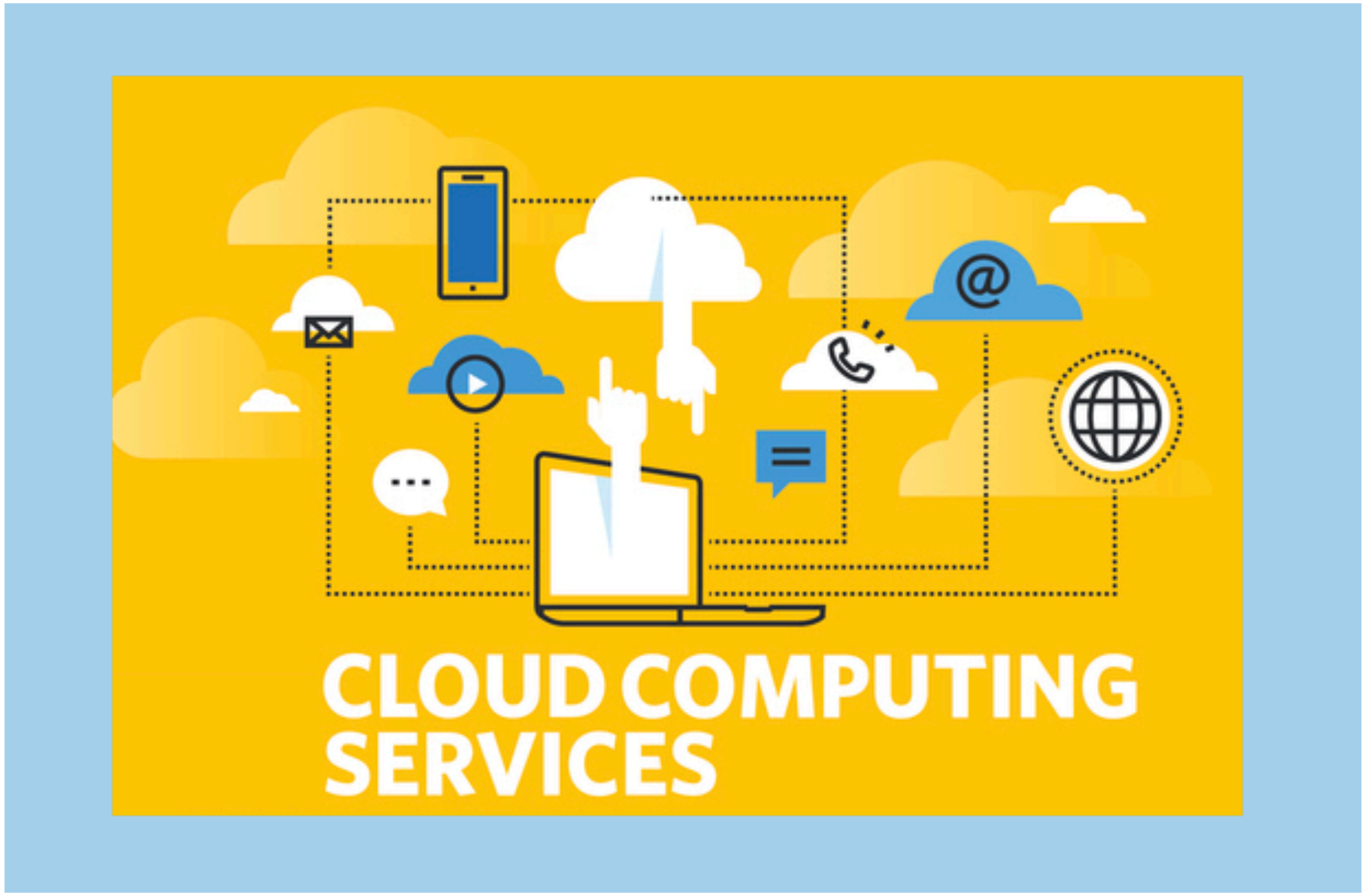


Cassandra and Kafka Support on AWS/EC2

Cloudurable
Introduction to Kafka

Support around Cassandra and Kafka running in EC2

CLOUDURABLE

Cassandra / Kafka Support in EC2/AWS

Kafka Introduction

Kafka messaging

What is Kafka?

- Distributed Streaming Platform
  - Publish and Subscribe to streams of records
  - Fault tolerant storage
  - Process records as they occur

Kafka Usage

- Build real-time streaming data pipe-lines
  - Enable in-memory microservices (actors, Akka, Vert.x, Qbit)
- Build real-time streaming applications that react to streams
  - Real-time data analytics
  - Transform, react, aggregate, join real-time data flows

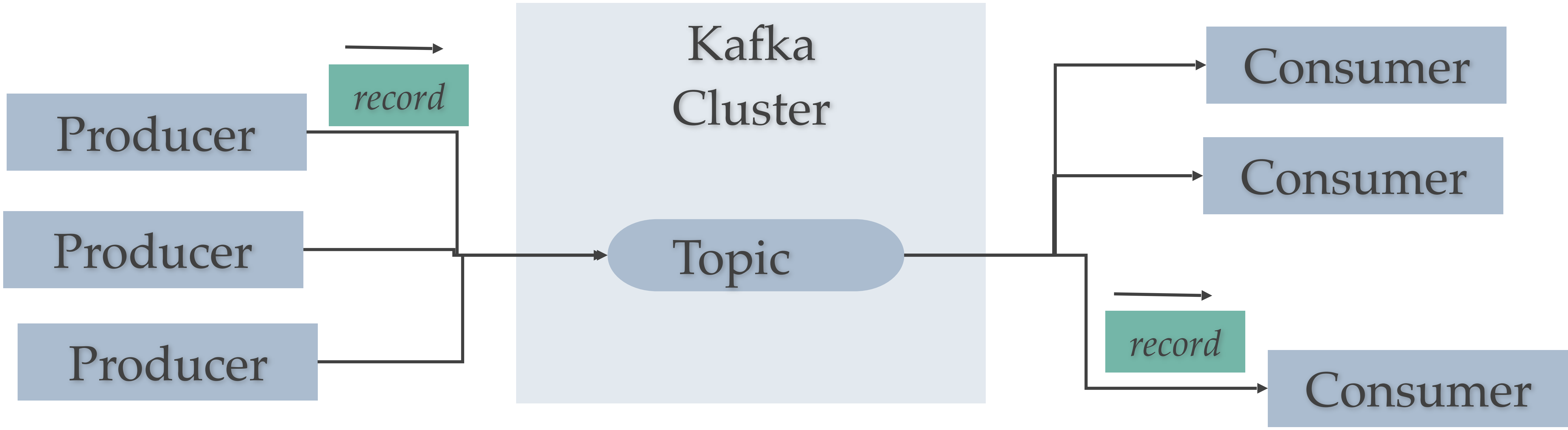
Kafka Use Cases

- Metrics / KPIs gathering
  - Aggregate statistics from many sources
- Even Sourcing
  - Used with microservices (in-memory) and actor systems
- Commit Log
  - External commit log for distributed systems. Replicated data between nodes, re-sync for nodes to restore state
- Real-time data analytics, Stream Processing, Log Aggregation, Messaging, Click-stream tracking, Audit trail, etc.

Who uses Kafka?

- ***LinkedIn***: Activity data and operational metrics
- ***Twitter***: Uses it as part of Storm – stream processing infrastructure
- ***Square***: Kafka as bus to move all system events to various Square data centers (logs, custom events, metrics, an so on). Outputs to Splunk, Graphite, Esper-like alerting systems
- Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, CloudFlare, DataDog, LucidWorks, MailChimp, NetFlix, etc.

Kafka: Topics, Producers, and Consumers



Kafka Fundamentals

- *Records* have a ***key, value*** and ***timestamp***
- *Topic* a stream of records ("/orders", "/user-signups"), feed name
  - *Log* topic storage on disk
  - ***Partition*** / Segments (parts of Topic Log)
- *Producer* API to produce a streams or records
- *Consumer* API to consume a stream of records
- *Broker*: Cluster of Kafka servers running in cluster form broker. Consists on many processes on many servers
- *ZooKeeper*: Does coordination of broker and consumers. Consistent file system for configuration information and leadership election

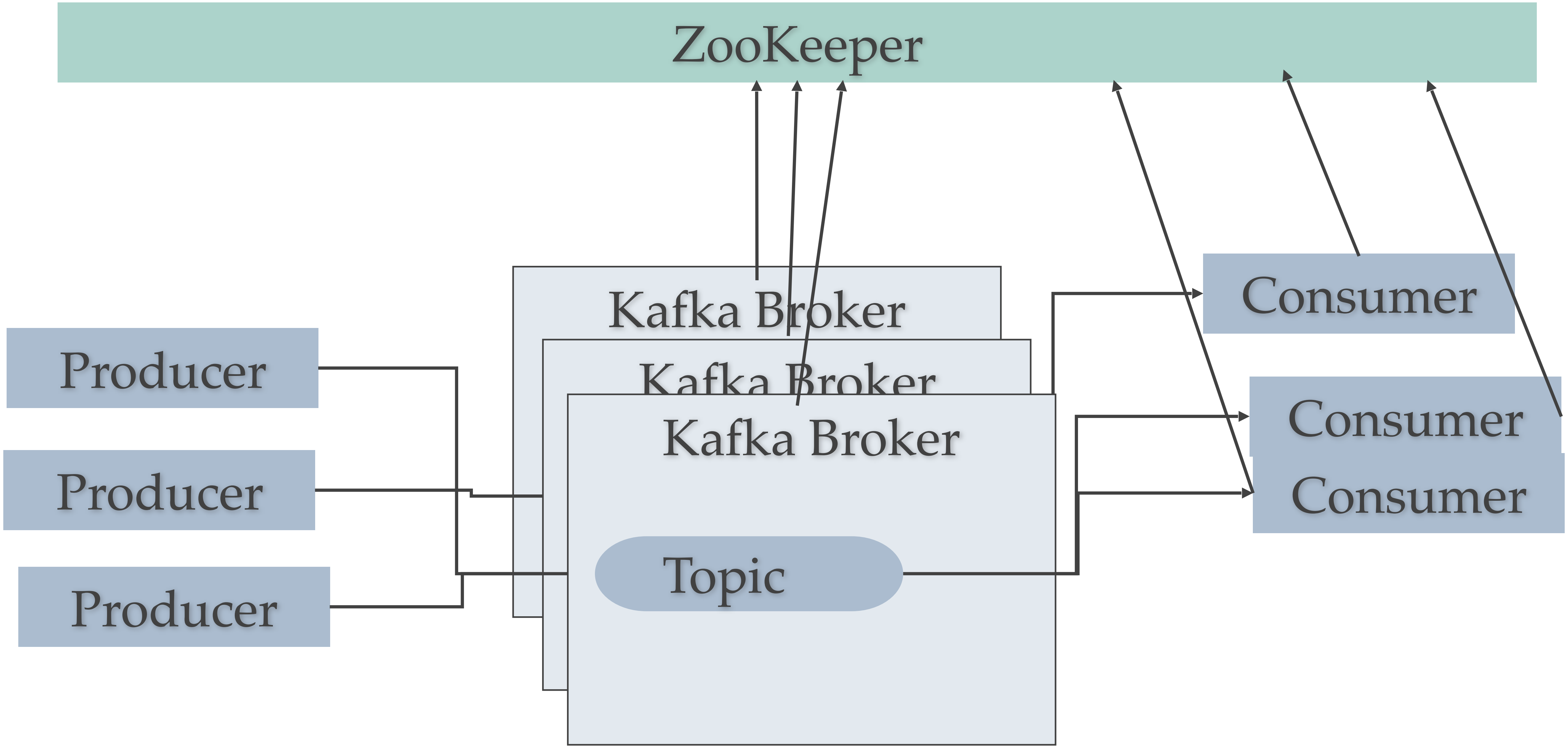
Kafka Performance details

- ***Topic*** is like a feed name ***"/shopping-cart-done", "/user-signups"***, which Producers write to and Consumers read from
- ***Topic*** associated with a log which is data structure on disk
- ***Producer***(s) append ***Records*** at end of Topic log
- Whilst many ***Consumers*** read from Kafka at their own cadence
  - Each Consumer (Consumer Group) tracks offset from where they left off reading
- How can Kafka scale if multiple producers and consumers read/write to the same Kafka Topic log?
  - Sequential writes to filesystem are ***fast*** (700 MB or more a second)
  - Kafka scales writes and reads by ***sharding*** Topic logs into ***Partitions*** (parts of a Topic log)
  - Topics logs can be split into multiple Partitions ***different machines/different disks***
  - Multiple Producers can write to different Partitions of the same Topic
  - Multiple Consumers Groups can read from different partitions efficiently
- ***Partitions*** can be distributed on different machines in a cluster
  - high performance with horizontal scalability and failover

Kafka Fundamentals 2

- ***Kafka*** uses ***ZooKeeper*** to form ***Kafka Brokers*** into a cluster
- Each ***node*** in Kafka cluster is called a ***Kafka Broker***
- ***Partitions*** can be ***replicated*** across ***multiple nodes*** for failover
- One node/partition's replicas is chosen as ***leader***
- Leader handles all reads and writes of Records for partition
- Writes to partition are ***replicated*** to ***followers*** (node/partition pair)
- An ***follower*** that is ***in-sync*** is called an ***ISR (in-sync replica)***
- If a partition leader fails, one ISR is chosen as new leader

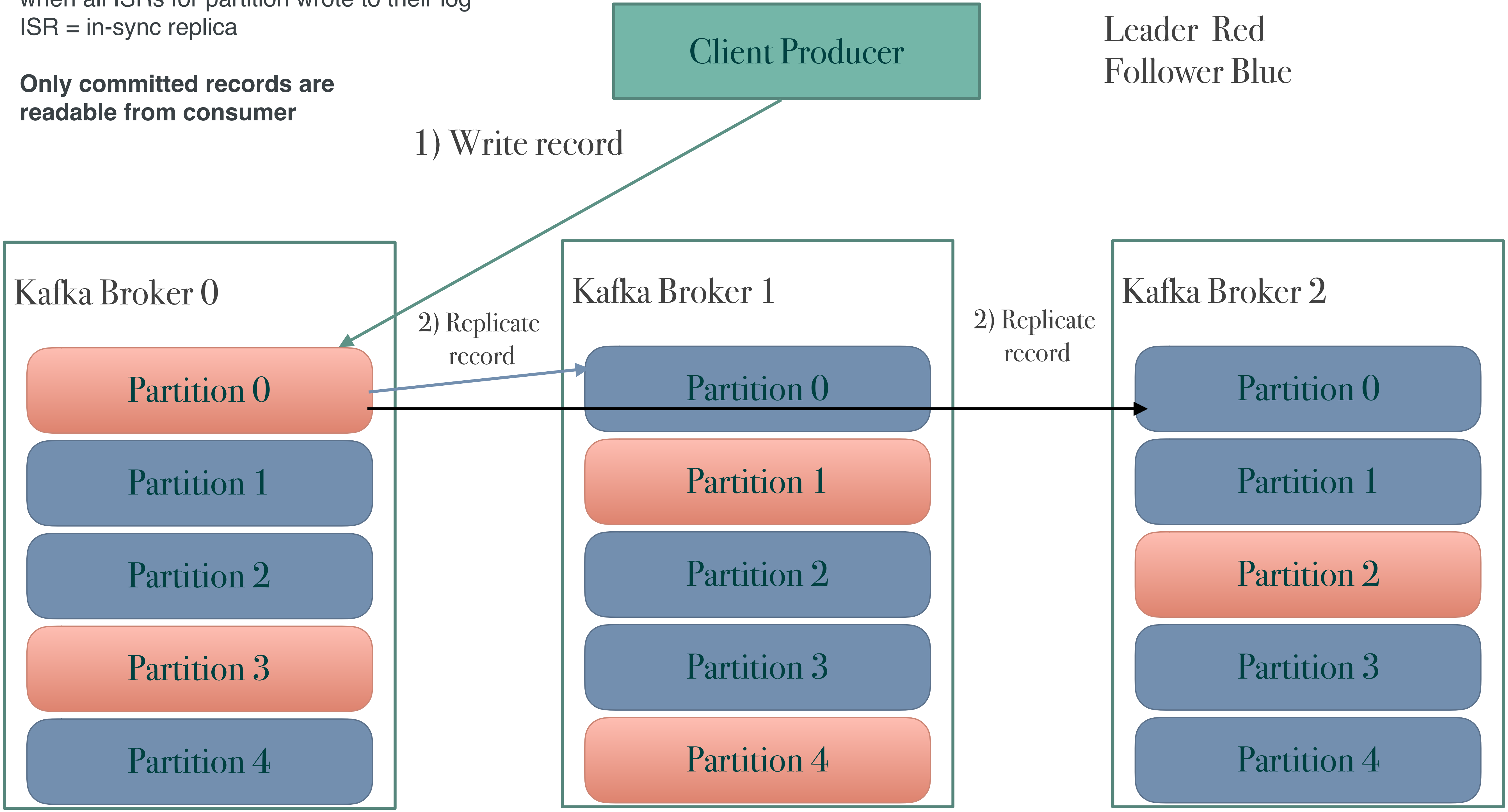
ZooKeeper does coordination for Kafka Consumer and Kafka Cluster



