Kafka Producer Advanced

Objectives Create Producer

Tos

- Cover advanced topics regarding Java Kafka Producer
- Custom Serializers
- Custom Partitioners
- Batching
- Compression
- Retries and Timeouts
- Message serialization using Avro

Kafka Producer

Tos

- Kafka client that publishes records to Kafka cluster
- Thread safe
- Producer has pool of buffer that holds to-be-sent records
  - background I/O threads turning records into request bytes and transmit requests to Kafka
- Close producer so producer will not leak resources

Kafka Producer Send, Acks and Buffers

Tos

- ***send***() method is asynchronous
  - adds the record to output buffer and return right away
  - buffer used to batch records for efficiency IO and compression
- acks config controls Producer record durability. ”all" setting ensures full commit of record, and is most durable and least fast setting
- Producer can retry failed requests
- Producer has buffers of unsent records per topic partition (sized at ***batch.size***)

Kafka Producer: Buffering and batching

Tos

- Kafka Producer buffers are available to send immediately as fast as broker can keep up (limited by inflight ***max.in.flight.requests.per.connection***)
- To reduce requests count, set ***linger.ms*** > 0
  - wait up to linger.ms before sending or until batch fills up whichever comes first
  - Under heavy load linger.ms not met, under light producer load used to increase broker IO throughput and increase compression
- ***buffer.memory*** controls total memory available to producer for buffering
  - If records sent faster than they can be transmitted to Kafka then this buffer gets exceeded then additional send calls block. If period blocks (**max.block.ms**) after then Producer throws a TimeoutException

Producer Acks

Tos

- Producer Config property ***acks***
  - ***(default all)***
- Write Acknowledgment received count required from partition leader before write request deemed complete
- Controls ***Producer*** sent records durability
- Can be all (-1), none (0), or leader (1)

Acks 0 (NONE)

Tos

- acks=0
- Producer does not wait for any ack from broker at all
- Records added to the socket buffer are considered sent
- No guarantees of durability - maybe
- Record Offset returned is set to -1 (unknown)
- Record loss if leader is down
- Use Case: maybe log aggregation

Acks 1 (LEADER)

Tos

- acks=1
- Partition leader wrote record to its local log but responds without followers confirmed writes
- If leader fails right after sending ack, record could be lost
  - Followers might have not replicated the record
- Record loss is rare but possible
- Use Case: log aggregation

Acks -1 (ALL)

Tos

- acks=all or acks=-1
- Leader gets write confirmation from full set of ISRs before sending ack to producer
- Guarantees record not be lost as long as one ISR remains alive
- Strongest available guarantee
- Even stronger with broker setting ***min.insync.replicas*** (specifies the minimum number of ISRs that must acknowledge a write)
- Most Use Cases will use this and set a ***min.insync.replicas > 1***

Kafka Producer config Acks

Tos

StockPriceKafkaProducer.java

```
StockPriceKafkaProducer
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Stc
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27
28          //Set number of acknowledgments - acks - default is all
29          props.put(ProducerConfig.ACKS_CONFIG, "all");
30
31          return new KafkaProducer<>(props);
32      }
```

Producer Buffer Memory Size

Tos

- Producer config property: ***buffer.memory***
  - ***default 32MB***
- Total memory (bytes) producer can use to buffer records to be sent to broker
- Producer blocks up to ***max.block.ms*** if ***buffer.memory*** is exceeded
  - if it is sending faster than the broker can receive, exception is thrown

Batching by Size

Tos

- Producer config property: ***batch.size in*** total bytes
  - Default 16K
- Producer batch records
  - fewer requests for multiple records sent to same partition
  - Improved IO throughput and performance on both producer and server
- If record is larger than the batch size, it will not be batched
- Producer sends requests containing multiple batches
  - batch per partition
- Small batch size reduce throughput and performance. If batch size is too big, memory allocated for batch is wasted

Batching by Time and Size - 1

Tos

- ***Producer*** config property: ***linger.ms***
  - Default 0
- Producer groups together any records that arrive before they can be sent into a batch
  - good if records arrive faster than they can be sent out
- ***Producer*** can reduce requests count even under moderate load using ***linger.ms***

Batching by Time and Size - 2

Tos

- ***linger.ms*** adds delay to wait for more records to build up so larger batches are sent
  - ***good brokers throughput*** at ***cost of producer latency***
- If ***producer*** gets records who size is ***batch.size*** or more for a broker's leader partitions, then it is sent right away
- If ***Producers*** gets less than ***batch.size*** but ***linger.ms*** interval has passed, then records for that partition are sent
- Increase to improve throughput of Brokers and reduce broker load (common improvement)

Compressing Batches

Tos

- ***Producer*** config property: ***compression.type***
  - Default 0
- Producer compresses request data
- By default producer does not compress
- Can be set to none, gzip, snappy, or lz4
- Compression is by batch
  - improves with larger batch sizes
- End to end compression possible if Broker config “compression.type” set to producer. Compressed data from producer sent to log and consumer by broker

Batching and Compression

Tos

```
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
```

```
57
58      private static void setupBatchingAndCompression(Properties props) {
59          //Wait up to 50 ms to batch to Kafka - linger.ms
60          props.put(ProducerConfig.LINGER_MS_CONFIG, 200);
61
62          //Holds up to 64K per partition default is 16K - batch.size
63          props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);
64
65          //Holds up to 64 MB default is 32MB for all partition buffers
66          // - "buffer.memory"
67          props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33_554_432 * 2);
68
69          //Set compression type to snappy - compression.type
70          props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
71      }
```

Custom Serializers

Tos

- You don’t have to use built in serializers
- You can write your own
- Just need to be able to convert to/fro a byte[]
- Serializers work for keys and values
- ***value.serializer*** and ***key.serializer***

