Presents

Kafka Architecture

Kafka Command Utilities in BIN

- Starting Kafka brokers
  - ✓ bin/kafka-server-start
  - ✓ bin/kafka-server-stop
- Managing topics
  - ✓ bin/kafka-topics: Lists / create / delete topics
- Sending Messages
  - ✓ bin/kafka-console-producer.sh
- Consuming messages
  - ✓ bin/kafka-console-consumer.sh

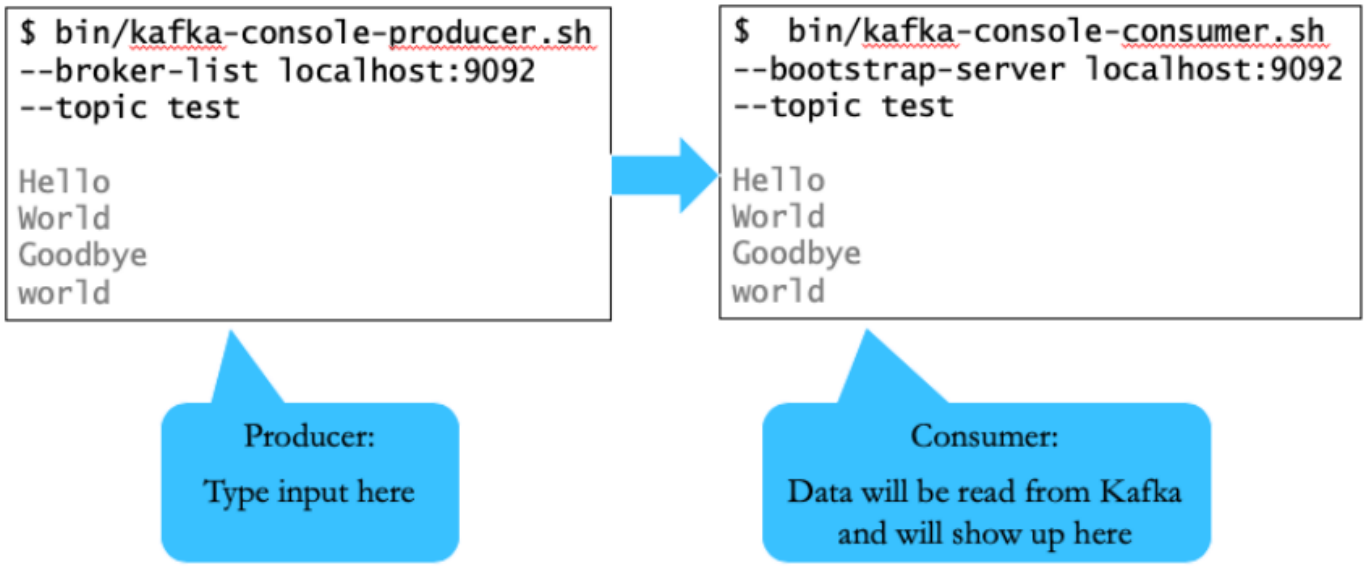
Creating Topics

```
$   bin/kafka-topics.sh  --bootstrap-server localhost:9092 --list
# ... empty ...

## create a topic with one replica and two partitions
$  bin/kafka-topics.sh  --bootstrap-server localhost:9092  --create
--topic test --replication-factor 1  --partitions 2


$  bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic test

# Topic:test    PartitionCount:2    ReplicationFactor:1    Configs:
# Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
# Topic: test    Partition: 1    Leader: 0    Replicas: 0    Isr: 0
```

LearnQuest

Using Producer / Consumer Utils

- bin/ kafka-console-producer:
  - ✓ utility for producing messages
- bin/kafka-console-consumer:
  - ✓ utility for reading messages



Kafka Clients

- Java is the 'first class' citizen in Kafka
  - ✓ Officially maintained
- Python on par with Java
  - ✓ Maintained by Confluent.io
- Other language libraries are independently developed
  - ✓ May not have 100% coverage
  - ✓ May not be compatible with latest versions of Kafka

Kafka Java API

- Rich library that provides high level abstractions
  - ✓ No need to worry about networking / data format ..etc
- Write message / Read message
- Supports native data types
  - ✓ String
  - ✓ Bytes
  - ✓ Primitives (int, long ...etc.)

