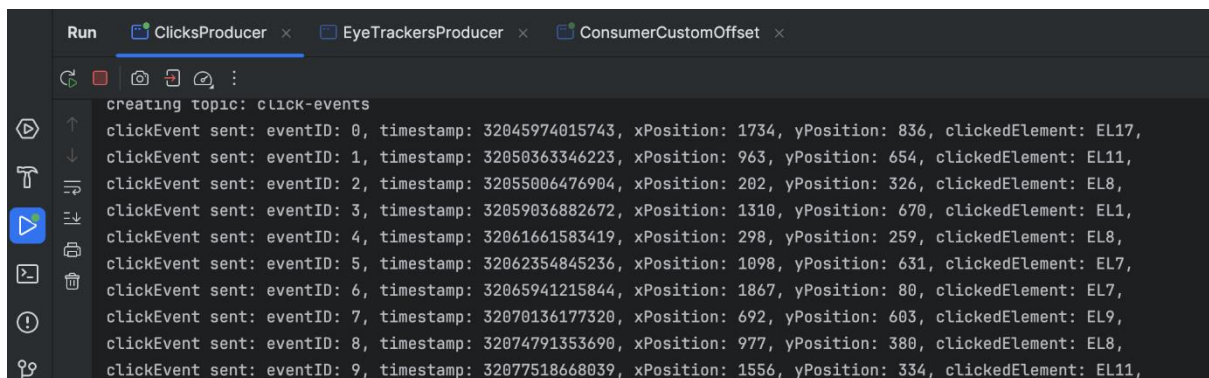


## Exercise 1 Kafka – Getting Started

### Offset misconfigurations- Lab02Part2-kafka-EyeTracking

The goal of this observation was to figure out how to read from a topic when an offset was set.

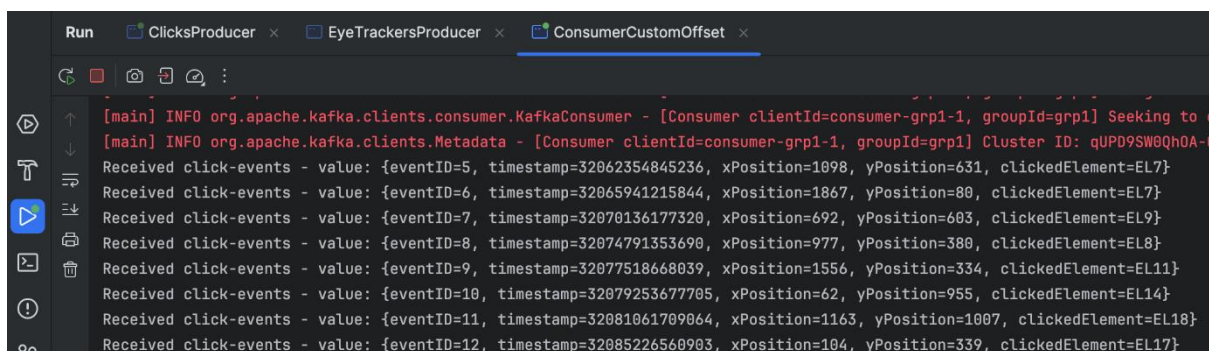
A producer is producing clicks as seen in Figure 1: Producer Output in a running process. In the consumer the instance is customized to read from offset 5 onwards. Please refer to class `ConsumerCustomOffset.java` in order to see this. In Figure 2: Consumer Output it is clearly recognizable that only events with eventid above 5 will be printed out.



```

Run ClicksProducer x EyeTrackersProducer x ConsumerCustomOffset x
creating topic: click-events
clickEvent sent: eventID: 0, timestamp: 32045974015743, xPosition: 1734, yPosition: 836, clickedElement: EL17,
clickEvent sent: eventID: 1, timestamp: 32050363346223, xPosition: 963, yPosition: 654, clickedElement: EL11,
clickEvent sent: eventID: 2, timestamp: 32055006476904, xPosition: 202, yPosition: 326, clickedElement: EL8,
clickEvent sent: eventID: 3, timestamp: 32059036882672, xPosition: 1310, yPosition: 670, clickedElement: EL1,
clickEvent sent: eventID: 4, timestamp: 32061661583419, xPosition: 298, yPosition: 259, clickedElement: EL8,
clickEvent sent: eventID: 5, timestamp: 32062354845236, xPosition: 1098, yPosition: 631, clickedElement: EL7,
clickEvent sent: eventID: 6, timestamp: 32065941215844, xPosition: 1867, yPosition: 80, clickedElement: EL7,
clickEvent sent: eventID: 7, timestamp: 32070136177320, xPosition: 692, yPosition: 603, clickedElement: EL9,
clickEvent sent: eventID: 8, timestamp: 32074791353690, xPosition: 977, yPosition: 380, clickedElement: EL8,
clickEvent sent: eventID: 9, timestamp: 32077518668039, xPosition: 1556, yPosition: 334, clickedElement: EL11,
  