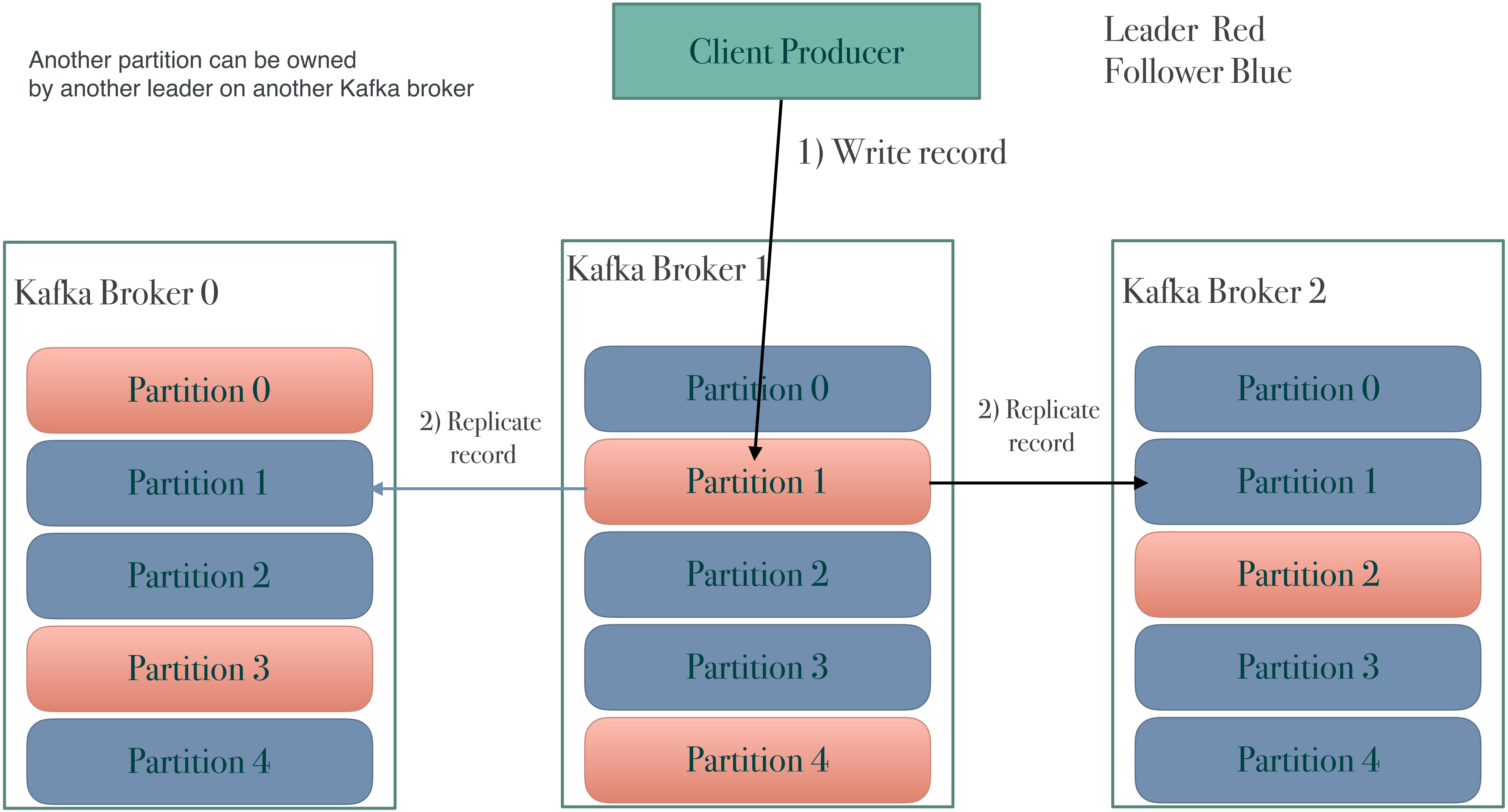
Replication of Kafka Partitions 0

Record is considered "committed" when all ISRs for partition wrote to their log
ISR = in-sync replica

Only committed records are readable from consumer



Replication of Kafka Partitions 1



Another partition can be owned by another leader on another Kafka broker

Client Producer

Leader Red
Follower Blue

1) Write record

Kafka Broker 0

Partition 0
Partition 1
Partition 2
Partition 3
Partition 4

2) Replicate record

Kafka Broker 1

Partition 0
Partition 1
Partition 2
Partition 3
Partition 4

2) Replicate record

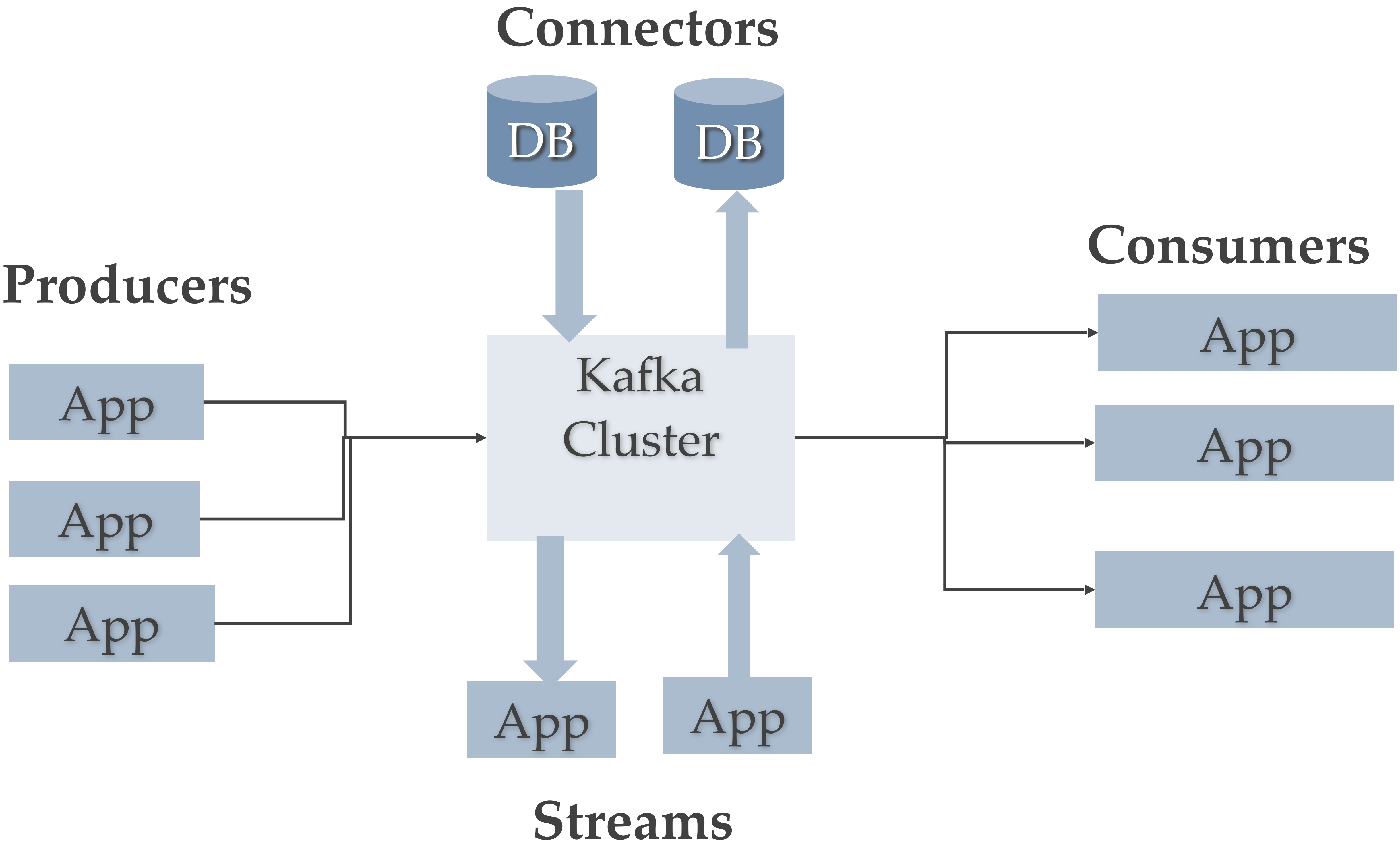
Kafka Broker 2

Partition 0
Partition 1
Partition 2
Partition 3
Partition 4

Kafka Extensions

- ***Streams*** API to transform, aggregate, process records from a stream and produce derivative streams
- ***Connector*** API reusable producers and consumers (e.g., stream of changes from DynamoDB)

Kafka Connectors and Streams



Kafka Polyglot clients / Wire protocol

- Kafka communication from clients and servers wire protocol over TCP protocol
- Protocol versioned
- Maintains backwards compatibility
- Many languages supported

Topics and Logs

- *Topic* is a stream of records
- *Topics* stored in log
- *Log* broken up into ***partitions*** and ***segments***
- *Topic* is a category or stream name
- Topics are pub/sub
  - Can have zero or many consumer groups (subscribers)
- *Topics* are broken up into partitions for speed and size

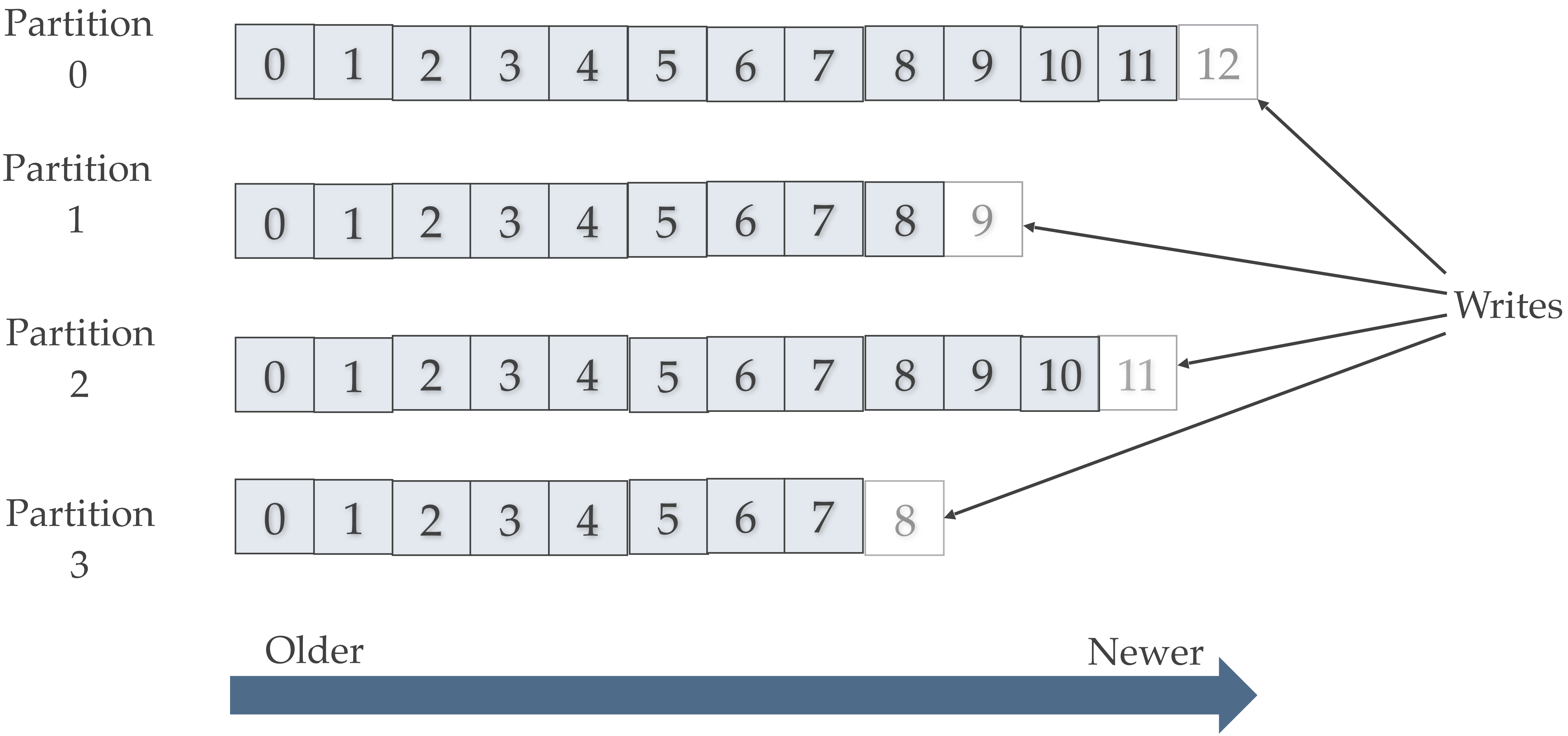
Topic Partitions

- *Topics* are broken up into ***partitions***
- ***Partitions*** are decided usually by key of record
  - Key of record determines which partition
- ***Partitions*** are used to scale Kafka across many servers
  - Record sent to correct partition by key
- ***Partitions*** are used to facilitate parallel consumers
  - Records are consumed in parallel up to the number of partitions

Partition Log

- ***Order*** is maintained only in a single ***partition***
  - ***Partition*** is ordered, immutable sequence of records that is continually appended to—a structured commit ***log***
- Producers write at their own cadence so order of Records cannot be guaranteed across partitions
- Producers pick the partition such that Record/messages goes to a given same partition based on the data
  - Example have all the events of a certain 'employeeId' go to same partition
  - If order within a partition is not needed, a 'Round Robin' partition strategy can be used so Records are evenly distributed across partitions.
- ***Records*** in partitions are assigned ***sequential id*** number called the *offset*
- ***Offset*** identifies each record within the partition
- ***Topic Partitions*** allow Kafka log to scale beyond a size that will fit on a single server
  - Topic partition must fit on servers that host it, but topic can span many partitions hosted by many servers
- Topic Partitions are unit of ***parallelism*** - each consumer in a consumer group can work on one partition at a time

Kafka Topic Partitions Layout



Partition 0: 0 1 2 3 4 5 6 7 8 9 10 11 12

Partition 1: 0 1 2 3 4 5 6 7 8 9

Partition 2: 0 1 2 3 4 5 6 7 8 9 10 11

Partition 3: 0 1 2 3 4 5 6 7 8

Writes

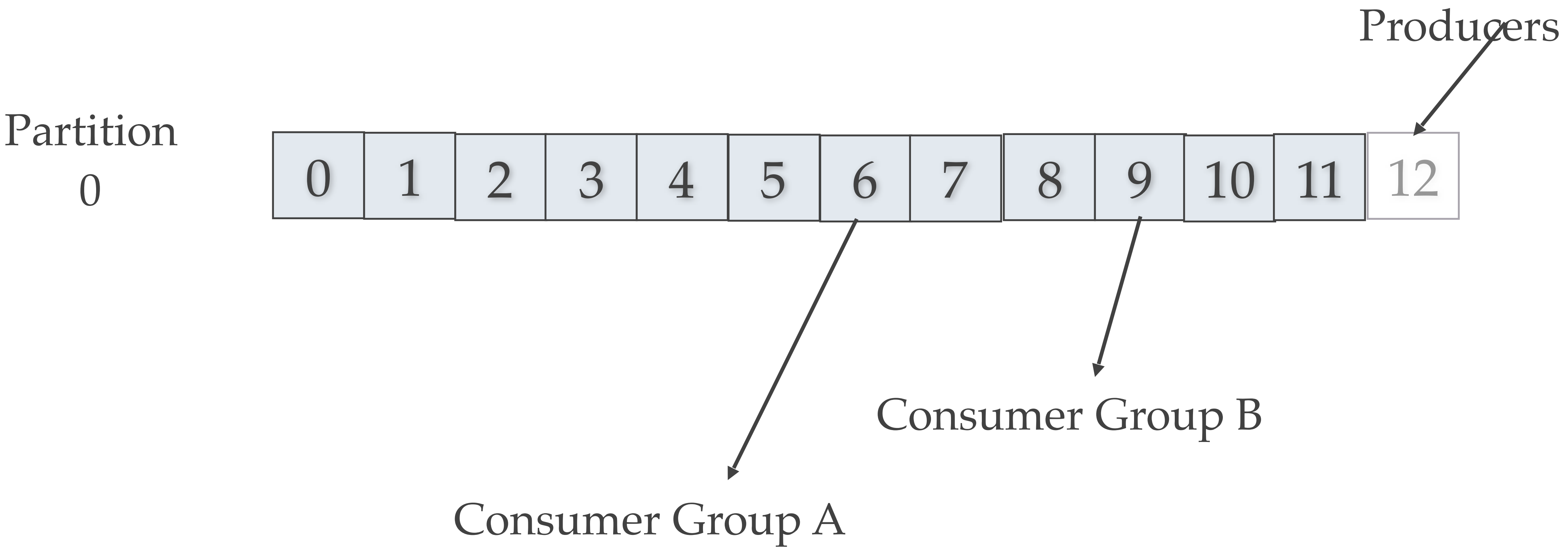
Older

Newer

Kafka Record retention

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based
  - Compaction
- Retention policy of three days or two weeks or a month
- It is available for consumption until discarded by time, size or compaction
- Consumption speed not impacted by size

Kafka Consumers / Producers



Consumers remember offset where they left off.