Custom Serializers Config

Tos

```
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19
```

```
30
31      private static void setupBootstrapAndSerializers(Properties props) {
32          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
33                  StockAppConstants.BOOTSTRAP_SERVERS);
34          props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
35          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
36                  StringSerializer.class.getName());
37
38          //Custom Serializer - config "value.serializer"
39          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
40                  StockPriceSerializer.class.getName());
41      }
```

Custom Serializer

Tos

StockPriceSerializer.java

```java
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Serializer;

import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockPriceSerializer implements Serializer<StockPrice> {

    @Override
    public byte[] serialize(String topic, StockPrice data) {
        return data.toJson().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

Stock Price

Tos

StockPrice.java

```java
package com.cloudurable.kafka.producer.model;

import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;

    public String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }

```

Broker Follower Write Timeout

Tos

- ***Producer*** config property: ***request.timeout.ms***
  - Default 30 seconds (30,000 ms)
- Maximum time broker waits for confirmation from followers to meet Producer acknowledgment requirements for ***ack=all***
- Measure of broker to broker latency of request
- 30 seconds is high, long process time is indicative of problems

Producer Retries

Tos

- Producer config property: ***retries***
  - ***Default 0***
- Retry count if ***Producer*** does not get ack from Broker
  - only if record send fail deemed a transient error (API)
  - as if your producer code resent record on failed attempt
- timeouts are retried, ***retry.backoff.ms*** (default to 100 ms) to wait after failure before ***retry***

Retry, Timeout, Back-off Example

Tos

```
13
14  public class StockPriceKafkaProducer {
15
16      private static Producer<String, StockPrice> createProducer() {
17          final Properties props = new Properties();
18          setupBootstrapAndSerializers(props);
19          setupBatchingAndCompression(props);
20          setupRetriesInFlightTimeout(props);
```

```
40
41      private static void setupRetriesInFlightTimeout(Properties props) {
42
43          //Only two in-flight messages per Kafka broker connection
44          // - max.in.flight.requests.per.connection (default 5)
45          props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
46                  1);
47
48          //Set the number of retries - retries
49          props.put(ProducerConfig.RETRIES_CONFIG, 2);
50
51          //Request timeout - request.timeout.ms
52          props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 15_000);
53
54          //Only retry after one second.
55          props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 1_000);
56      }
57
```

Producer Partitioning

Tos

- ***Producer*** config property: ***partitioner.class***
  - org.apache.kafka.clients.producer.internals.DefaultPartitioner
- Partitioner class implements Partitioner interface
- Default Partitioner partitions using hash of key if record has key
- Default Partitioner partitions uses round-robin if record has no key

Configuring Partitioner

Tos

StockPriceKafkaProducer.java

StockPriceKafkaProducer | createProducer()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install partitioner -- "partitioner.class"
28          props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
29                  StockPricePartitioner.class.getName());
30
31          props.put("importantStocks", "IBM,UBER");
```

StockPricePartitioner

Tos

StockPriceKafkaProducer.java × StockPricePartitioner.java ×

StockPricePartitioner StockPricePartitioner()

```
1   package com.cloudurable.kafka.producer;
2
3   import org.apache.kafka.clients.producer.Partitioner;
4   import org.apache.kafka.common.Cluster;
5   import org.apache.kafka.common.PartitionInfo;
6
7   import java.util.*;
8
9   public class StockPricePartitioner implements Partitioner{
10
11      private final Set<String> importantStocks;
12      public StockPricePartitioner() { importantStocks = new HashSet<>(); }
15
16      @Override
17      public int partition(final String topic,
18                           final Object objectKey,
19                           byte[] keyBytes, final Object value,
20                           final byte[] valueBytes,
21                           final Cluster cluster) {...}
38
39      @Override
40      public void close() {
41      }
42
43      @Override
44      public void configure(Map<String, ?> configs) {...}
48  }
```

StockPricePartitioner Partition()

Tos

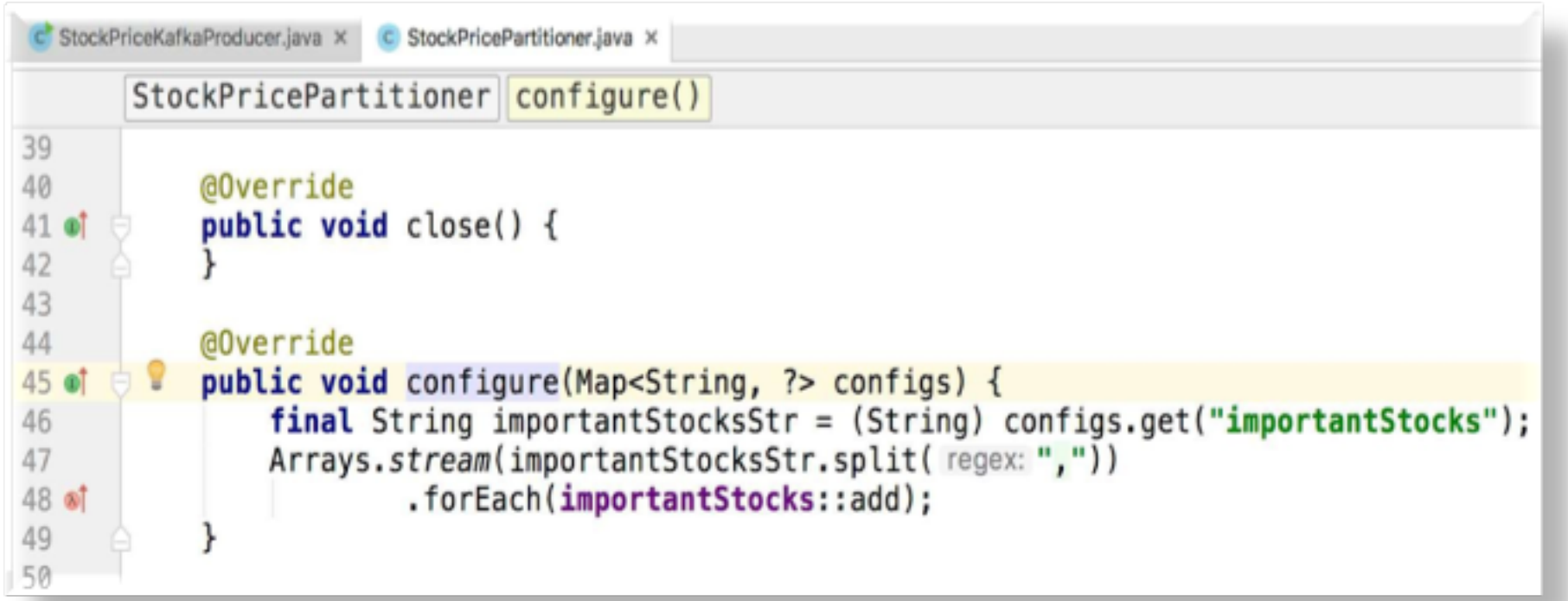
StockPriceKafkaProducer.java × StockPricePartitioner.java ×