Java Producer Code (Abbreviated)

```
// ** 1 **
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.common.serialization.IntegerSerializer;
...

// ** 2 **
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "SimpleProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer< Integer, String > producer = new KafkaProducer<>(props);

// ** 3 **
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord < Integer, String > record = new ProducerRecord<> (topic, key, value);
producer.send(record);
producer.close();
```

Producer Code Walkthrough

```
// ** 2 **  Recommended approach: use constants

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.common.serialization.IntegerSerializer

Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "SimpleProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);
```

```
// ** 2 ** another approach
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("key.serializer",  "org.apache.kafka.common.serialization.IntegerSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);
```

LearnQuest

Producer Code Walkthrough

- Each record represents a message
  - ✓ Here we have a <key,value> message
  - ✓ send() doesn't wait for confirmation
- We send in batches
  - ✓ For increased throughput
  - ✓ Minimize network round trips

```
// ** 3 **
String topic = "test";
Integer key = new Integer(1);
String value = "Hello world";
ProducerRecord< Integer, String > record = new ProducerRecord<> (topic, key, value);
producer.send(record);
producer.close();
```

LearnQuest

Producer Properties

```
Properties props = new Properties();
props.put("boostrap.servers", "localhost:9092");
props.put("client.id", "SimpleProducer");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);  // 16k
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432); // 32 M
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

KafkaProducer < Integer, String > producer = new KafkaProducer<>(props);

for(int i = 0; i < 100; i++) {
  producer.send(new ProducerRecord < String, String >(
      "my-topic", Integer.toString(i), Integer.toString(i)));
}
producer.close();
```