Consumers groups each have their own offset.

Kafka Partition Distribution

- Each partition has ***leader server*** and zero or more ***follower servers***
  - ***Leader*** handles all read and write requests for partition
  - ***Followers*** replicate leader, and take over if leader dies
  - Used for parallel consumer handling within a group
- Partitions of log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of partitions
- Each partition can be replicated across a configurable number of Kafka servers
  - Used for fault tolerance

Kafka Producers

- ***Producers*** send records to topics
- ***Producer*** picks which partition to send record to per topic
  - Can be done in a ***round-robin***
  - Can be based on priority
  - Typically based on ***key*** of ***record***
  - Kafka ***default partitioner*** for Java uses hash of keys to choose partitions, or a round-robin strategy if no key
- Important: *Producer picks partition*

Kafka Consumer Groups

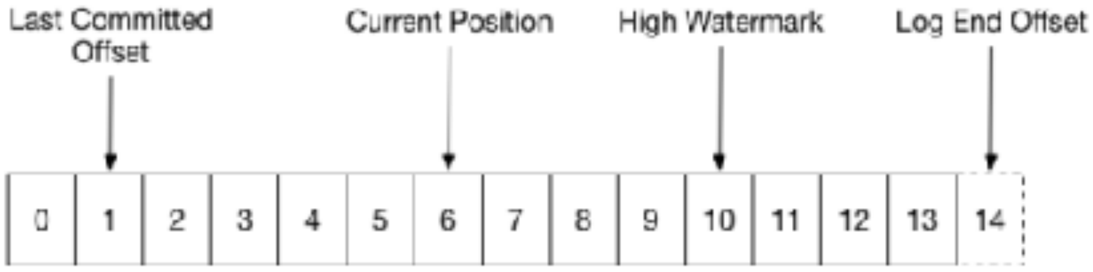
- Consumers are grouped into a ***Consumer Group***
  - ***Consumer group*** has a unique id
  - Each ***consumer group*** is a subscriber
  - Each ***consumer group*** maintains its own offset
  - Multiple subscribers = multiple consumer groups
- ***A Record*** is delivered to one ***Consumer*** in a ***Consumer Group***
- Each consumer in consumer groups takes records and only one consumer in group gets same record
- Consumers in Consumer Group ***load balance record consumption***

Kafka Consumer Groups 2

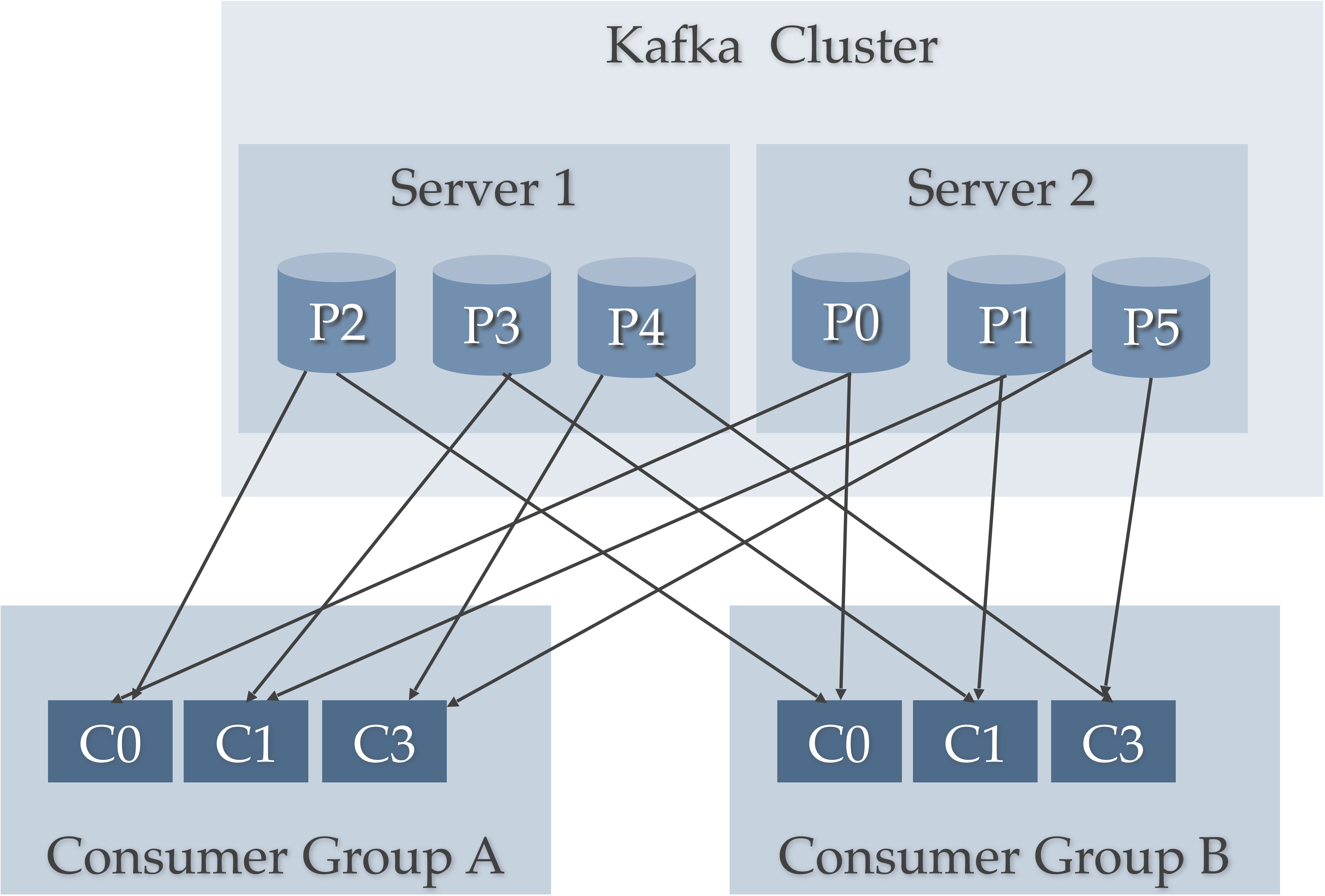
- How does Kafka divide up topic so multiple Consumers in a consumer group can process a topic?
- Kafka makes you group consumers into consumers group with a group id
- Consumer with same id belong in same Consumer Group
- One ***Kafka broker*** becomes ***group coordinator*** for Consumer Group
  - assigns partitions when new members arrive (older clients would talk direct to ZooKeeper now broker does coordination)
  - or reassign partitions when group members leave or topic changes (config / meta-data change
- When ***Consumer group*** is created, offset set according to reset policy of topic

Kafka Consumer Group 3

- If ***Consumer*** fails before sending commit offset XXX to Kafka broker,
  - different ***Consumer*** can continue from the last committed offset
  - some Kafka records could be reprocessed (***at least once behavior***)
- ***"Log end offset"*** is offset of last record written to log partition and where ***Producers*** write to next
- ***"High watermark"*** is offset of last record that was successfully replicated to all partitions followers
- ***Consumer*** only reads up to the "high watermark". ***Consumer can't read un-replicated data***
- Only a single ***Consumer*** from the same ***Consumer Group*** can access a single ***Partition***
- If ***Consumer Group*** count ***exceeds*** Partition count:
  - Extra Consumers remain idle; can be used for failover
- If more Partitions than Consumer Group instances,
  - Some Consumers will read from more than one partition



2 server Kafka cluster hosting 4 partitions (P0-P5)



Kafka Consumer Consumption

- Kafka ***Consumer*** consumption divides partitions over consumer instances
  - Each Consumer is exclusive consumer of a "fair share" of partitions
  - Consumer membership in group is handled by the Kafka protocol dynamically
  - If new Consumers join Consumer group they get share of partitions
  - If Consumer dies, its partitions are split among remaining live Consumers in group
- Order is only guaranteed within a single partition
- Since ***records*** are typically stored ***by key into a partition*** then order per partition is sufficient for most use cases

Kafka vs JMS Messaging

- It is a bit like both Queues and Topics in JMS
- Kafka is a queue system per consumer in consumer group so load balancing like JMS queue
- Kafka is a topic/pub/sub by offering Consumer Groups which act like subscriptions
    - Broadcast to multiple consumer groups
- By design Kafka is better suited for scale due to partition topic log
- Also by moving location in log to client/consumer side of equation instead of the broker, less tracking required by Broker
- Handles parallel consumers better

Kafka scalable message storage

- Kafka acts as a good storage system for records/messages
- Records written to Kafka topics are persisted to disk and replicated to other servers for fault-tolerance
- Kafka Producers can wait on acknowledgement
  - Write not complete until fully replicated
- Kafka disk structures scales well
  - Writing in large streaming batches is fast
- Clients/Consumers control read position (offset)
  - Kafka acts like high-speed file system for commit log storage, replication

Kafka Stream Processing

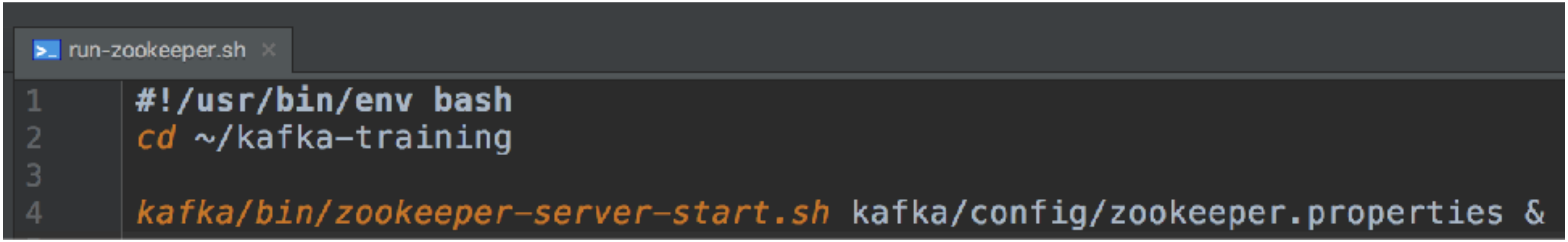
- Kafka for Stream Processing
  - Kafka enable ***real-time*** processing of streams.
- Kafka supports stream processor
  - Stream processor takes continual streams of records from input topics, performs some processing, transformation, aggregation on input, and produces one or more output streams
- A video player app might take in input streams of videos watched and videos paused, and output a stream of user preferences and gear new video recommendations based on recent user activity or aggregate activity of many users to see what new videos are hot
- Kafka Stream API solves hard problems with out of order records, aggregating across multiple streams, joining data from multiple streams, allowing for stateful computations, and more
- Stream API builds on core Kafka primitives and has a life of its own

Using Kafka Single Node

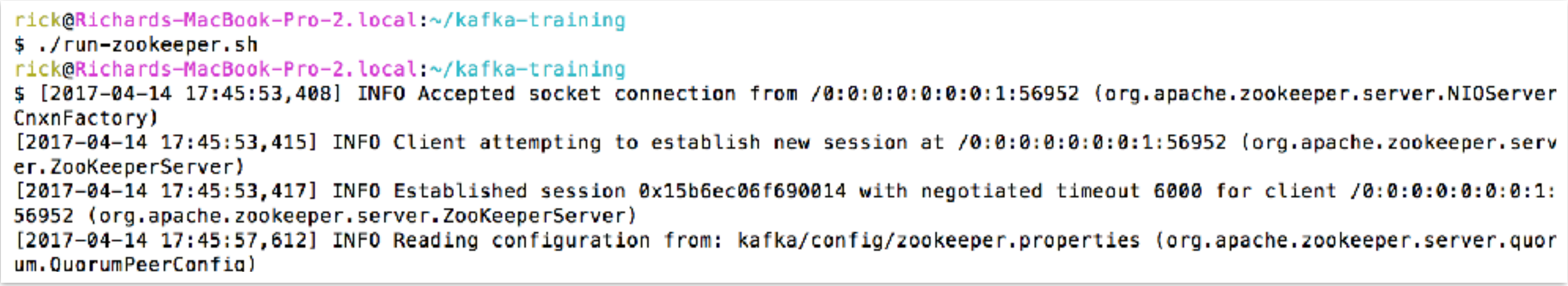
Run Kafka

- Run ZooKeeper
- Run Kafka Server/Broker
- Create Kafka Topic
- Run producer
- Run consumer

Run ZooKeeper

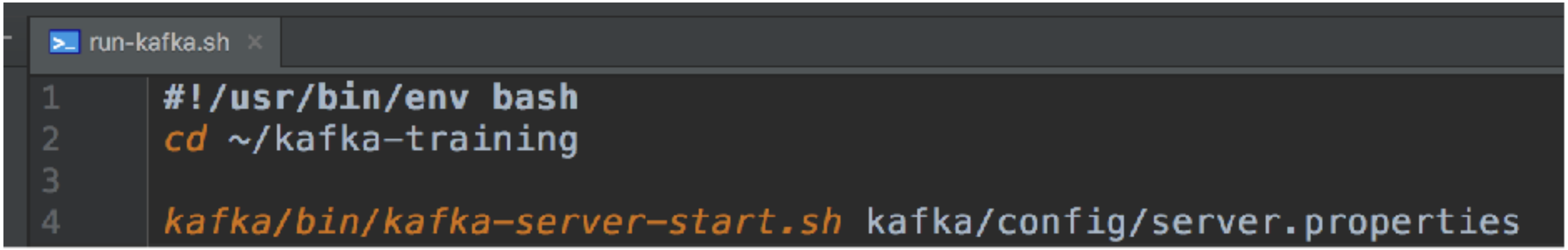


```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh kafka/config/zookeeper.properties &
```

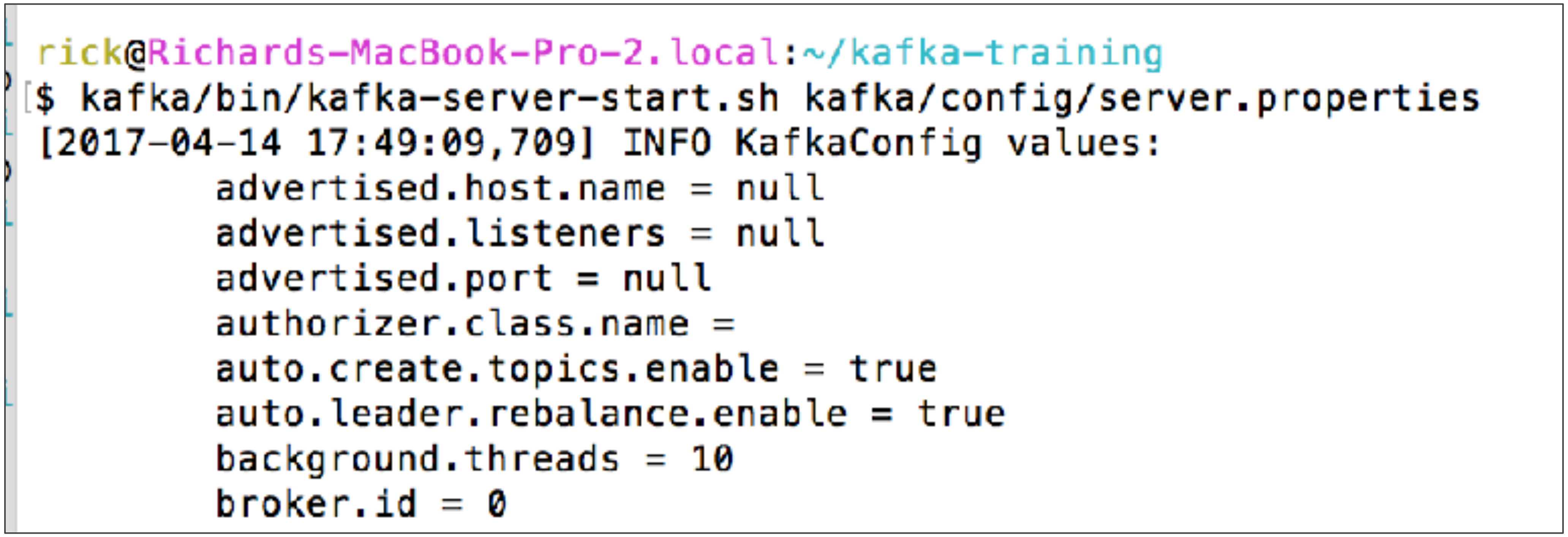


```
rick@Richards-MacBook-Pro-2.local:~/kafka-training
$ ./run-zookeeper.sh
rick@Richards-MacBook-Pro-2.local:~/kafka-training
$ [2017-04-14 17:45:53,408] INFO Accepted socket connection from /0:0:0:0:0:0:0:1:56952 (org.apache.zookeeper.server.NIOServer
CnxnFactory)
[2017-04-14 17:45:53,415] INFO Client attempting to establish new session at /0:0:0:0:0:0:0:1:56952 (org.apache.zookeeper.serv
er.ZooKeeperServer)
[2017-04-14 17:45:53,417] INFO Established session 0x15b6ec06f690014 with negotiated timeout 6000 for client /0:0:0:0:0:0:0:1:
56952 (org.apache.zookeeper.server.ZooKeeperServer)
[2017-04-14 17:45:57,612] INFO Reading configuration from: kafka/config/zookeeper.properties (org.apache.zookeeper.server.quor
um.QuorumPeerConfig)
```