StockPricePartitioner | partition()

```
17      public int partition(final String topic,
18                           final Object objectKey,
19                           final byte[] keyBytes,
20                           final Object value,
21                           final byte[] valueBytes,
22                           final Cluster cluster) {
23
24          final List<PartitionInfo> partitionInfoList =
25                  cluster.availablePartitionsForTopic(topic);
26          final int partitionCount = partitionInfoList.size();
27          final int importantPartition = partitionCount -1;
28          final int normalPartitionCount = partitionCount -1;
29
30          final String key = ((String) objectKey);
31
32          if (importantStocks.contains(key)) {
33              return importantPartition;
34          } else {
35              return Math.abs(key.hashCode()) % normalPartitionCount;
36          }
37
38      }
```

StockPricePartitioner

Tos



Producer Interception

Tos

- ***Producer*** config property: ***interceptor.classes***
  - empty (you can pass an comma delimited list)
- interceptors implementing ProducerInterceptor interface
- intercept records producer sent to broker and after acks
- you could mutate records

Kafka Producer - Interceptor

Tos

StockPriceKafkaProducer.java

StockPriceKafkaProducer | getStockSenderList()

```
17  public class StockPriceKafkaProducer {
18      private static final Logger logger = LoggerFactory.getLogger(Sto
19
20
21      private static Producer<String, StockPrice> createProducer() {
22          final Properties props = new Properties();
23          setupBootstrapAndSerializers(props);
24          setupBatchingAndCompression(props);
25          setupRetriesInFlightTimeout(props);
26
27          //Install interceptor list - config "interceptor.classes"
28          props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
29                  StockProducerInterceptor.class.getName());
30
31          //Set number of acknowledgments - acks - default is all
32          props.put(ProducerConfig.ACKS_CONFIG, "all");
33
34          return new KafkaProducer<>(props);
35      }
```

KafkaProducer ProducerInterceptor

Tos

StockProducerInterceptor.java

```
StockProducerInterceptor
10
11  public class StockProducerInterceptor implements ProducerInterceptor {
12
13      private final Logger logger = LoggerFactory
14              .getLogger(StockProducerInterceptor.class);
15      private int onSendCount;
16      private int onAckCount;
17
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {...}
36
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {...}
54
55      @Override
56      public void close() {...}
59
60      @Override
61      public void configure(Map<String, ?> configs) {...}
64  }
```

ProducerInterceptor onSend

Tos

StockProducerInterceptor.java

```
StockProducerInterceptor
18      @Override
19      public ProducerRecord onSend(final ProducerRecord record) {
20          onSendCount++;
21          if (logger.isDebugEnabled()) {
22              logger.debug(String.format("onSend topic=%s key=%s value=%s %d \n",
23                      record.topic(), record.key(), record.value().toString(),
24                      record.partition()
25              ));
26          } else {
27              if (onSendCount % 100 == 0) {
28                  logger.info(String.format("onSend topic=%s key=%s value=%s %d \n",
29                          record.topic(), record.key(), record.value().toString(),
30                          record.partition()
31                  ));
32              }
33          }
34          return record;
35      }
```

Output

ic=stock-prices2 key=UBER value=StockPrice{dollars=737, cents=78, name='