```

Figure 1: Producer Output



```

Run ClicksProducer x EyeTrackersProducer x ConsumerCustomOffset x
[main] INFO org.apache.kafka.clients.consumer.KafkaConsumer - [Consumer clientId=consumer-grp1-1, groupId=grp1] Seeking to offset 5
[main] INFO org.apache.kafka.clients.Metadata - [Consumer clientId=consumer-grp1-1, groupId=grp1] Cluster ID: qUPD9SW0qh0A-0
Received click-events - value: {eventID=5, timestamp=32062354845236, xPosition=1098, yPosition=631, clickedElement=EL7}
Received click-events - value: {eventID=6, timestamp=32065941215844, xPosition=1867, yPosition=80, clickedElement=EL7}
Received click-events - value: {eventID=7, timestamp=32070136177320, xPosition=692, yPosition=603, clickedElement=EL9}
Received click-events - value: {eventID=8, timestamp=32074791353690, xPosition=977, yPosition=380, clickedElement=EL8}
Received click-events - value: {eventID=9, timestamp=32077518668039, xPosition=1556, yPosition=334, clickedElement=EL11}
Received click-events - value: {eventID=10, timestamp=32079253677705, xPosition=62, yPosition=955, clickedElement=EL14}
Received click-events - value: {eventID=11, timestamp=32081061709064, xPosition=1163, yPosition=1007, clickedElement=EL18}
Received click-events - value: {eventID=12, timestamp=32085226560903, xPosition=104, yPosition=339, clickedElement=EL17}
  
```

Figure 2: Consumer Output

Please note that the consumer service should only be started after the producer has already produced more than 5 (as defined with the offset) events. Otherwise, the consumer will also consume the events below the defined offset. This is something that was not expected and due to further investigation, the reason could be found. According to the description of the poll method: “On each poll, consumer will try to use the last consumed offset as the starting offset and fetch sequentially. The last consumed offset can be manually set through `seek(TopicPartition, long)` or automatically set as the last committed offset for the subscribed list of partitions”<sup>1</sup> Since there is no last consumed offset when the consumer

<sup>1</sup> <https://kafka.apache.org/26/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#poll-java.time.Duration->

service is started before the producer service, it will probably just be set to 0 and that might be the reason why the consumer gets all the events below the offset.

### Negative Offset

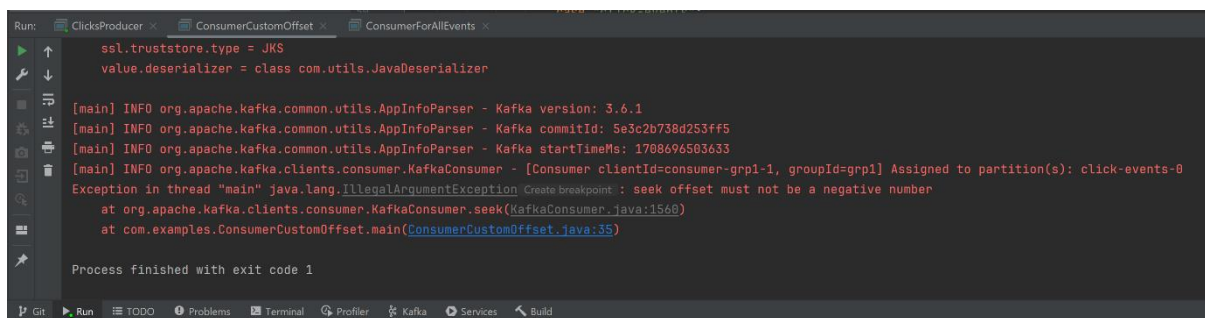
The goal of this experiment was to figure out Kafka's behavior with a negative offset boundary. Intuitively, one could assume either the consumer skips to the lowest valid offset, e.g. zero, or that this is an invalid case leading to an error in the system.