Run Kafka Server

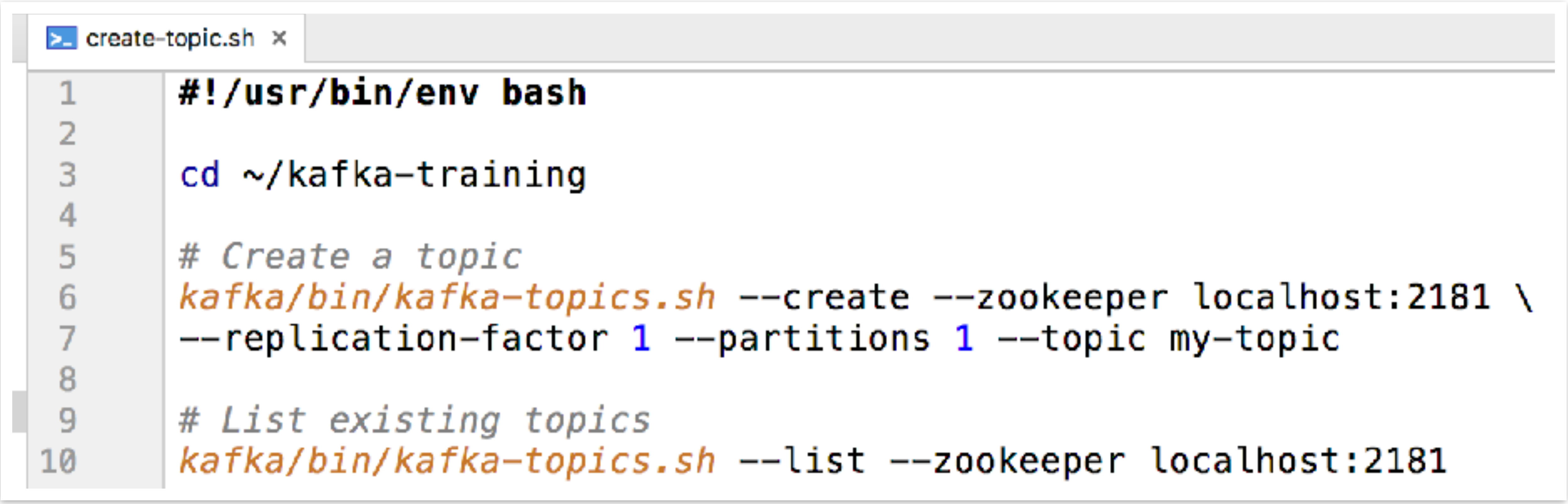


```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-server-start.sh kafka/config/server.properties
```



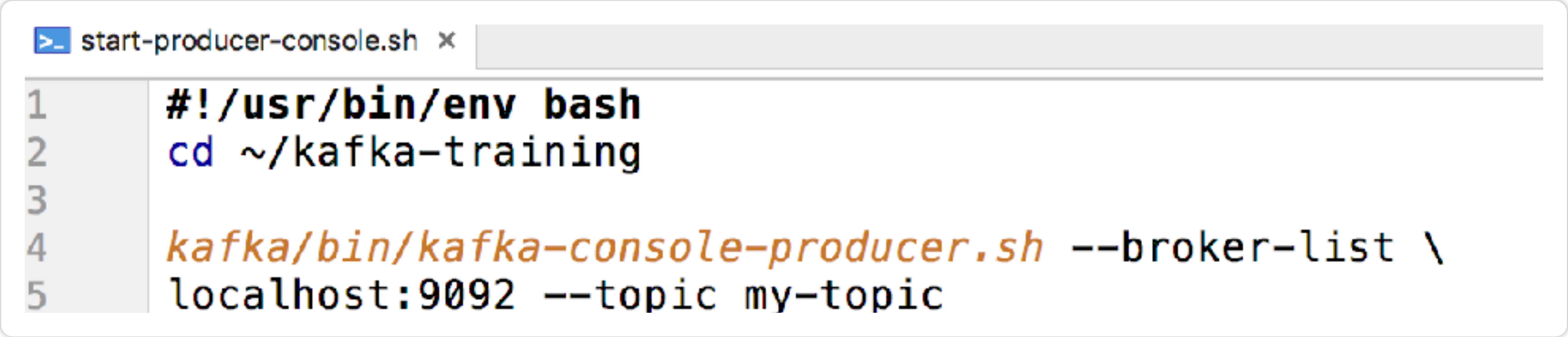
```
rick@Richards-MacBook-Pro-2.local:~/kafka-training
$ kafka/bin/kafka-server-start.sh kafka/config/server.properties
[2017-04-14 17:49:09,709] INFO KafkaConfig values:
        advertised.host.name = null
        advertised.listeners = null
        advertised.port = null
        authorizer.class.name =
        auto.create.topics.enable = true
        auto.leader.rebalance.enable = true
        background.threads = 10
        broker.id = 0
```

Create Kafka Topic



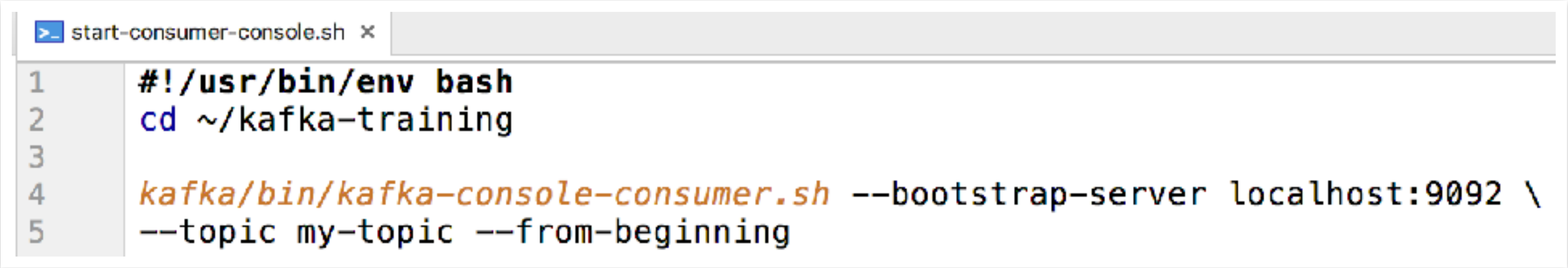
```
#!/usr/bin/env bash

cd ~/kafka-training

# Create a topic
kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
--replication-factor 1 --partitions 1 --topic my-topic

# List existing topics
kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Kafka Producer



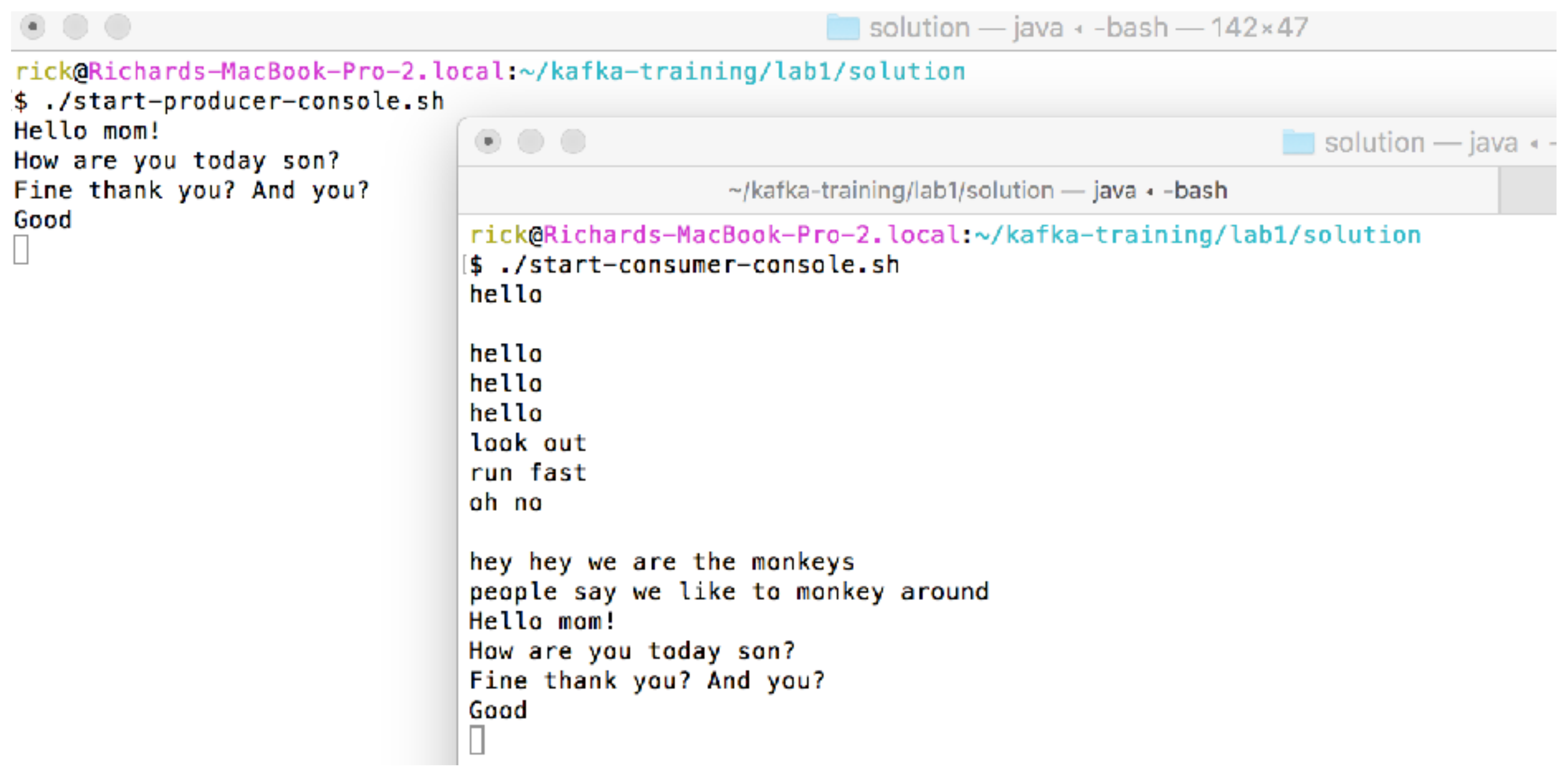
```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-producer.sh --broker-list \
localhost:9092 --topic my-topic
```

Kafka Consumer



```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
--topic my-topic --from-beginning
```

Running Kafka Producer and Consumer


solution — java ◂ -bash — 142×47
solution — java ◂ -
~/kafka-training/lab1/solution — java ◂ -bash

```
rick@Richards-MacBook-Pro-2.local:~/kafka-training/lab1/solution
$ ./start-producer-console.sh
Hello mom!
How are you today son?
Fine thank you? And you?
Good
```

```
rick@Richards-MacBook-Pro-2.local:~/kafka-training/lab1/solution
$ ./start-consumer-console.sh
hello

hello
hello
hello
look out
run fast
oh no

hey hey we are the monkeys
people say we like to monkey around
Hello mom!
How are you today son?
Fine thank you? And you?
Good
```

Use Kafka to send and receive messages

Lab 1-A Use Kafka

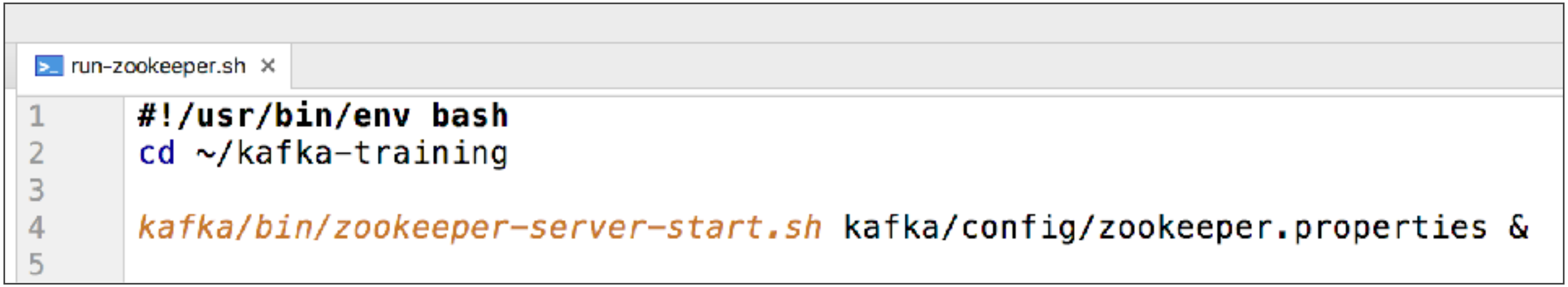
Use single server version of Kafka

Using Kafka Cluster

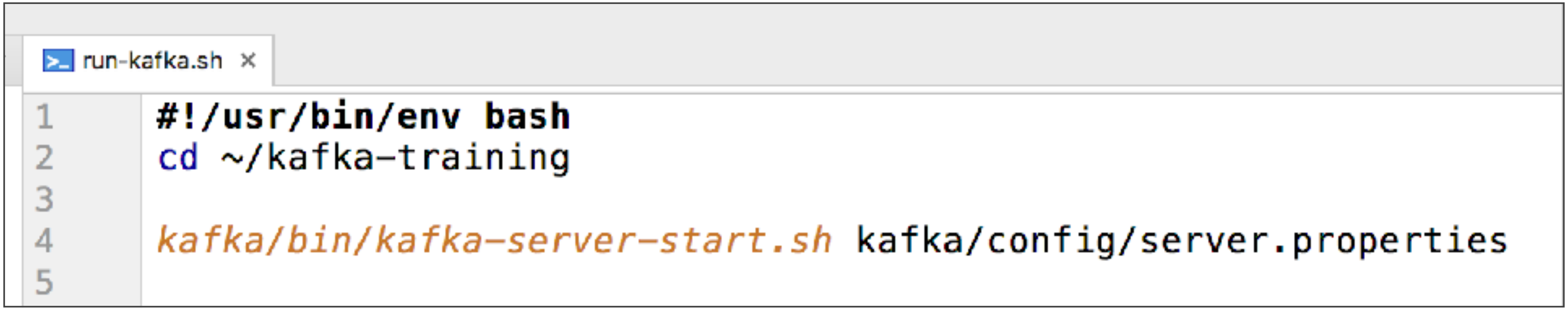
Running many nodes

- Modify properties files
  - Change port
  - Change Kafka log location
- Start up many Kafka server instances
- Create Replicated Topic

Leave everything from before running