ProducerInterceptor onAck

Tos

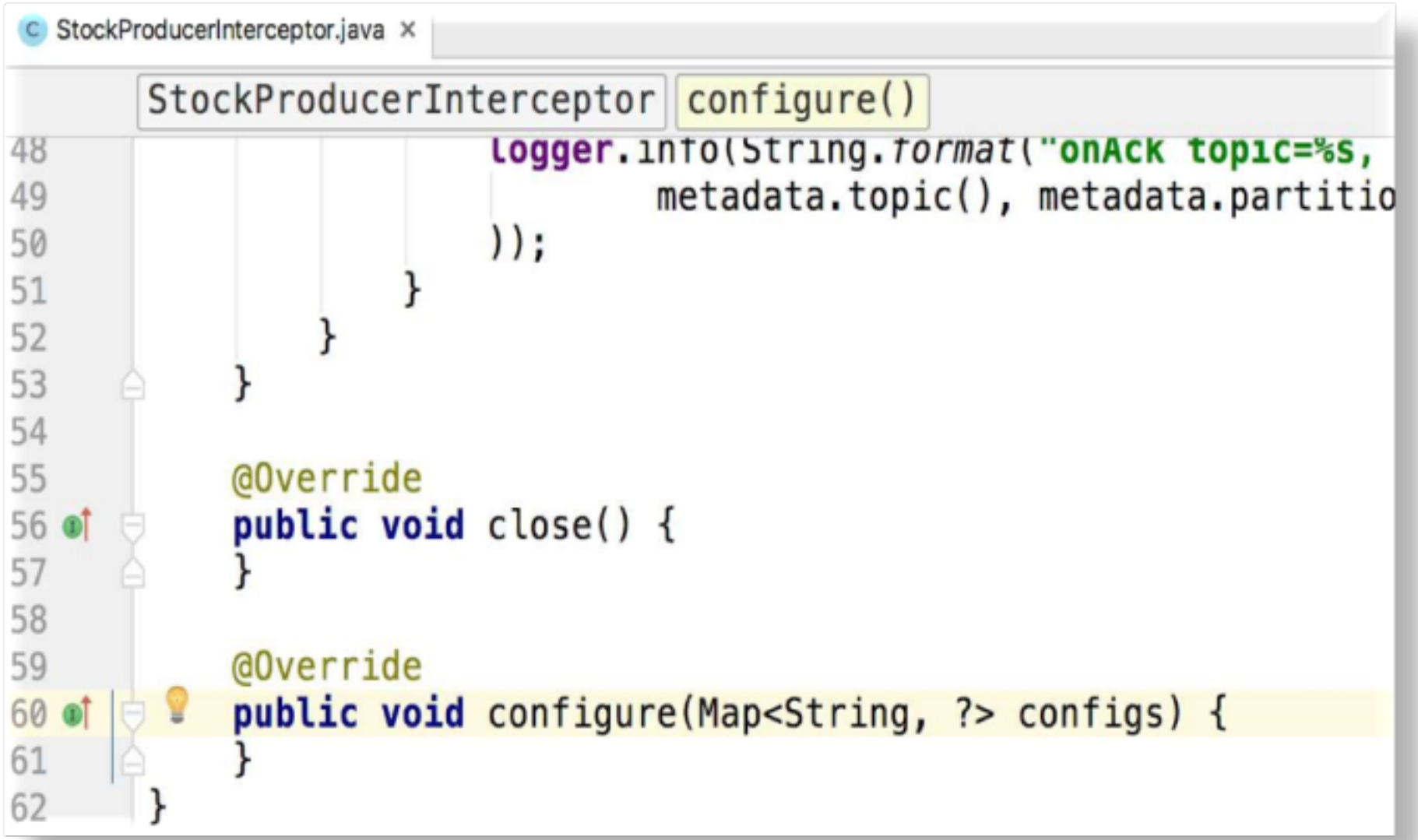
StockProducerInterceptor.java

StockProducerInterceptor | onAcknowledgement()

```
37      @Override
38      public void onAcknowledgement(final RecordMetadata metadata,
39                                    final Exception exception) {
40          onAckCount++;
41
42          if (logger.isDebugEnabled()) {
43              logger.debug(String.format("onAck topic=%s, part=%d, offset=%d\n",
44                      metadata.topic(), metadata.partition(), metadata.offset()
45              ));
46          } else {
47              if (onAckCount % 100 == 0) {
48                  logger.info(String.format("onAck topic=%s, part=%d, offset=%d\n",
49                          metadata.topic(), metadata.partition(), metadata.offset()
50                  ));
51              }
52          }
53      }
```

ProducerInterceptor the rest

Tos



KafkaProducer send() Method

Tos

- Two forms of send with callback and with no callback both return Future
  - Asynchronously sends a record to a topic
  - Callback gets invoked when send has been acknowledged.
- send is asynchronous and return right away as soon as record has added to send buffer
- Sending many records at once without blocking for response from Kafka broker
- Result of send is a RecordMetadata
  - record partition, record offset, record timestamp
- Callbacks for records sent to same partition are executed in order

KafkaProducer send() Exceptions

Tos

- ***InterruptException*** - If the thread is interrupted while blocked (API)
- ***SerializationException*** - If key or value are not valid objects given configured serializers (API)
- ***TimeoutException*** - If time taken for fetching metadata or allocating memory exceeds max.block.ms, or getting acks from Broker exceed timeout.ms, etc. (API)
- ***KafkaException*** - If Kafka error occurs not in public API. (API)

Using send method

Tos

StockSender.java

StockSender > run()

```
51              try {
52
53                  final Future<RecordMetadata> future = producer.send(record);
54
55                  if (sentCount % 100 == 0) {
56                      displayRecordMetaData(record, future);
```

StockSender.java

StockSender > displayRecordMetaData()

```
74      private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
75                                         final Future<RecordMetadata> future)
76                                 throws InterruptedException, ExecutionException {
77          final RecordMetadata recordMetadata = future.get();
78          logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
79                          "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
80                  record.key(),
81                  record.value().toJson(),
82                  recordMetadata.topic(),
83                  recordMetadata.partition(),
84                  recordMetadata.offset(),
85                  new Date(recordMetadata.timestamp())
86                  ));
87      }
```

KafkaProducer flush() method

Tos

flush() method sends all buffered records now (even if ***linger.ms*** > 0)

- blocks until requests complete

- Useful when consuming from some input system and pushing data into Kafka

- ***flush***() ensures all previously sent messages have been sent
  - you could mark progress as such at completion of flush

KafkaProducer close()

Tos

- close() closes producer
  - frees resources (threads and buffers) associated with producer
- Two forms of method
- both block until all previously sent requests complete or duration passed in as args is exceeded
- close with no params equivalent to close(Long.MAX_VALUE, TimeUnit.MILLISECONDS).
- If producer is unable to complete all requests before the timeout expires, all unsent requests fail, and this method fails