### Setup

While the Producer is producing events (in this case, the click producer, same as can be seen in Figure 1), the manually set offset in `ConsumerCustomOffset.java` is set to `-5`, while the rest of the implementation remains the same.

### Result

The experiment shows us that Kafka cannot handle negative offset numbers. Trying to use the `seek` method of the consumer class with a negative offset leads to an `IllegalArgumentException` (Figure 3):



```
Run: ClicksProducer x ConsumerCustomOffset x ConsumerForAllEvents x
ssl.truststore.type = JKS
value.deserializer = class com.utils.JavaDeserializer

[main] INFO org.apache.kafka.common.utils.AppInfoParser - Kafka version: 3.6.1
[main] INFO org.apache.kafka.common.utils.AppInfoParser - Kafka commitId: 5e3c2b738d253ff5
[main] INFO org.apache.kafka.common.utils.AppInfoParser - Kafka startTimeMs: 1708696503633
[main] INFO org.apache.kafka.clients.consumer.KafkaConsumer - [Consumer clientId=consumer-grp1-1, groupId=grp1] Assigned to partition(s): click-events-0
Exception in thread "main" java.lang.IllegalArgumentException: seek offset must not be a negative number
    at org.apache.kafka.clients.consumer.KafkaConsumer.seek(KafkaConsumer.java:1569)
    at com.examples.ConsumerCustomOffset.main(ConsumerCustomOffset.java:35)

Process finished with exit code 1
```

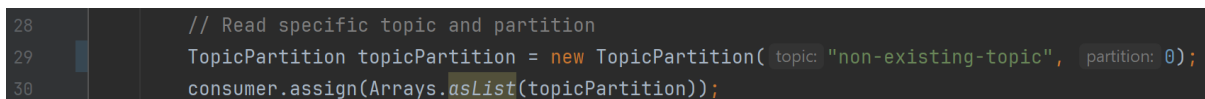
Figure 3: *IllegalArgumentException with negative offset*

### Subscribing to a non-existent topic

We conducted an experiment to find out how Kafka deals with consumers that attempt to subscribe to non-existing topics. Our expectations were that Kafka simply ignores the consumer or notifies it that no such topic was found.

### Setup

We adjusted the `ConsumerForGazeEventsForSingleEyeTracker.java` file, so that the consumer subscribed to a topic that was not set up beforehand:



```
28 // Read specific topic and partition
29 TopicPartition topicPartition = new TopicPartition( topic: "non-existing-topic", partition: 0);
30 consumer.assign(Arrays.asList(topicPartition));
```

Figure 4: *subscribing to a non-existing topic*

The rest of the file was not modified.

## Result

Running the code leads to some interesting results. First, the code runs with no problems and does not get notified, that the topic it's trying to subscribe to does not exist:

```
[main] INFO org.apache.kafka.common.utils.AppInfoParser - Kafka version: 3.6.1
[main] INFO org.apache.kafka.common.utils.AppInfoParser - Kafka commitId: 5e3c2b738d253ff5
[main] INFO org.apache.kafka.common.utils.AppInfoParser - Kafka startTimeMs: 1788698125584
[main] INFO org.apache.kafka.clients.consumer.KafkaConsumer - [Consumer clientId=consumer-grp1-1, groupId=grp1] Assigned to partition(s): non-existing-topic-0
[main] INFO org.apache.kafka.clients.Metadata - [Consumer clientId=consumer-grp1-1, groupId=grp1] Cluster ID: 6WfoYSeKRs6strSYMS58Jg
[main] INFO org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - [Consumer clientId=consumer-grp1-1, groupId=grp1] Discovered group coordinator localhost:9092
[main] INFO org.apache.kafka.clients.consumer.internals.ConsumerCoordinator - [Consumer clientId=consumer-grp1-1, groupId=grp1] Setting offset for partition non-existing-top
[main] INFO org.apache.kafka.clients.NetworkClient - [Consumer clientId=consumer-grp1-1, groupId=grp1] Node -1 disconnected.
```