```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh kafka/config/zookeeper.properties &

```

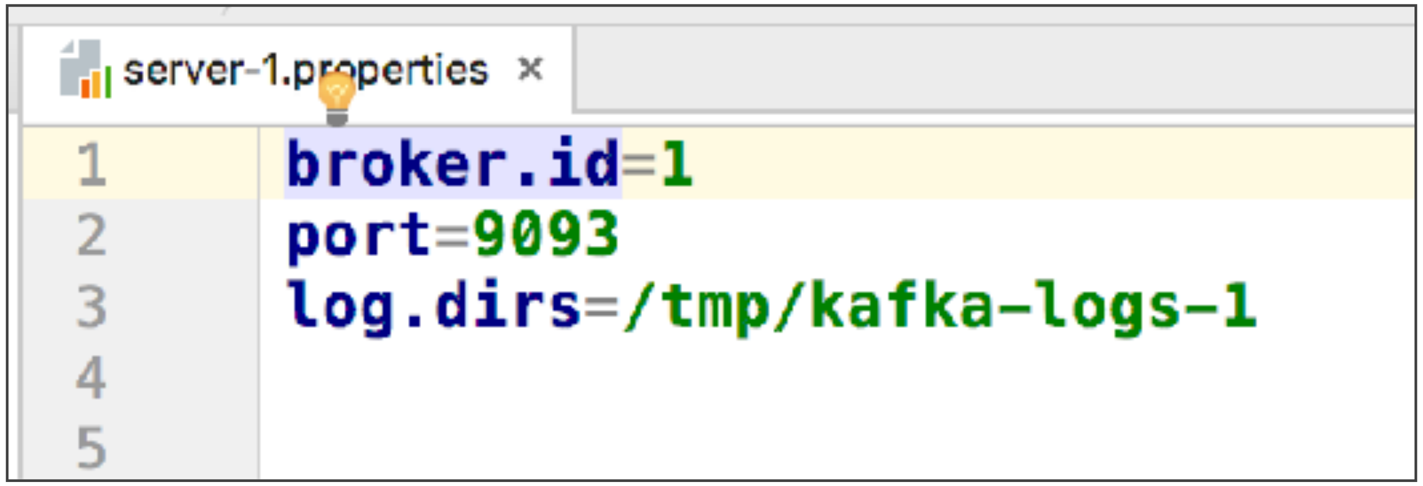


```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-server-start.sh kafka/config/server.properties

```

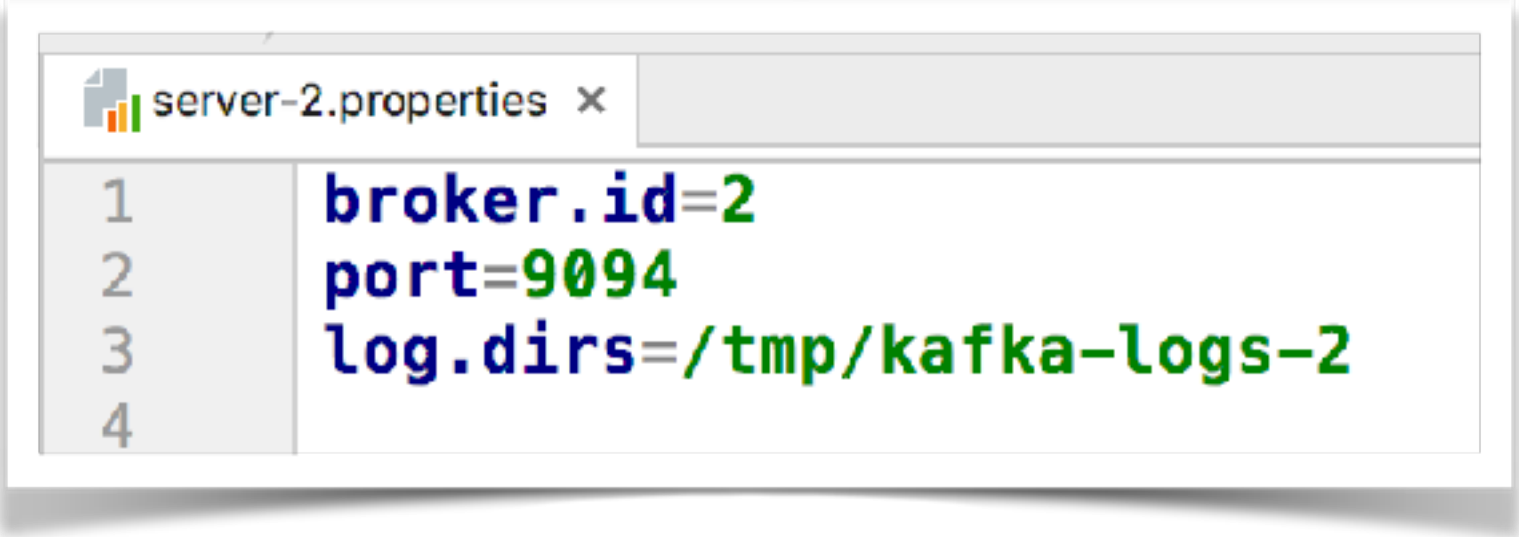
Create two new server.properties files

- Copy existing *server.properties* to *server-1.properties, server-2.properties*
- Change *server-1.properties* to use *port 9093, broker id 1,* and *log.dirs "/tmp/kafka-logs-1"*
- Change *server-2.properties* to use *port 9094, broker id 2,* and *log.dirs "/tmp/kafka-logs-2"*

server-x.properties

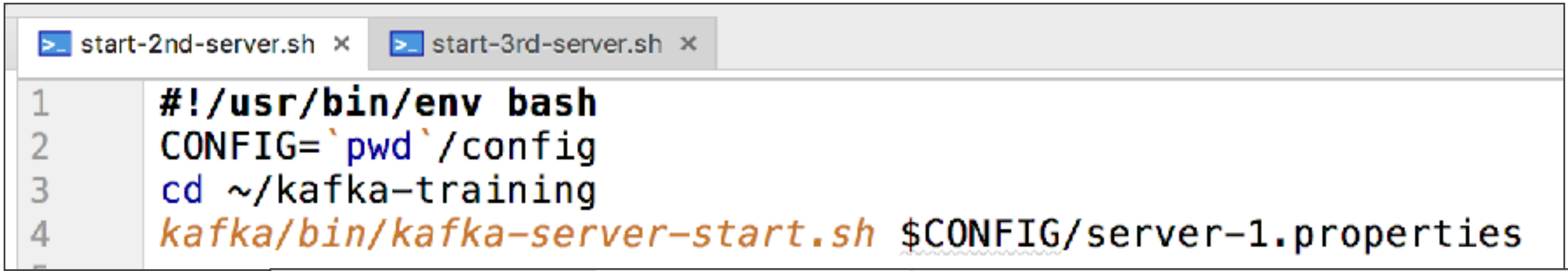


```
broker.id=1
port=9093
log.dirs=/tmp/kafka-logs-1
```

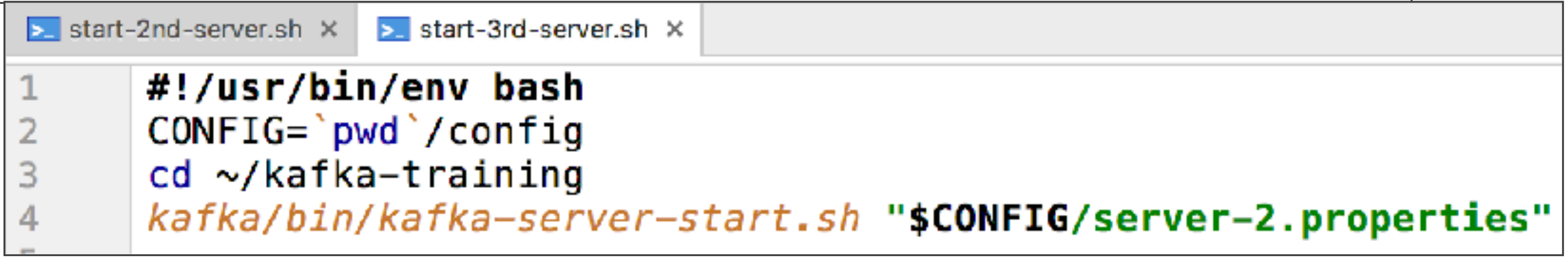


```
broker.id=2
port=9094
log.dirs=/tmp/kafka-logs-2
```

Start second and third servers

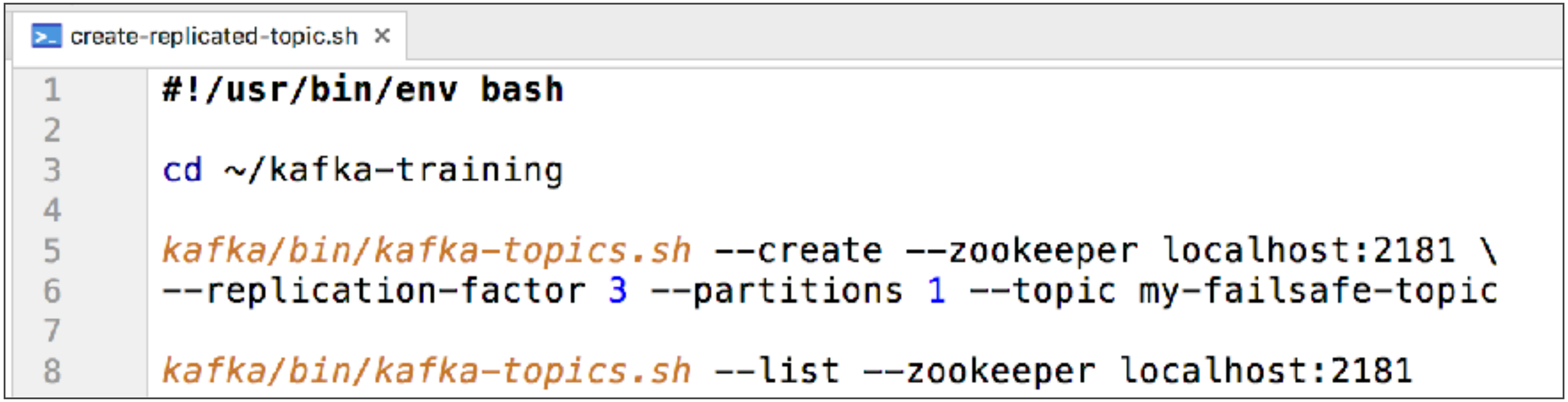


```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
kafka/bin/kafka-server-start.sh $CONFIG/server-1.properties
```



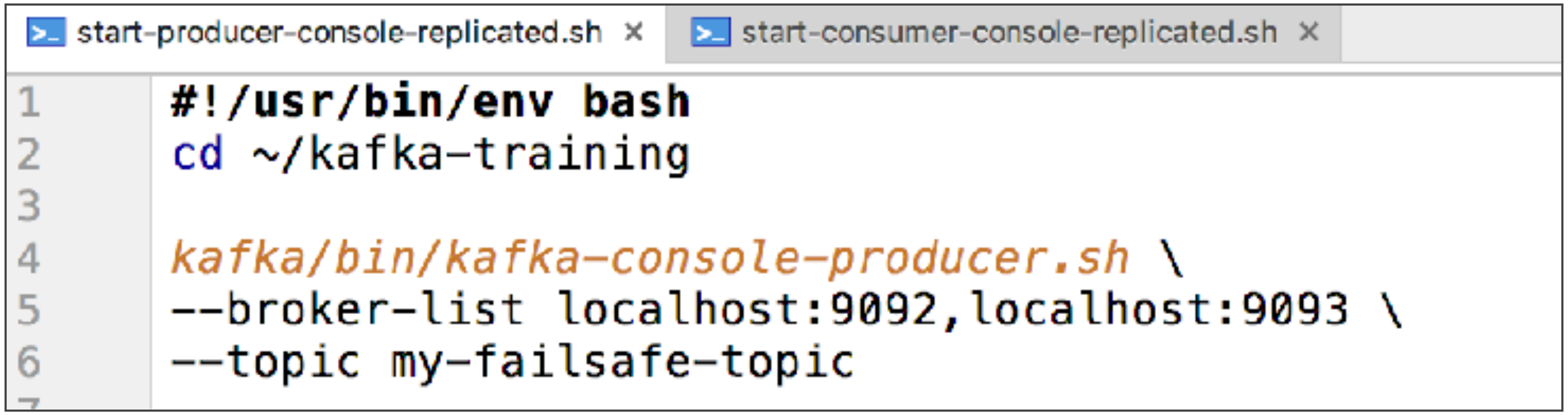
```
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
kafka/bin/kafka-server-start.sh "$CONFIG/server-2.properties"
```

Create Kafka replicated topic **my-failsafe-topic**

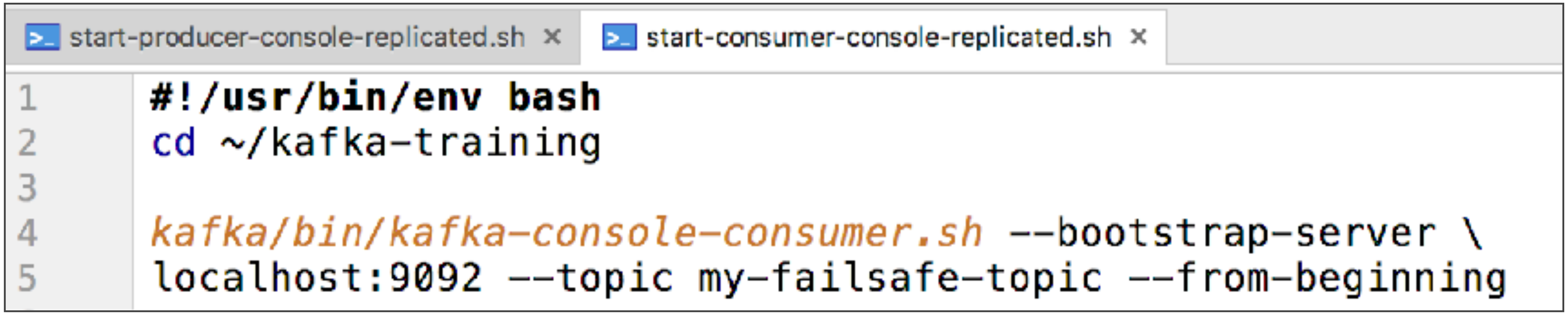


```
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
--replication-factor 3 --partitions 1 --topic my-failsafe-topic

kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Start Kafka consumer and producer

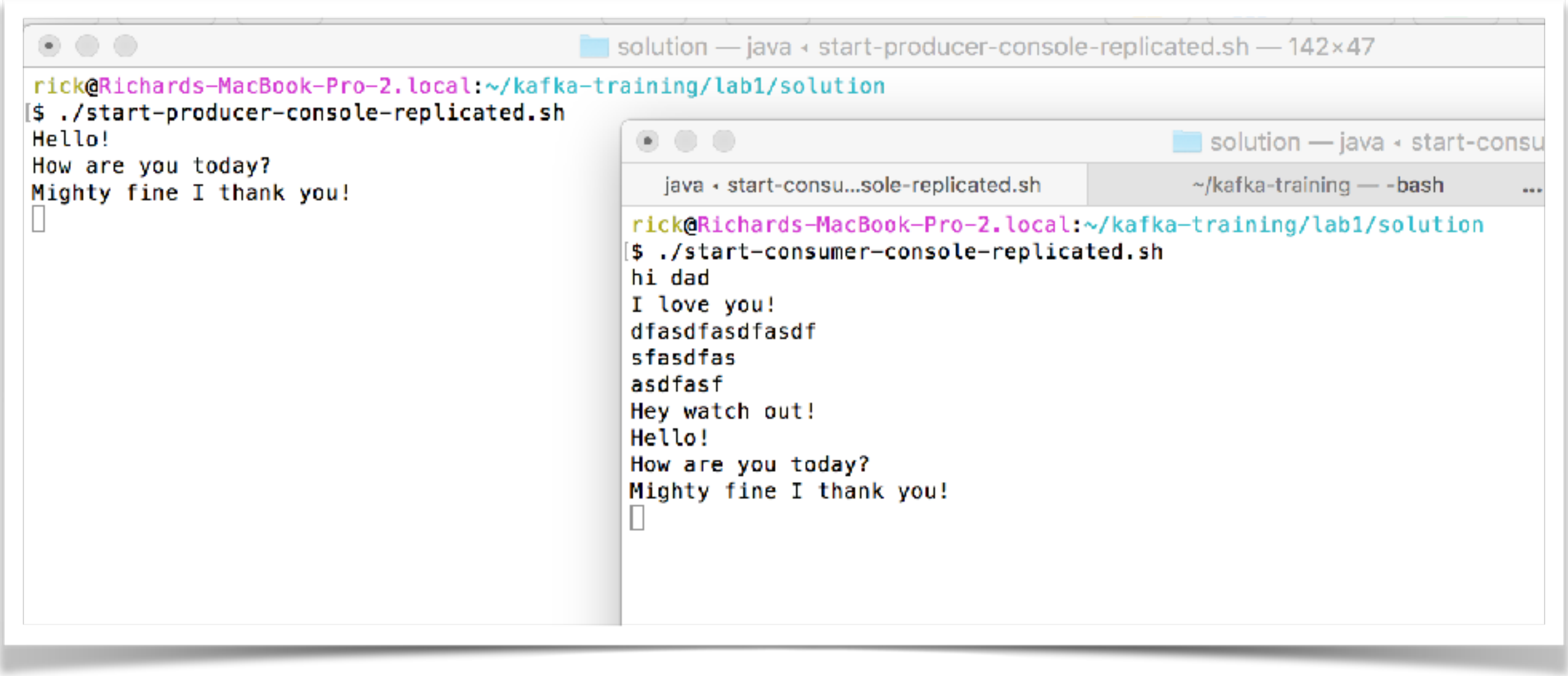

start-producer-console-replicated.sh ×
start-consumer-console-replicated.sh ×

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-producer.sh \
--broker-list localhost:9092,localhost:9093 \
--topic my-failsafe-topic
```


start-producer-console-replicated.sh ×
start-consumer-console-replicated.sh ×

```
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/kafka-console-consumer.sh --bootstrap-server \
localhost:9092 --topic my-failsafe-topic --from-beginning
```

Kafka consumer and producer running

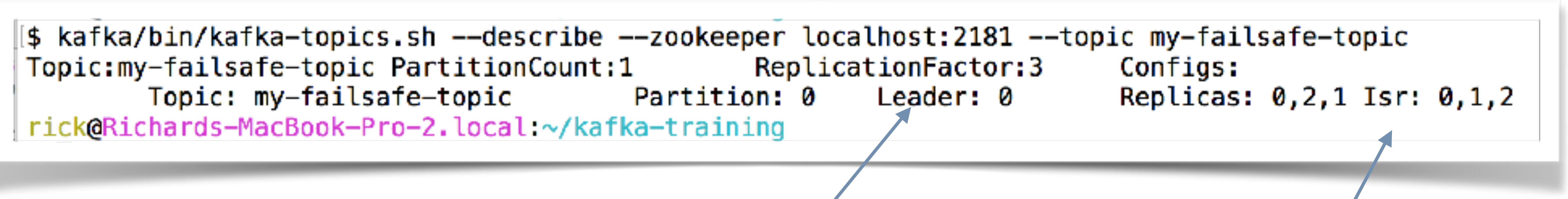

solution — java ◂ start-producer-console-replicated.sh — 142×47

```
rick@Richards-MacBook-Pro-2.local:~/kafka-training/lab1/solution
[$ ./start-producer-console-replicated.sh
Hello!
How are you today?
Mighty fine I thank you!
```

solution — java ◂ start-consu
java ◂ start-consu...sole-replicated.sh
~/kafka-training — -bash
...