Orderly shutdown using close

Tos

StockPriceKafkaProducer.java

StockPriceKafkaProducer › main()

```
95        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97            executorService.shutdown();
98            try {
99                executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
100               logger.info("Shutting down executorService for workers nicely");
101           } catch (InterruptedException e) {
102               logger.warn("shutting down", e);
103           }
104
105           logger.info("Flushing producer");
106           producer.flush();
107           logger.info("Closing producer");
108           producer.close();
109
110           if (!executorService.isShutdown()) {
111               logger.info("Forcing shutdown of workers");
112               executorService.shutdownNow();
113           }
114       }));
```

Wait for clean close

Tos

StockPriceKafkaProducer.java

StockPriceKafkaProducer › main()

```
95        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
96
97            executorService.shutdown();
98            try {
99                executorService.awaitTermination( timeout: 200, TimeUnit.MILLISECONDS);
100               logger.info("Flushing and closing producer");
101               producer.flush();
102               producer.close( timeout: 10_000, TimeUnit.MILLISECONDS);
103           } catch (InterruptedException e) {
104               logger.warn("shutting down", e);
105           }
106
107       }));
```

KafkaProducer partitionsFor() method

Tos

- partitionsFor(topic) returns meta data for partitions
- **public** List<PartitionInfo> partitionsFor(String topic)
- Get partition metadata for give topic
- Produce that do their own partitioning would use this
  - for custom partitioning
- PartitionInfo(String topic, int partition, Node leader, Node[] replicas, Node[] inSyncReplicas)
  - Node(int id, String host, int port, optional String rack)

StockPrice Producer Java Example

StockPrice App to demo Advanced Producer

Tos

- ***StockPrice*** - holds a stock price has a name, dollar, and cents
- ***StockPriceKafkaProducer -*** Configures and creates ***KafkaProducer***<String, ***StockPrice***>, ***StockSender*** list, ThreadPool (ExecutorService), starts ***StockSender*** runnable into thread pool
- ***StockAppConstants*** - holds topic and broker list
- ***StockPriceSerializer -*** can serialize a ***StockPrice*** into ***byte[]***
- ***StockSender -*** generates somewhat random stock prices for a given ***StockPrice*** name, Runnable, 1 thread per StockSender
  - Shows using ***KafkaProducer*** from many threads

StockPrice domain object

Tos

StockPrice.java

```
StockPrice
1  package com.cloudurable.kafka.producer.model;
2
3  import io.advantageous.boon.json.JsonFactory;
4
5  public class StockPrice {
6
7      private final int dollars;
8      private final int cents;
9      private final String name;
10
11     public String toJson() {
12         return "{" +
13                 "\"dollars\": " + dollars +
14                 ", \"cents\": " + cents +
15                 ", \"name\": \"" + name + '\"' +
16                 '}';
17     }
18
```

- has name
- dollars
- cents
- converts itself to JSON

StockPriceKafkaProducer

Tos

- Import classes and setup logger
- Create ***createProducer*** method to create ***KafkaProducer*** instance
- Create ***setupBootstrapAndSerializers*** to initialize bootstrap servers, client id, key serializer and custom serializer (***StockPriceSerializer***)
- Write ***main***() method - creates ***producer***, create ***StockSender*** list passing each instance a ***producer,*** creates a thread pool so every stock sender gets it own thread, runs each stockSender in its own thread

StockPriceKafkaProducer imports, createProducer

Tos

StockPriceKafkaProducer | main()

```
1  package com.cloudurable.kafka.producer;
2
3  import com.cloudurable.kafka.StockAppConstants;
4  import com.cloudurable.kafka.producer.model.StockPrice;
5  import io.advantageous.boon.core.Lists;
6  import org.apache.kafka.clients.producer.*;
7  import org.apache.kafka.common.serialization.StringSerializer;
8  import org.slf4j.Logger;
9  import org.slf4j.LoggerFactory;
10
11 import java.util.List;
12 import java.util.Properties;
13 import java.util.concurrent.ExecutorService;
14 import java.util.concurrent.Executors;
15 import java.util.concurrent.TimeUnit;
16
17 public class StockPriceKafkaProducer {
18     private static final Logger logger =
19             LoggerFactory.getLogger(StockPriceKafkaProducer.class);
20
21     private static Producer<String, StockPrice> createProducer() {
22         final Properties props = new Properties();
23         setupBootstrapAndSerializers(props);
24         return new KafkaProducer<>(props);
25     }
```

Import classes and setup logger

createProducer used to create a KafkaProducer

createProducer() calls setupBoostrapAnd Serializers()

Configure Producer Bootstrap and Serializer

Tos

StockPriceKafkaProducer

```
27      private static void setupBootstrapAndSerializers(Properties props) {
28          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
29                  StockAppConstants.BOOTSTRAP_SERVERS);
30          props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
31          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
32                  StringSerializer.class.getName());
33
34          //Custom Serializer - config "value.serializer"
35          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
36                  StockPriceSerializer.class.getName());
37      }
```

- Create ***setupBootstrapAndSerializers*** to initialize bootstrap servers, client id, key serializer and custom serializer (***StockPriceSerializer***)
- ***StockPriceSerializer*** will serialize ***StockPrice*** into bytes

StockPriceKafkaProducer.main()

Tos

StockPriceKafkaProducer.java

StockPriceKafkaProducer › main()

```
39      public static void main(String... args)
40                                  throws Exception {
41
42          //Create Kafka Producer
43          final Producer<String, StockPrice> producer = createProducer();
44
45          //Create StockSender list
46          final List<StockSender> stockSenders = getStockSenderList(producer);
47
48          //Create a thread pool so every stock sender gets it own.
49          final ExecutorService executorService =
50                  Executors.newFixedThreadPool(stockSenders.size());
51
52          //Run each stock sender in its own thread.
53          stockSenders.forEach(executorService::submit);
54
55      }
```

