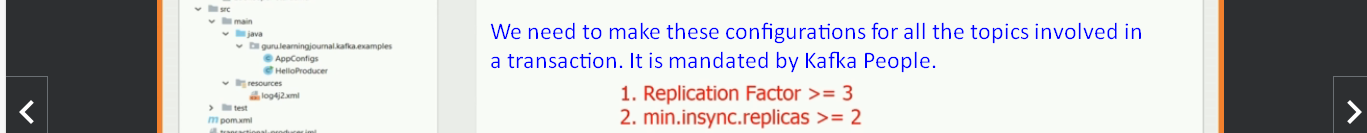
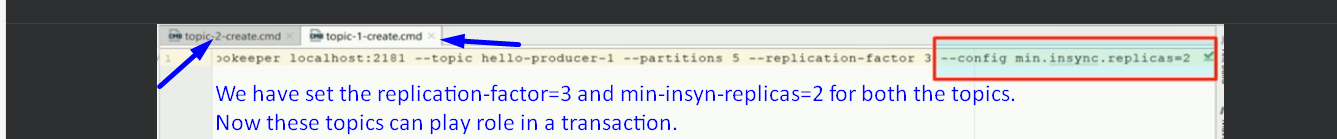
1. Text

   Description automatically generated 
2. In the earlier lecture, we learnt about the idempotent producer.  
   There is advaned producer concept.  
   We call it as **Transactional Producer**.
3. 
4. The **Transactional Producer** goes one step ahead of **idempotent Producer** and provides the **transactional guarantees**.  
   That is an ability to write to several partitions atomically.  
   The atomicity has the same meaning here as in DB that is either all the msgs in the same transaction is committed or none of them are saved.
5. Diagram

   Description automatically generated
6. Let’s create an example to understand the implementation details of a transaction in Kafka Producer.
   1. A screenshot of a computer

      Description automatically generated
   2. 
   3. We’re going to create a transaction that would send msgs to both the topics.
   4. When we commit the transaction, the msgs will be delivered to these topics.
   5. If we abort or rollback, the msgs should not be sent to any of these topics.  
      That is what atomicity means.
   6. Implementing transactions requires **some mandatory topic level configurations**.
   7.   
      Let’s do the above in the script creation script below.
   8.   
      Great!!! We’re all set from the topic perspective.
   9. Now these two topics can participate in a transaction.  
      A screenshot of a computer code

      Description automatically generated
   10. To create a Transactional Producer, it is mandatory to set **ProducerConfig.TRANSACTION\_ID\_CONFIG** when creating KafkaProducer.
   11. Code can be found along with the notes named 🡺 04-transactional-producer-completed.zip or you can download from this lecture.
   12. Here there are two things to remember.  
       A close-up of a white background

       Description automatically generated
       1. When we set TRANSACTIONAL\_ID\_CONFIG, idempotent is automatically enabled as transaction depends on idempotent.
       2. TRANSACTIONAL\_ID\_CONFIG must be unique for each **KafkaProducer instance**.  
          It means we can’t run two or more instances of a Kafka Producer with same transaction id.  
          If we do so, one of those transactions will be aborted as two instances of the same transaction are illegal.  
          Actually, the **primary purpose of the transaction id** is to rollback the older unfinished transactions for the same transaction id in case of producer app bounces or restarts.  
          A picture containing shape

          Description automatically generated  
          Then you may be wondering how to scale?  
          I mean how do I run multiple instances of a producer to achieve horizontal scalability.  
          Well that is simple.  
          Each instance (Kafka Producer) can set its own unique transaction id and all of those instances will be sending data to the same topic in a transaction with its own transaction id.  
          A screenshot of a computer

          Description automatically generated  
          This should be the case anyway. Two customers performing two parallel transactions should have two unique transaction ids.   
          Text

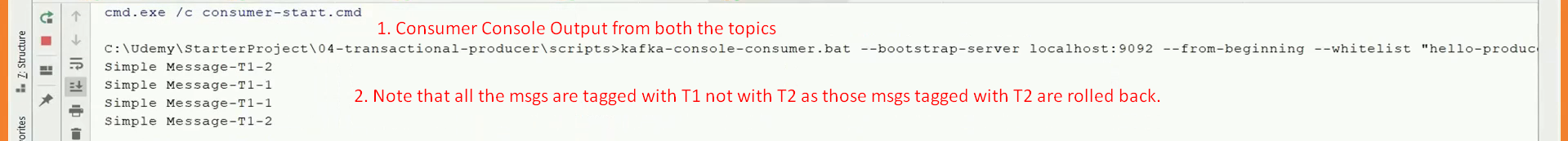
          Description automatically generated
   13. Implementing transaction in a Producer is 3-step process.
   14. **Step 01**: Initialize a transaction by calling **KafkaProducer.initTransactions()**;  
       This method performs the necessary checks to ensure such as any other transaction initiated by previous instances of the same producer is closed (when a producer restarts with same producer id).  
       That means if the previous producer app dies, the next instance guarantees that any unfinished transaction is either completed or aborted leaving the new instance in a clean state before resuming the work.  
       It also retrieves an internal producer id (as we discussed in Idempotent lecture) which will be used in all future msgs sent by this producer instance. This Producer ID is used by the Broker to implement Idempotent (along with msg seq#).  
       
   15. **Step 02**: Wrap all the send() API calls within the **KafkaProducer.beginTransaction() and KafkaProducer.commitTransaction()**;  
       All the msgs sent within the **KafkaProducer.beginTransaction() and KafkaProducer.commitTransaction()** will be part of the single transaction.  
       Text

       Description automatically generated with medium confidence  
       Let’s add code for rollback transaction along with the above code.   
       Graphical user interface, text, application

       Description automatically generated  
         
       One more info before we run the code:  
       Graphical user interface, text, application, email

       Description automatically generatedThe 4 msgs sent from Line#49,50 will be rolled back.
   16. Graphical user interface, text

       Description automatically generated Graphical user interface, text, application, email

       Description automatically generated
   17. 
7. One final note about the transaction.
8. The below snapshot says that the next transaction should start when the previous one is either committed or aborted. Other case is not allowed.  
   Graphical user interface, text, application

   Description automatically generated
9. In multi-threaded producer, you will call send() from different threads.  
   So, before you run any thread, create transaction then start all the threads then when all the threads are done, commit or rollback the transaction.  
   Like if main() is running 3 threads then main will begin transaction and will run the 3 threads and all the threads will join the main() and when 3 threads are done, the main thread will either commit or rollback the transaction.