```
rick@Richards-MacBook-Pro-2.local:~/kafka-training/lab1/solution
[$ ./start-consumer-console-replicated.sh
hi dad
I love you!
dfasdfasdfasdf
sfasdfas
asdfasf
Hey watch out!
Hello!
How are you today?
Mighty fine I thank you!
```

Use Kafka Describe Topic



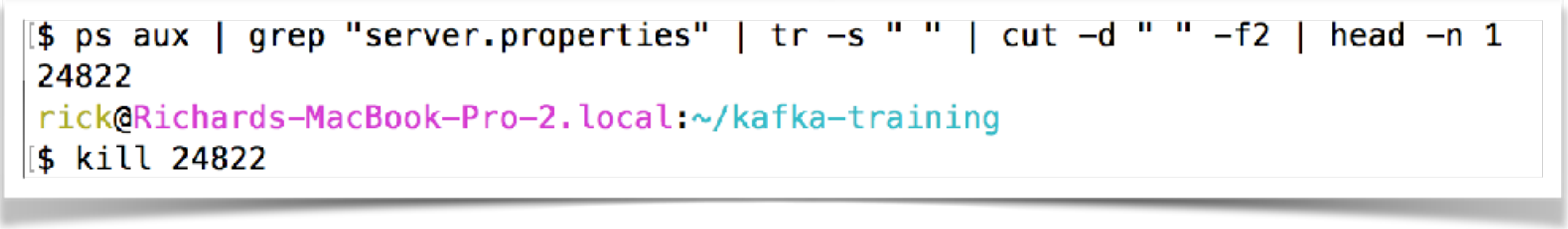
```
$ kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-failsafe-topic
Topic:my-failsafe-topic PartitionCount:1        ReplicationFactor:3     Configs:
        Topic: my-failsafe-topic        Partition: 0    Leader: 0       Replicas: 0,2,1 Isr: 0,1,2
rick@Richards-MacBook-Pro-2.local:~/kafka-training
```

The leader is broker 0

There is only one partition

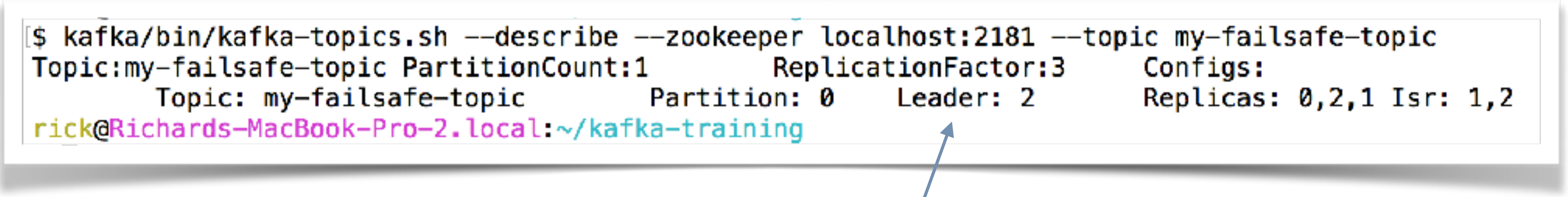
There are three in-sync replicas (ISR)

Test Failover by killing 1st server



```
[$ ps aux | grep "server.properties" | tr -s " " | cut -d " " -f2 | head -n 1
24822
rick@Richards-MacBook-Pro-2.local:~/kafka-training
[$ kill 24822
```

Use Kafka topic describe to see that a new leader was elected!



```
[$ kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-failsafe-topic
Topic:my-failsafe-topic PartitionCount:1        ReplicationFactor:3     Configs:
        Topic: my-failsafe-topic        Partition: 0    Leader: 2       Replicas: 0,2,1 Isr: 1,2
rick@Richards-MacBook-Pro-2.local:~/kafka-training
```

NEW LEADER IS 2!

Use Kafka to send and receive messages

Lab 2-A Use Kafka

Use a Kafka Cluster to replicate a Kafka topic log

Kafka Consumer and Producers

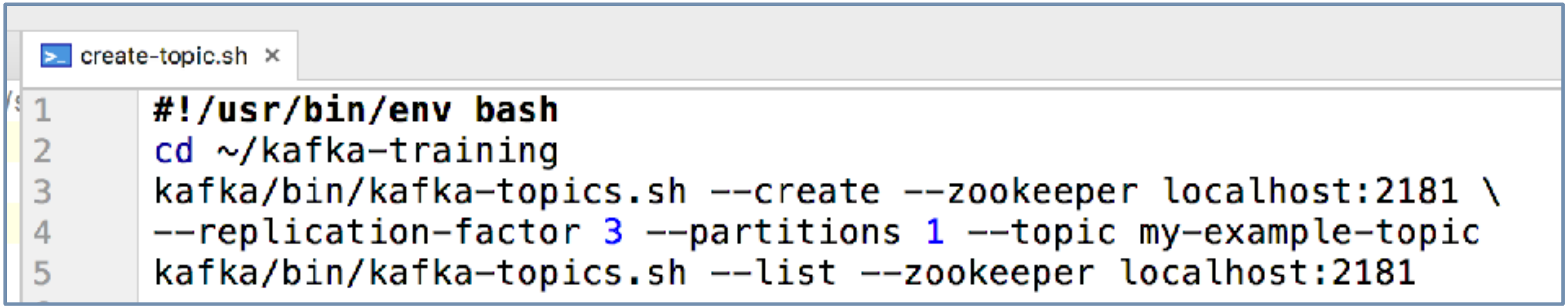
Working with producers and consumers

Step by step first example

Objectives Create Producer and Consumer example

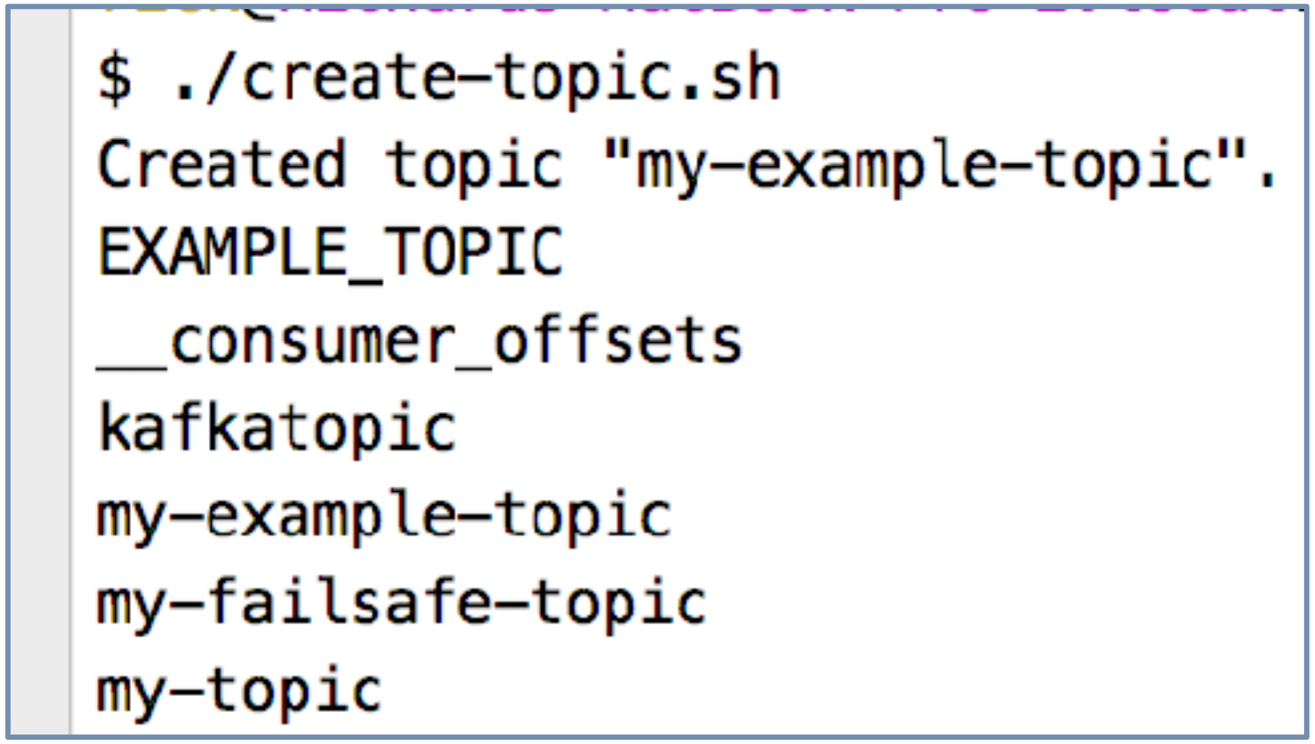
- Create simple example that creates a ***Kafka Consumer*** and a ***Kafka Producer***
- Create a new replicated ***Kafka topic***
- ***Create Producer*** that uses topic to send records
- ***Send records*** with ***Kafka Producer***
- ***Create Consumer*** that uses topic to receive messages
- ***Process messages*** from Kafka with ***Consumer***

Create Replicated Kafka Topic



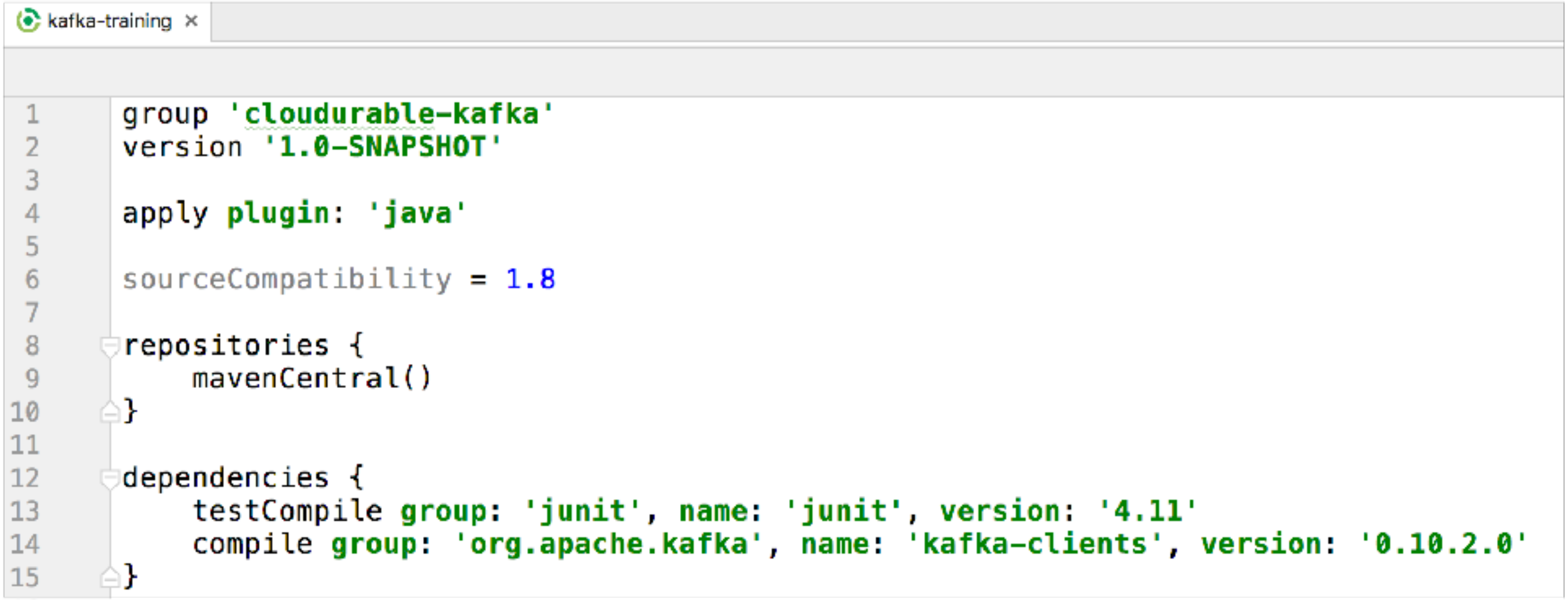
create-topic.sh ×

```
#!/usr/bin/env bash
cd ~/kafka-training
kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 \
--replication-factor 3 --partitions 1 --topic my-example-topic
kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
```



```
$ ./create-topic.sh
Created topic "my-example-topic".
EXAMPLE_TOPIC
__consumer_offsets
kafkatopic
my-example-topic
my-failsafe-topic
my-topic
```

Build script

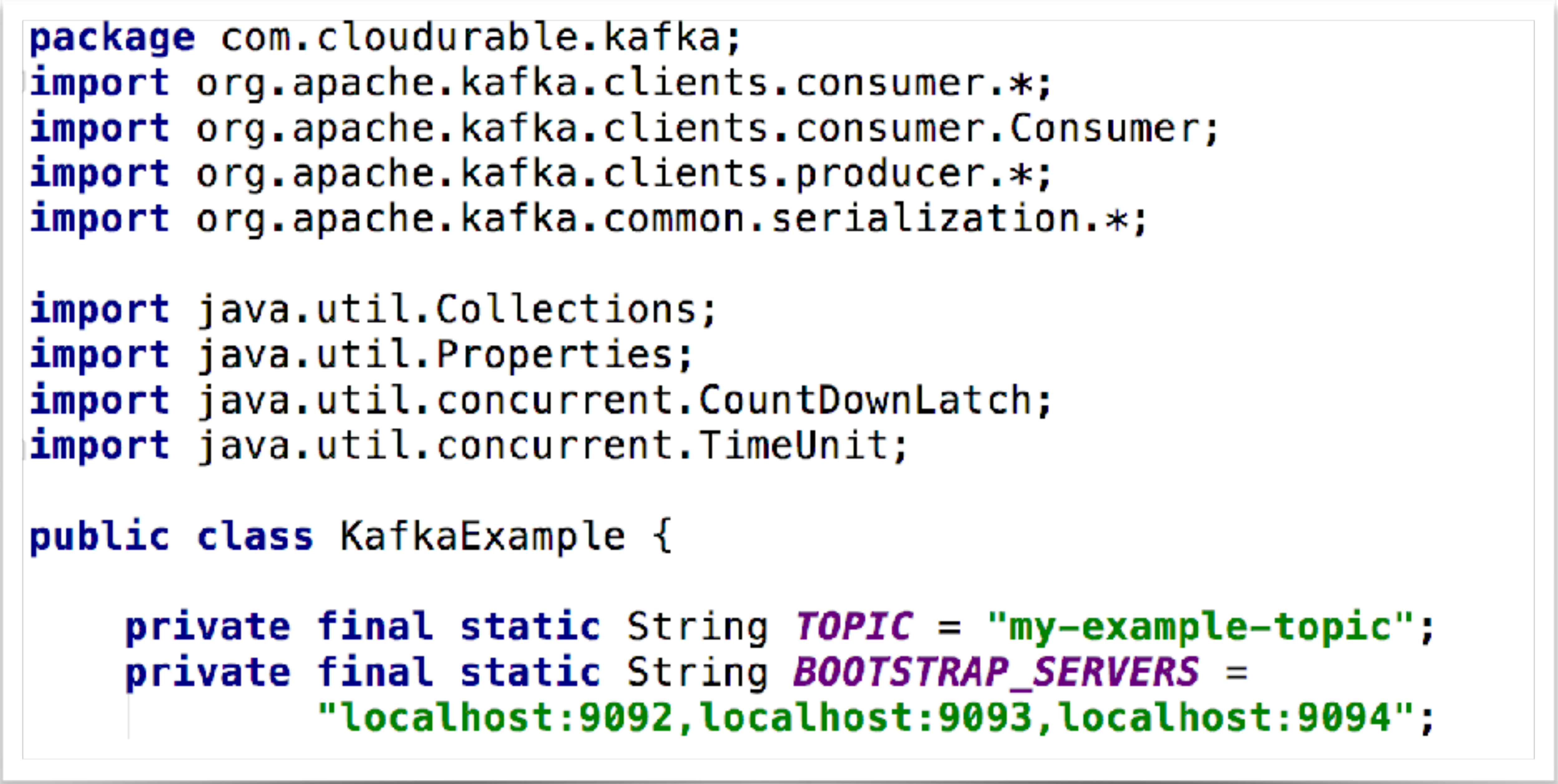