- ***main*** method - creates ***producer***,
- create ***StockSender*** list passing each instance a ***producer***
- creates a thread pool (executorService)
- every StockSender runs in its own thread

StockAppConstants

Tos

```
1  package com.cloudurable.kafka;
2
3  public class StockAppConstants {
4      public final static String TOPIC = "stock-prices";
5      public final static String BOOTSTRAP_SERVERS =
6              "localhost:9092,localhost:9093,localhost:9094";
7
8  }
```

- topic name for Producer example
- List of bootstrap servers

StockPriceKafkaProducer.getStockSenderList

Tos

StockPriceKafkaProducer › getStockSenderList()

```
57  @   private static List<StockSender> getStockSenderList(
58              final Producer<String, StockPrice> producer) {
59          return Lists.list(
60                  new StockSender(StockAppConstants.TOPIC,
61                          new StockPrice( name: "IBM",  dollars: 100,  cents: 99),
62                          new StockPrice( name: "IBM",  dollars: 50,  cents: 10),
63                          producer,
64                           delayMinMs: 1,  delayMaxMs: 10
65                          ),
66                  new StockSender(
67                          StockAppConstants.TOPIC,
68                          new StockPrice( name: "SUN",  dollars: 100,  cents: 99),
69                          new StockPrice( name: "SUN",  dollars: 50,  cents: 10),
70                          producer,
71                           delayMinMs: 1,  delayMaxMs: 10
72                  ),
73                  new StockSender(
74                          StockAppConstants.TOPIC,
75                          new StockPrice( name: "GOOG",  dollars: 500,  cents: 99),
76                          new StockPrice( name: "GOOG",  dollars: 400,  cents: 10),
77                          producer,
78                           delayMinMs: 1,  delayMaxMs: 10
79                  ),
80                  new StockSender(
81                          StockAppConstants.TOPIC,
82                          new StockPrice( name: "INEL",  dollars: 100,  cents: 99),
83                          new StockPrice( name: "INEL",  dollars: 50,  cents: 10),
84                          producer,
85                           delayMinMs: 1,  delayMaxMs: 10
```

StockPriceSerializer

Tos

```java
package com.cloudurable.kafka.producer;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Serializer;
import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockPriceSerializer implements Serializer<StockPrice> {

    @Override
    public byte[] serialize(String topic, StockPrice data) {
        return data.toJson().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

- Converts *StockPrice* into byte array

StockSender run()

Tos

```
StockSender  run()
38
39      public void run() {
40          final Random random = new Random(System.currentTimeMillis());
41          int sentCount = 0;
42
43          while (true) {
44              sentCount++;
45              final ProducerRecord <String, StockPrice> record =
46                                              createRandomRecord(random);
47              final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);
48
49              try {
50                  final Future<RecordMetadata> future = producer.send(record);
51                  if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
52                  Thread.sleep(delay);
53              } catch (InterruptedException e) {
54                  if (Thread.interrupted()) {
55                      break;
56                  }
57              } catch (ExecutionException e) {
58                  logger.error("problem sending record to producer", e);
59              }
60          }
61      }
```

- In loop, creates random record, *send* record, waits random time

StockSender displayRecordMetaData

Tos

```
StockSender | displayRecordMetaData()

62
63      private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
64                                         final Future<RecordMetadata> future)
65                                 throws InterruptedException, ExecutionException {
66          final RecordMetadata recordMetadata = future.get();
67          logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
68                                    "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
69                  record.key(),
70                  record.value().toJson(),
71                  recordMetadata.topic(),
72                  recordMetadata.partition(),
73                  recordMetadata.offset(),
74                  new Date(recordMetadata.timestamp())
75                  ));
76      }
```

- Every 100 records displayRecordMetaData gets called
- Prints out record info, and recordMetadata info:
  - key, JSON value, topic, partition, offset, time
  - uses Future from call to ***producer.send***()

Run All 3 Brokers

Tos

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"
```

start-2nd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"
```

start-3rd-server.sh

```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-2.properties"
```

server-1.properties

```
broker.id=1
port=9093
log.dirs=./logs/kafka-1
min.insync.replicas=3
compression.type=producer
```

server-2.properties

```
broker.id=2
port=9094
log.dirs=./logs/kafka-2
min.insync.replicas=3
compression.type=producer
auto.create.topics.enable=false
message.max.bytes=65536
replica.lag.time.max.ms=5000
delete.topic.enable=true
```

Run create-topic.sh script

Tos

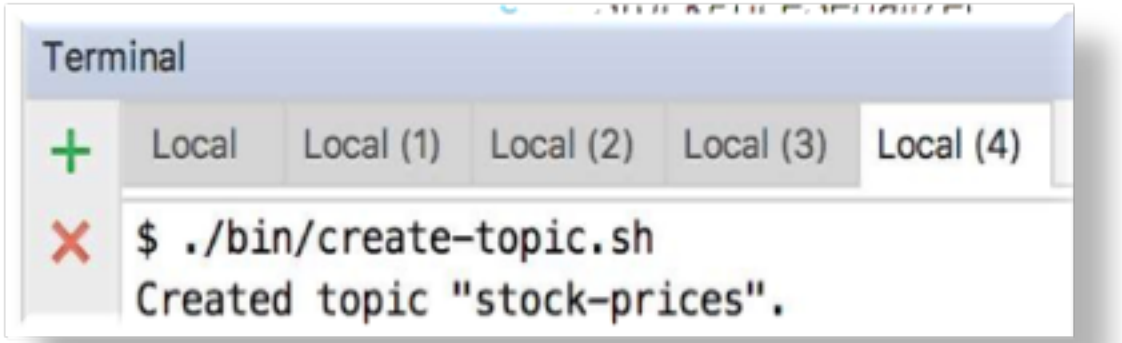
create-topic.sh