Figure 5: Output result of subscribing to a non-existing topic

Contrary to our assumption, we can see that Kafka has no problem with topics that don't exist, it simply initiates a new topic once a consumer asks for it:

Topics		Partitions	Configuration				
		Partition ID	Message Count	Start Offset	End Offset	Leader	Replicas
	click-events						
	gaze-events						
	non-existing-topic	0	0	0	0	1001	1001
	Consumer Groups						
	grp1						

Figure 6: Overview of Kafkas topics after running the experiment

Looking into the consumers, we can see that the consumer is registered and in the same group as specified:

Topics		Topic	Partition	Lag	Offset
	click-events	non-existing-topic	0	0	0
	gaze-events				
	non-existing-topic	click-events	0	972	206
	Consumer Groups				
	grp1				

Figure 7: Consumer group after running the experiment

To investigate the setup of the created topic, we connected the gaze event producer to the new topic, and tried to send messages to the subscriber. We removed the manual deletion and creation of the topic from the beginning of the script and limited the receiving partition to the one created by the consumer. The producer was able to send messages through the new topic without any problems, and the consumer was able to read them all. We further attempted to create a new partition within the topic, by subscribing the consumer to the same topic but with additional non-existing partitions. This did not work, it seems that while

a consumer can create topics, Kafka uses the default topic configuration and only creates one partition in the topic. The consumer cannot create new partitions beyond the first partition.

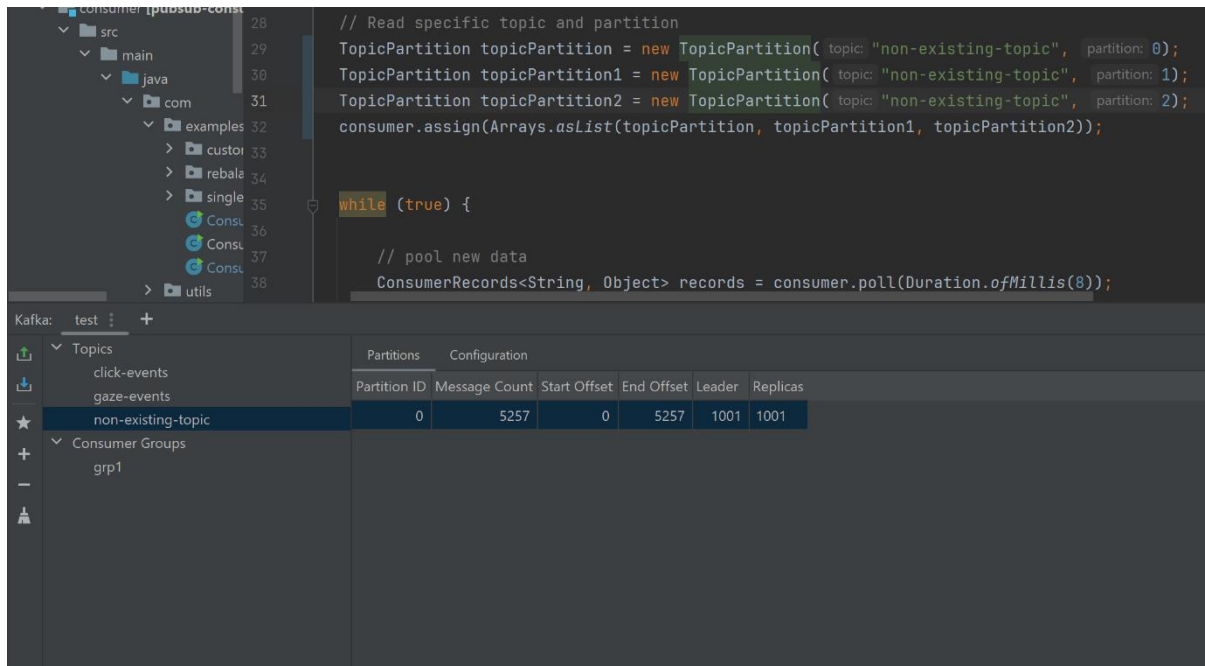


Figure 8: Setup and Result of attempting to create additional partitions through a consumer