```
group 'cloudurable-kafka'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    compile group: 'org.apache.kafka', name: 'kafka-clients', version: '0.10.2.0'
}
```

Create Kafka Producer to send records

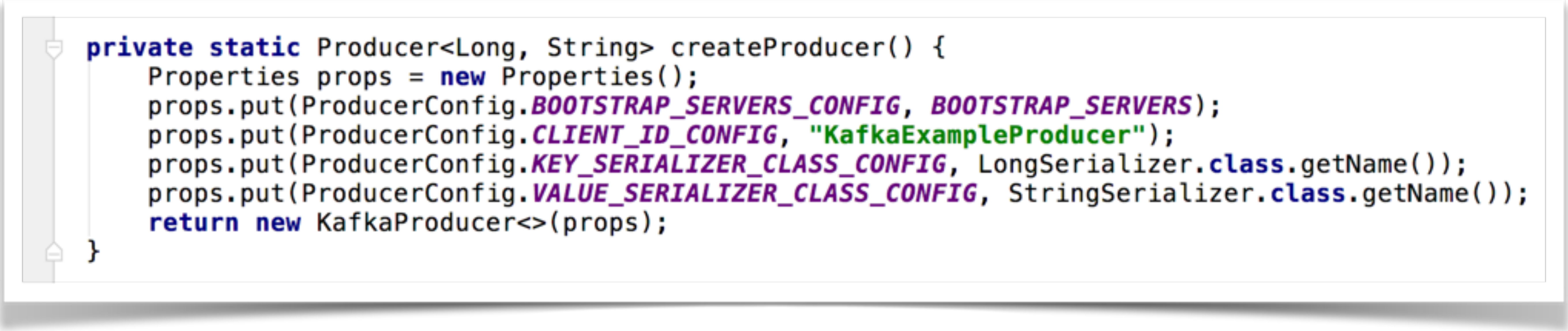
- Specify bootstrap servers
- Specify client.id
- Specify Record Key serializer
- Specify Record Value serializer

Common Kafka imports and constants



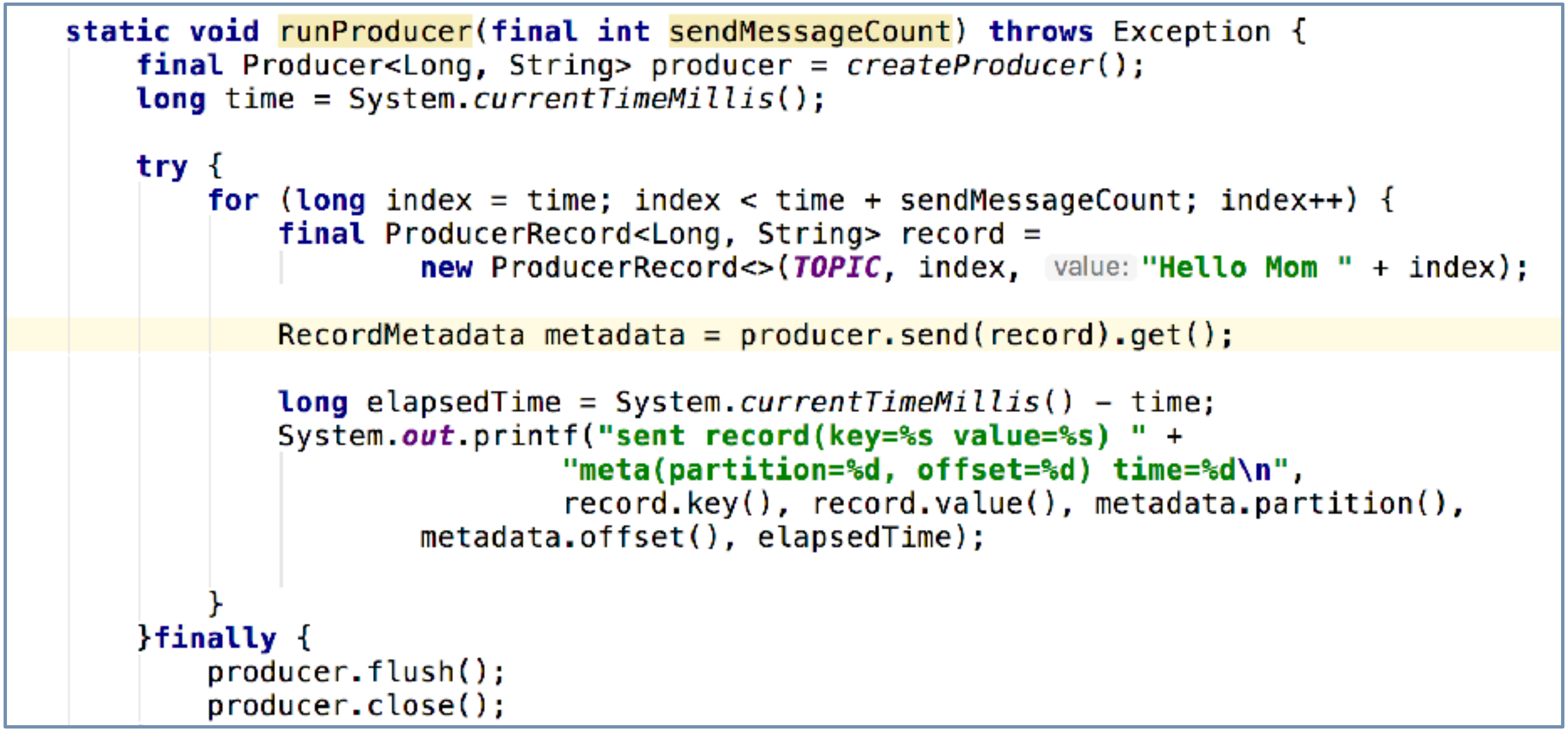
```
package com.cloudurable.kafka;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

public class KafkaExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";
```

Create Kafka Producer to send records



```java
private static Producer<Long, String> createProducer() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
    return new KafkaProducer<>(props);
}
```

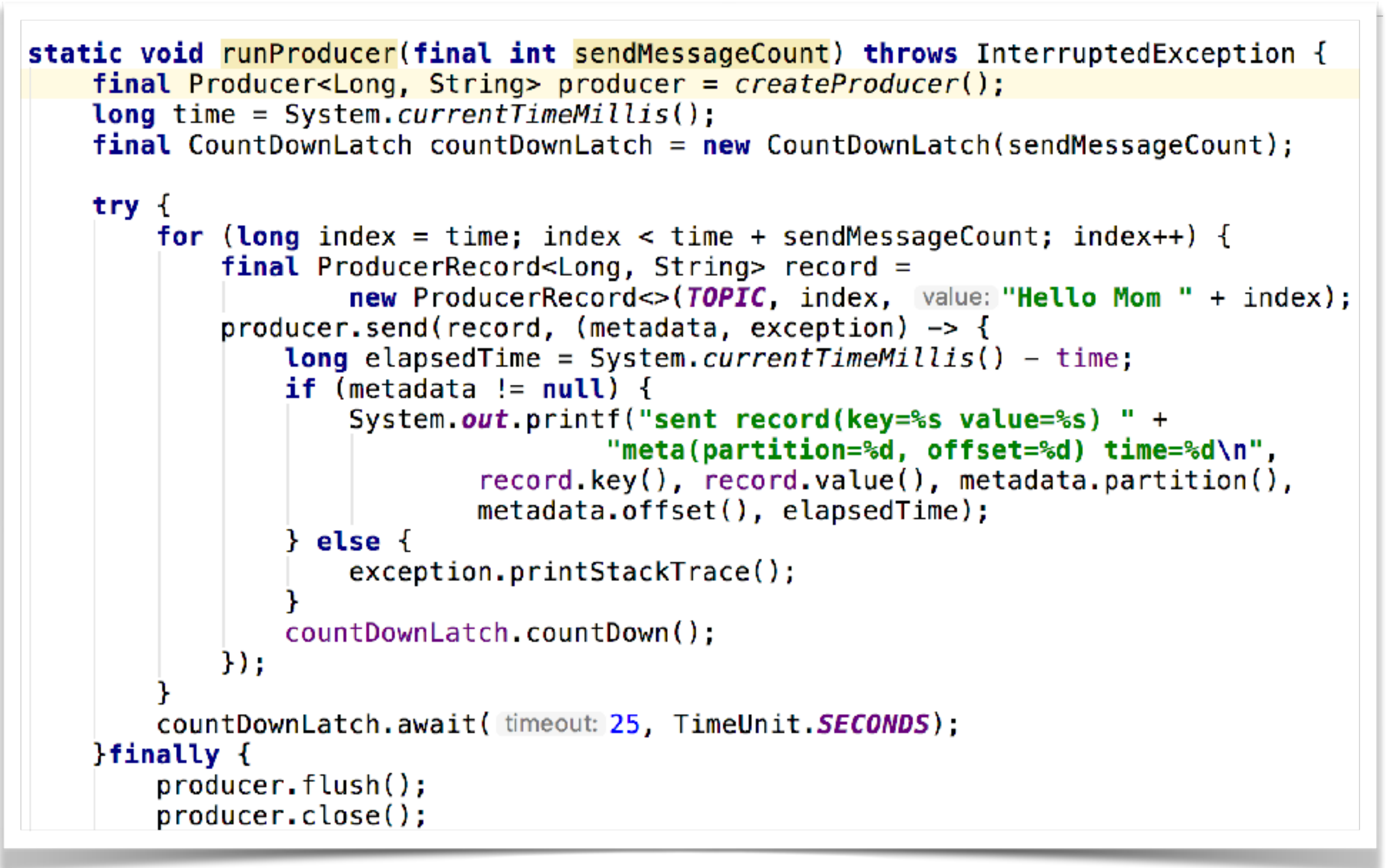
Send sync records with Kafka Producer



```
static void runProducer(final int sendMessageCount) throws Exception {
    final Producer<Long, String> producer = createProducer();
    long time = System.currentTimeMillis();

    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            final ProducerRecord<Long, String> record =
                    new ProducerRecord<>(TOPIC, index,  value: "Hello Mom " + index);

            RecordMetadata metadata = producer.send(record).get();

            long elapsedTime = System.currentTimeMillis() - time;
            System.out.printf("sent record(key=%s value=%s) " +
                            "meta(partition=%d, offset=%d) time=%d\n",
                            record.key(), record.value(), metadata.partition(),
                    metadata.offset(), elapsedTime);

        }
    }finally {
        producer.flush();
        producer.close();
```

The response **RecordMetadata** has 'partition' where record was written and the 'offset' of the record.

Send async records with Kafka Producer

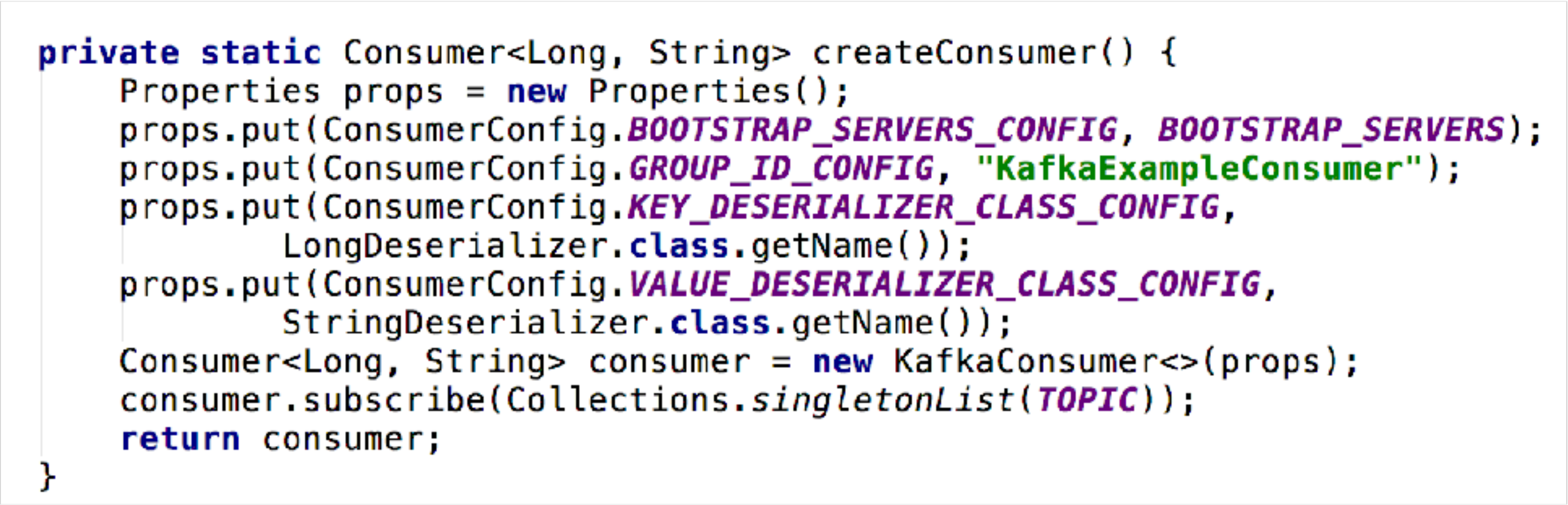


```
static void runProducer(final int sendMessageCount) throws InterruptedException {
    final Producer<Long, String> producer = createProducer();
    long time = System.currentTimeMillis();
    final CountDownLatch countDownLatch = new CountDownLatch(sendMessageCount);