```
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh \
    --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 3 \
    --topic stock-prices

```

Name of the topic is stock-prices

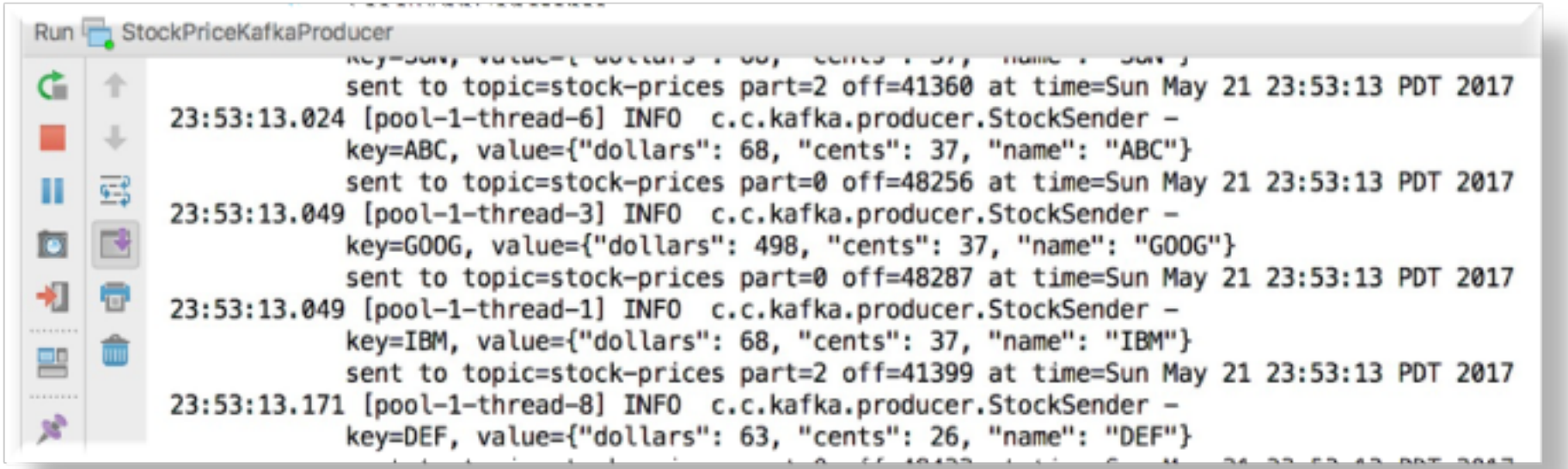
Three partitions

Replication factor of three



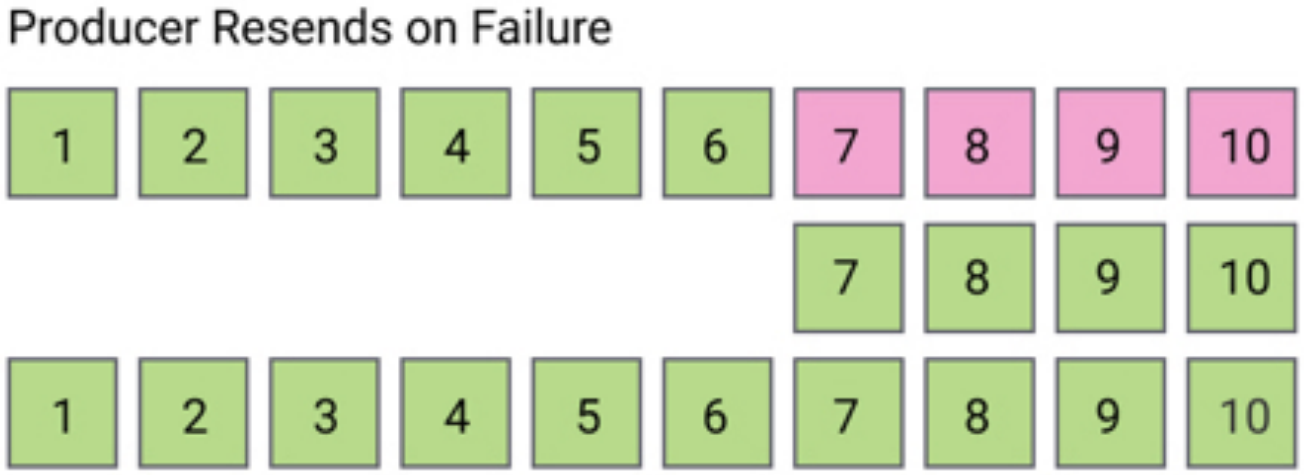
Run StockPriceKafkaProducer

Tos



- Run ***StockPriceKafkaProducer*** from the IDE
- You should see log messages from ***StockSender***(s) with ***StockPrice*** name, JSON value, partition, offset, and time

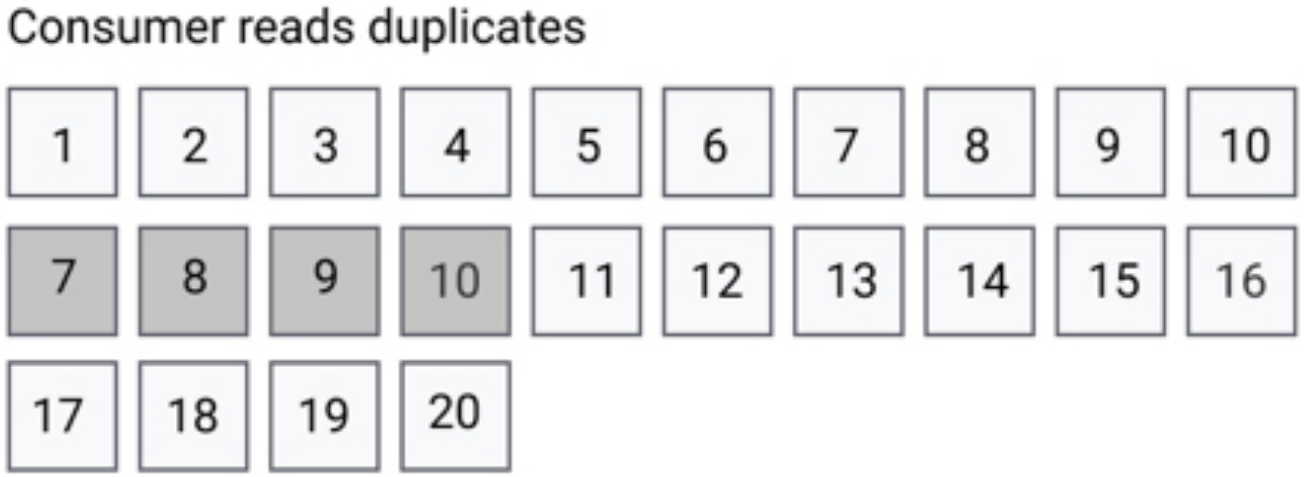
idempotent producer



enable.idempotence=true.

Limitation 1: Acks=All

Limitation 2: max.in.flight.requests.per.connection <= 5



How it Works

Tos

- M1 (PID: 1, SN: 1) - written to partition. For PID 1, Max SN=1
- M2 (PID: 1, SN: 2) - written to partition. For PID 1, Max SN=2
- M3 (PID: 1, SN: 3) - written to partition. For PID 1, Max SN=3
- M4 (PID: 1, SN: 4) - written to partition. For PID 1, Max SN=4
- M5 (PID: 1, SN: 5) - written to partition. For PID 1, Max SN=5
- M6 (PID: 1, SN: 6) - written to partition. For PID 1, Max SN=6
- M4 (PID: 1, SN: 4) - rejected, SN <= Max SN
- M5 (PID: 1, SN: 5) - rejected, SN <= Max SN
- M6 (PID: 1, SN: 6) - rejected, SN <= Max SN
- M7 (PID: 1, SN: 7) - written to partition. For PID 1, Max SN=7
- M8 (PID: 1, SN: 8) - written to partition. For PID 1, Max SN=8
- M9 (PID: 1, SN: 9) - written to partition. For PID 1, Max SN=9
- M10 (PID: 1, SN: 10) - written to partition. For PID 1, Max SN=10

How it Works

Tos

Each producer gets assigned a **Producer Id (PID)** and it includes its PID every time it sends messages to a broker.
Additionally, each message gets a monotonically increasing **sequence number.**
A separate sequence is maintained for each topic partition that a producer sends messages to.
On the broker side, on a per partition basis, it keeps track of the largest PID-Sequence Number combination is has successfully written.
When a lower sequence number is received, it is discarded

transactional producer

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerialize

producer.initTransactions();

try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Integer.toString(i)));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // We can't recover from these exceptions, so our only option is to close the producer and exit.
    producer.close();
} catch (KafkaException e) {
    // For all other exceptions, just abort the transaction and try again.
    producer.abortTransaction();
}
producer.close();
```