This behavior could lead to problems: In a setup, where applications can subscribe to a public Kafka server, a malicious entity could take advantage of this setup and create millions of topics and partitions on the server, possibly in an attempt to increase the software overhead, the latency or even in an DDOS style attack to shut down the server. After a quick research, it appears that Kafka is aware of the behavior, and offers an option to disable automatic creation of topics by consumers via the broker configuration “auto.create.topics.enable: false”. The default setup is true, thus enabling users to create topics.<sup>2</sup>

### Attempting to Overwhelm a Kafka Server with Topics

As a result of our previous discovery, we take the next step and try to test out how insecure Kafkas default setting really is. While creating inactive topics with a single partition seems innocent, it could be that the server reaches its breaking point after a while, or, if it doesn't lead to the server breaking, that the responsiveness suffers. This could be considered a security vulnerability of the type “insecure default”.

### Setup

In our first experiment setup, we wrote a very simple script that endlessly subscribes to a new, non-existing Kafka topic, and polls it for new events (see figure 9).

<sup>2</sup> <https://cwiki.apache.org/confluence/display/KAFKA/KIP-361%3A+Add+Consumer+Configuration+to+Disable+Auto+Topic+Creation>

```

18 public static void main(String[] args) throws IOException, ParseException {
19
20     KafkaConsumer<String, Object> consumer;
21     try (InputStream props = Resources.getResource("consumer.properties").openStream()) {
22         Properties properties = new Properties();
23         properties.load(props);
24         consumer = new KafkaConsumer<>(properties);
25     }
26
27     long newTopicCounter = 0;
28
29     while (true) {
30         TopicPartition topicPartition = new TopicPartition("non-existing-topic-" + newTopicCounter, 0);
31         consumer.assign(Arrays.asList(topicPartition));
32         ConsumerRecords<String, Object> records = consumer.poll(Duration.ofMillis(8));
33         newTopicCounter++;
34     }
35 }
36
37 }

```

Figure 9: script for our first stress-test experiment setup

## Result

After running this setup for about 10 minutes, the memory and CPU usage seems to have “stabilized”. When looking at the chart in figure 10, we can see that, while the CPU usage jumps around quite a bit, it never even comes close to maxing out, with the highest peaks just reaching 500% of available processing power. Interestingly, figure 11 tells us that after the expected rapid memory usage growth in the beginning of the experiment, it stabilizes after a while, remaining at around 2.4 GB usage for 7 minutes before the end of the experiment. While looking at the number of topics and partition on the server, it seems that there were topics being created during those 7 minutes of stability, leading to the question how Kafka manages to throttle the memory and CPU usage so effectively.

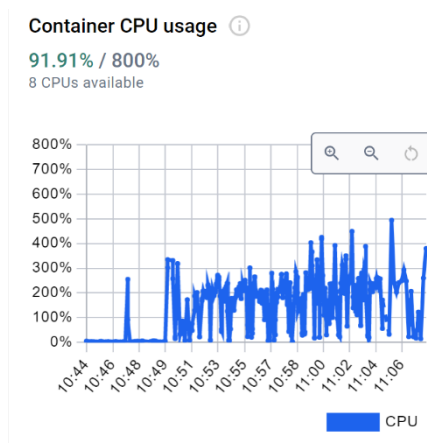


Figure 10: CPU usage of our Kafka container during the stress-test

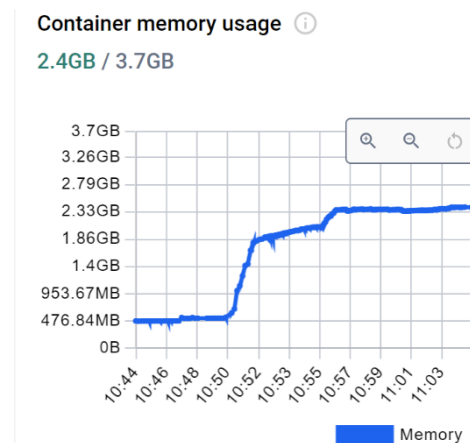


Figure 11: Memory usage of our Kafka container during the stress-test

After some research, we could not find a definite answer as to how and why this works, but some possible explanations could be the assigned heap size of the JVM Kafka is running on, effective compression or management of the file system being used to distinguish topics or

simply a built-in safety feature from either Kafka or Docker itself. The results are not so conclusive as to say that a more complex “attack” script could not bring down an unsecured Kafka server, but we were impressed by the effectiveness and stability Kafka has shown in this experiment.