    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            final ProducerRecord<Long, String> record =
                    new ProducerRecord<>(TOPIC, index,  value: "Hello Mom " + index);
            producer.send(record, (metadata, exception) -> {
                long elapsedTime = System.currentTimeMillis() - time;
                if (metadata != null) {
                    System.out.printf("sent record(key=%s value=%s) " +
                                    "meta(partition=%d, offset=%d) time=%d\n",
                            record.key(), record.value(), metadata.partition(),
                            metadata.offset(), elapsedTime);
                } else {
                    exception.printStackTrace();
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await( timeout: 25, TimeUnit.SECONDS);
    }finally {
        producer.flush();
        producer.close();
```

Create Consumer using Topic to Receive Records

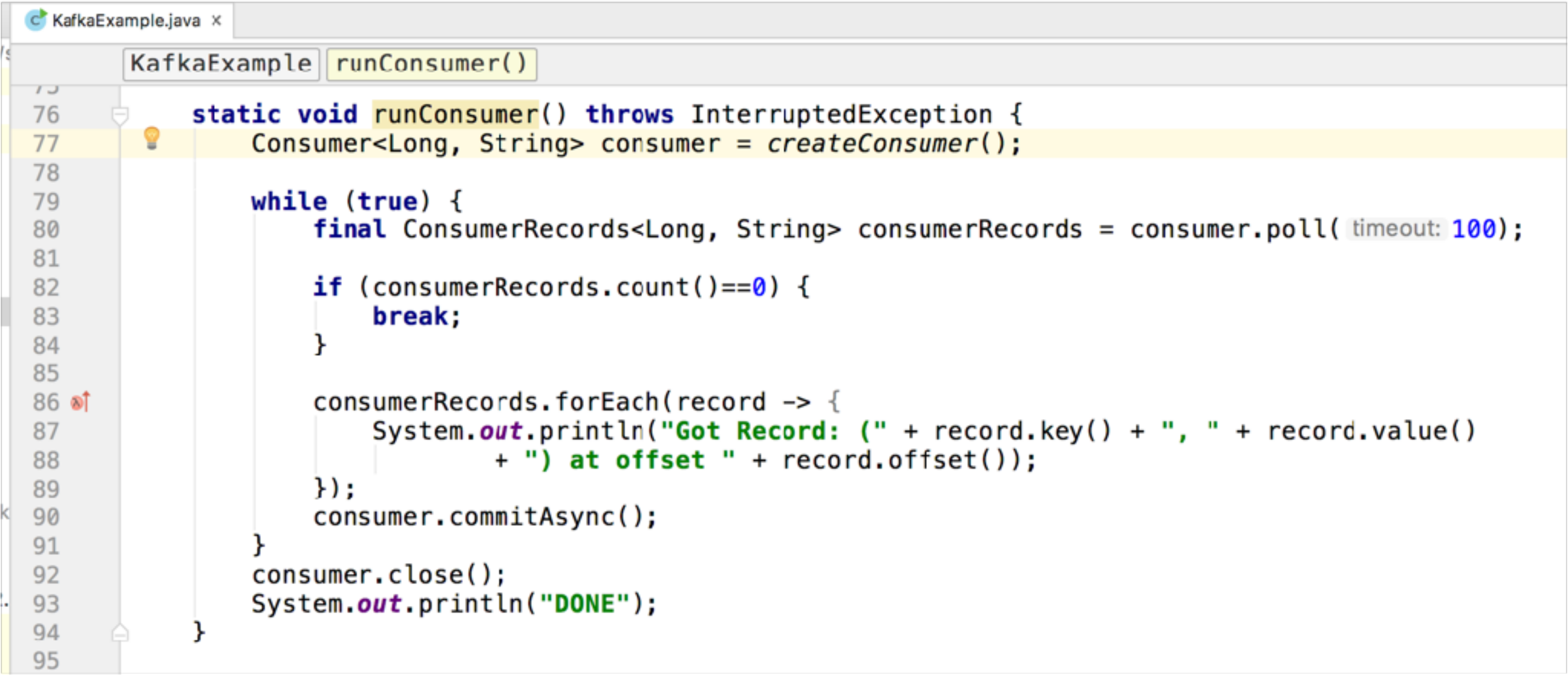
- Specify bootstrap servers
- Specify client.id
- Specify Record Key deserializer
- Specify Record Value deserializer
- Specify Consumer Group
- Subscribe to Topic

Create Consumer using Topic to Receive Records



```java
private static Consumer<Long, String> createConsumer() {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleConsumer");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            LongDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());
    Consumer<Long, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Collections.singletonList(TOPIC));
    return consumer;
}
```

Process messages from Kafka with Consumer

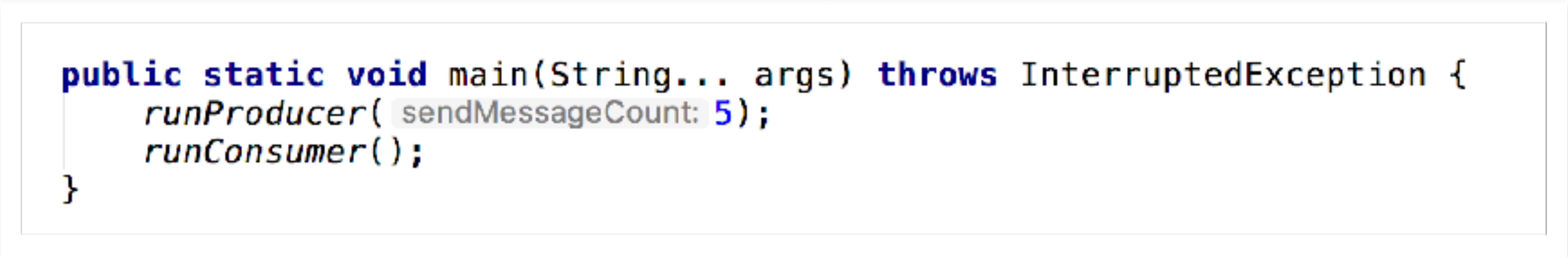


```java
    static void runConsumer() throws InterruptedException {
        Consumer<Long, String> consumer = createConsumer();

        while (true) {
            final ConsumerRecords<Long, String> consumerRecords = consumer.poll( timeout: 100);

            if (consumerRecords.count()==0) {
                break;
            }

            consumerRecords.forEach(record -> {
                System.out.println("Got Record: (" + record.key() + ", " + record.value()
                        + ") at offset " + record.offset());
            });
            consumer.commitAsync();
        }
        consumer.close();
        System.out.println("DONE");
    }
```

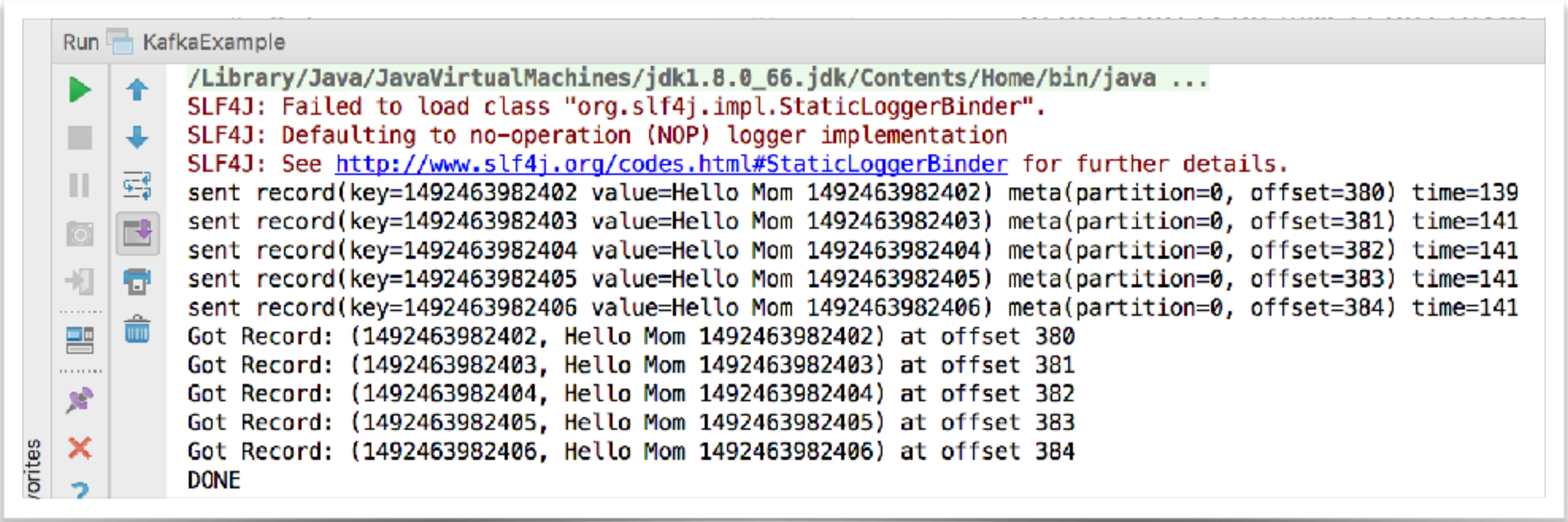
Consumer poll

- poll() method returns fetched records based on current partition offset
- Blocking method waiting for specified time if no records available
- When/If records available, method returns straight away
- Control the maximum records returned by the poll() with `props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);`
- poll() is not meant to be called from multiple threads

Running both Consumer and Producer



```
public static void main(String... args) throws InterruptedException {
    runProducer( sendMessageCount: 5);
    runConsumer();
}
```



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
sent record(key=1492463982402 value=Hello Mom 1492463982402) meta(partition=0, offset=380) time=139
sent record(key=1492463982403 value=Hello Mom 1492463982403) meta(partition=0, offset=381) time=141
sent record(key=1492463982404 value=Hello Mom 1492463982404) meta(partition=0, offset=382) time=141
sent record(key=1492463982405 value=Hello Mom 1492463982405) meta(partition=0, offset=383) time=141
sent record(key=1492463982406 value=Hello Mom 1492463982406) meta(partition=0, offset=384) time=141
Got Record: (1492463982402, Hello Mom 1492463982402) at offset 380
Got Record: (1492463982403, Hello Mom 1492463982403) at offset 381
Got Record: (1492463982404, Hello Mom 1492463982404) at offset 382
Got Record: (1492463982405, Hello Mom 1492463982405) at offset 383
Got Record: (1492463982406, Hello Mom 1492463982406) at offset 384
DONE
```

Java Kafka simple example recap

- Created simple example that creates a ***Kafka Consumer*** and a ***Kafka Producer***
- Created a new replicated ***Kafka topic***
- ***Created Producer*** that uses topic to send records
- ***Send records*** with ***Kafka Producer***
- ***Created Consumer*** that uses topic to receive messages
- ***Processed records*** from Kafka with ***Consumer***

Kafka design

Design discussion of Kafka

Kafka Design Motivation

- Kafka unified platform for handling real-time data feeds/streams
- High-throughput supports high volume event streams like log aggregation
- Must support real-time analytics
  - real-time processing of streams to create new, derived streams
  - inspired partitioning and consumer model
- Handle large data backlogs - periodic data loads from offline systems
- Low-latency delivery to handle traditional messaging use-cases
- Scale writes and reads via partitioned, distributed, commit logs
- Fault-tolerance for machine failures
- Kafka design is more like database transaction log than a traditional messaging system

Persistence: Embrace filesystem

- Kafka relies heavily on filesystem for storing and caching messages/records
- Disk performance of hard drives performance of sequential writes is fast
  - JBOD configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec
  - Sequential reads and writes are predictable, and are heavily optimized by operating systems
  - Sequential disk access can be faster than random memory access and SSD
- Operating systems use available of main memory for disk caching
- JVM GC overhead is high for caching objects whilst OS file caches are almost free
- Filesystem and relying on page-cache is preferable to maintaining an in-memory cache in the JVM
- By relying on the OS page cache Kafka greatly simplifies code for cache coherence
- Since Kafka disk usage tends to do sequential reads the read-ahead cache of the OS pre-populating its page-cache

Cassandra, Netty, and Varnish use similar techniques.
The above is explained well in the Kafka Documentation.
And there is a more entertaining explanation at the Varnish site.

Long sequential disk access

- Like Cassandra, LevelDB, RocksDB, and others Kafka uses a form of log structured storage and compaction instead of an on-disk mutable BTree
- Kafka uses tombstones instead of deleting records right away
- Since disks these days have somewhat unlimited space and are very fast, Kafka can provide features not usually found in a messaging system like holding on to old messages for a really long time
    - This flexibility allows for interesting application of Kafka

Kafka compression

- Kafka provides ***End-to-end Batch Compression***
- Bottleneck is not always CPU or disk but often network bandwidth
  - especially in cloud and virtualized environments
  - especially when talking datacenter to datacenter or WAN
- Instead of compressing records one at a time…
- Kafka enable efficient compression of a whole batch or a whole message set or message batch
- Message batch can be compressed and sent to Kafka broker/server in one go
- Message batch will be written in compressed form in log partition
  - don't get decompressed until they consumer
- GZIP, Snappy and LZ4 compression protocols supported

Read more at Kafka documents on end to end compression.

Kafka Producer Load Balancing

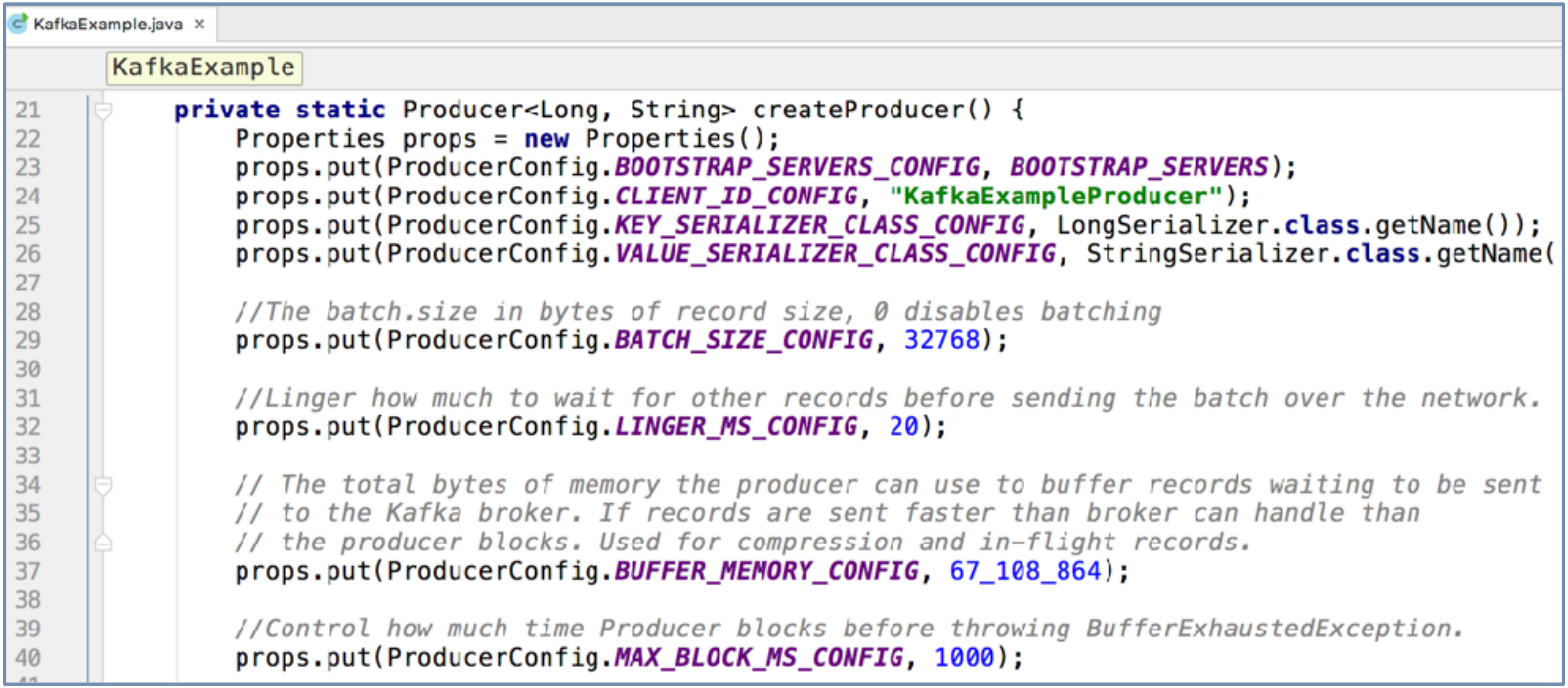
- Producer sends records directly to Kafka broker partition leader
- Producer asks Kafka broker for metadata about which Kafka broker has which topic partitions leaders - thus no routing layer needed
- Producer client controls which partition it publishes messages to
- Partitioning can be done by key, round-robin or using a custom semantic partitioner

Kafka Producer Record Batching

- Kafka producers support record batching
- Batching is good for efficient compression and network IO throughput
- Batching can be configured by size of records in bytes in batch
- Batches can be auto-flushed based on time
- See code example on the next slide
- Batching allows accumulation of more bytes to send, which equate to few larger I/O operations on Kafka Brokers and increase compression efficiency
- Buffering is configurable and lets you make a tradeoff between additional latency for better throughput
  - Or in the case of an heavily used system, it could be both better average throughput and

QBit a microservice library uses message batching in an identical fashion as Kafka to send messages over WebSocket between nodes and from client to QBit server.

More producer settings for performance



```java
    private static Producer<Long, String> createProducer() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(

        //The batch.size in bytes of record size, 0 disables batching
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);

        //Linger how much to wait for other records before sending the batch over the network.
        props.put(ProducerConfig.LINGER_MS_CONFIG, 20);

        // The total bytes of memory the producer can use to buffer records waiting to be sent
        // to the Kafka broker. If records are sent faster than broker can handle than
        // the producer blocks. Used for compression and in-flight records.
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67_108_864);

        //Control how much time Producer blocks before throwing BufferExhaustedException.
        props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 1000);
```

For higher throughput, Kafka Producer allows buffering based on time and size.
Multiple records can be sent as a batches with fewer network requests.
Speeds up throughput drastically.

Stay tuned

- More to come

References

- **Learning Apache Kafka**, Second Edition 2nd Edition by Nishant Garg (Author), 2015, ISBN 978-1784393090, Packet Press
- ***Apache Kafka Cookbook***, 1st Edition, Kindle Edition by Saurabh Minni (Author), 2015, ISBN 978-1785882449, Packet Press
- Kafka Streams for Stream processing: A few words about how Kafka works, Serban Balamaci, 2017, ***Blog: Plain Ol' Java***
- Kafka official documentation, 2017