Message serialization using Avro

Avro relies on schemas so as to provide efficient serialization of the data.

The schema is written in JSON format and describes the fields and their types.
There are 2 cases:

- when serializing to a file, the schema is written to the file
- in RPC - such as between Kafka and Spark - both systems should know the schema prior to exchanging data, or they could exchange the schema during the connection handshake.

What makes Avro handy is that you do not need to generate data classes.

Message serialization using Avro

Add dependency & Schema

```xml
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.8.0</version>
</dependency>
```

```
{
    "fields": [
        { "name": "str1", "type": "string" },
        { "name": "str2", "type": "string" },
        { "name": "int1", "type": "int" }
    ],
    "name": "myrecord",
    "type": "record"
}
```

Message serialization using Avro - The Producer

```java
public static void main(String[] args) throws InterruptedException {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer", "org.apache.kafka.common.serialization.ByteArraySerializer");

    Schema.Parser parser = new Schema.Parser();
    Schema schema = parser.parse(USER_SCHEMA);
    Injection<GenericRecord, byte[]> recordInjection = GenericAvroCodecs.toBinary(schema);

    KafkaProducer<String, byte[]> producer = new KafkaProducer<>(props);

    for (int i = 0; i < 1000; i++) {
        GenericData.Record avroRecord = new GenericData.Record(schema);
        avroRecord.put("str1", "Str 1-" + i);
        avroRecord.put("str2", "Str 2-" + i);
        avroRecord.put("int1", i);

        byte[] bytes = recordInjection.apply(avroRecord);

        ProducerRecord<String, byte[]> record = new ProducerRecord<>("mytopic", bytes);
        producer.send(record);

        Thread.sleep(250);

    }

    producer.close();
}
```

```java
public static final String USER_SCHEMA = "{"
        + "\"type\":\"record\","
        + "\"name\":\"myrecord\","
        + "\"fields\":["
        + "  { \"name\":\"str1\", \"type\":\"string\" },"
        + "  { \"name\":\"str2\", \"type\":\"string\" },"
        + "  { \"name\":\"int1\", \"type\":\"int\" }"
        + "]}";
```

Lab : Java API – Producer-II a API