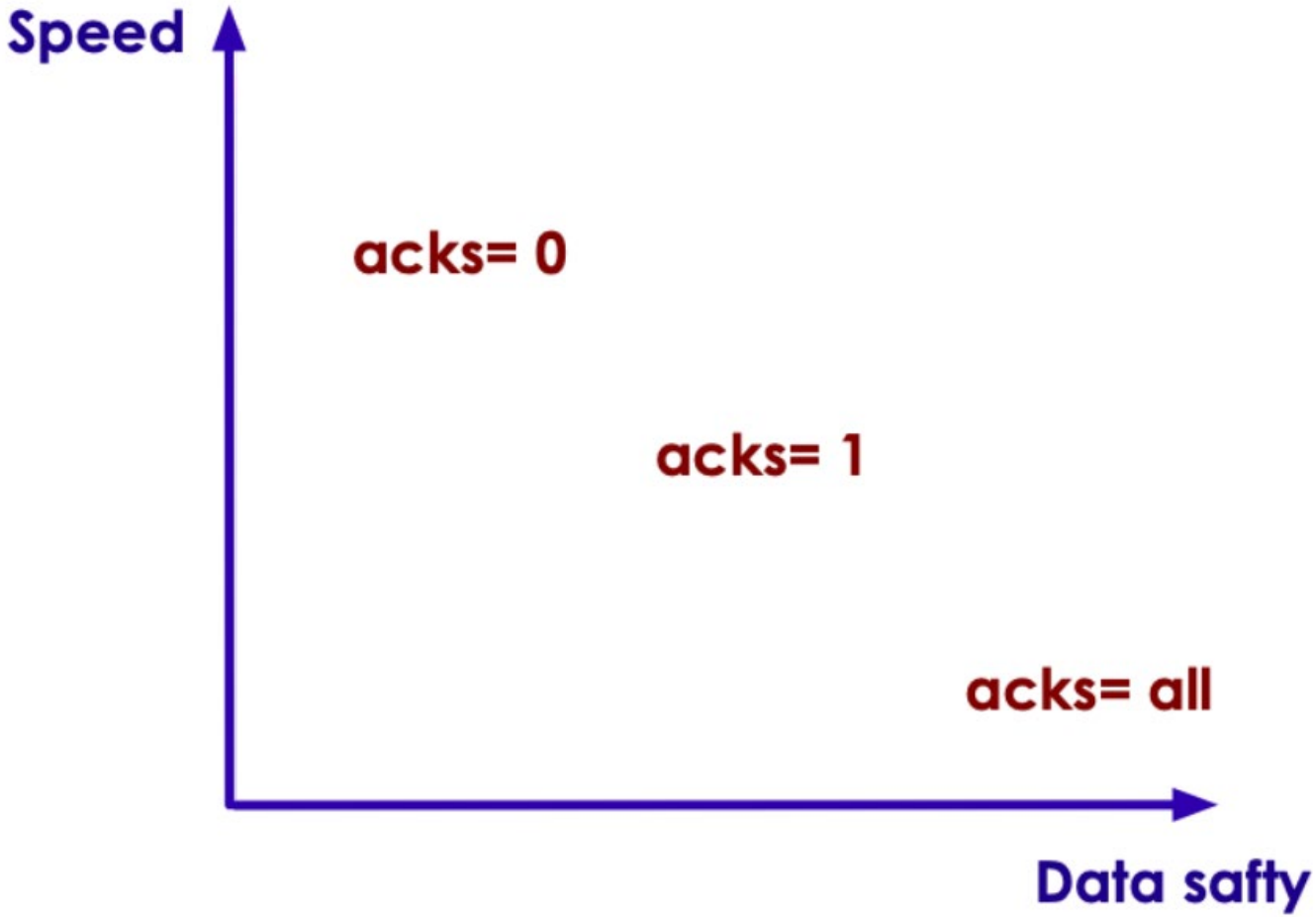
LearnQuest

Producer Acknowledgements

ACK	Description	Speed	Data safety
acks=0	- Producer doesn't wait for any acks from broker, - Producer won't know of any errors	High	Low No guarantee that broker received the message
acks=1, **(default)**	- Broker will write the message to local log, - Does not wait for replicas to complete	Medium	Medium Message is at least persisted on lead broker
acks=all	- Message is persisted on lead broker and in replicas, - Lead broker will wait for in-sync replicas to acknowledge the write	Low	High Message is persisted in multiple brokers

LearnQuest

Producer Acknowledgements



Consumer Code (Abbreviated)

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.IntegerDeSerializer

...

Properties props = new Properties(); // ** 1 **
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "Simple Consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeSerializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());


KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("topic1")); // ** 2 **

try {
    while (true) {
       ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000)); // ** 3 **
       System.out.println("Got " + records.count() + " messages");
       for (ConsumerRecord < Integer, String > record : records) {
          System.out.println("Received message : " + record);
       }
   }
}
finally {
    consumer.close(Duration.OfSeconds(60));
}
```

LearnQuest

Consumer Code Walkthrough

```
Properties props = new Properties(); // ** 1 **
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "Simple Consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeSerializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeSerializer.class.getName());


KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("topic1")); // ** 2 **
```

- bootstrap,servers: "broker1:9092,broker2:9092"
  - ✓ Connect to multiple brokers to avoid single point of failure
  - ✓ group.id: consumers belong in a Consumer Group
  - ✓ We are using standard serializers
- Consumers can subscribe to one or more subjects // ** 2 **

LearnQuest

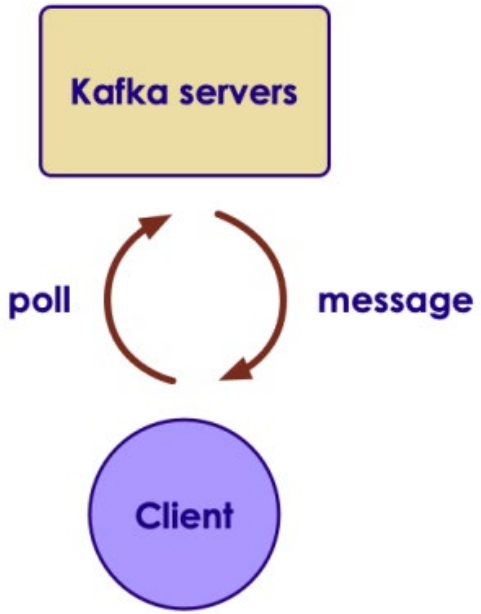
Consumer Code Walkthrough

```
try {
   while (true) {
      ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000)); // ** 3 **
      System.out.println("Got " + records.count() + " messages");
      for (ConsumerRecord < Integer, String > record : records) {
        System.out.println("Received message : " + record);
      }
   }
}
finally {
   consumer.close();
}
```

- Consumers must subscribe to topics before starting polling
  - ✓ Consumer.subscribe ("test.*") // wildcard subscribe
  - ✓ Poll: This call will return in 1000 ms, with or without records
  - ✓ Must keep polling, otherwise consumer is deemed dead and the partition is handed off to another consumer

Consumer Poll Loop

- Polling is usually done in an infinite loop.
  - ✓ First time poll is called
  - ✓ Finds the GroupCoordinator
  - ✓ Joining Consumer Group
  - ✓ Receiving partition assignment
- Work done in poll loop
  - ✓ Usually involves some processing
  - ✓ Saving data to a store
  - ✓ Don't do high latency work between polls; otherwise the consumer could be deemed dead.
  - ✓ Do heavy lifting in a separate thread



LearnQuest

ConsumerRecord

- org.apache.kafka.clients.consumer.ConsumerRecord <K,V>
- K key(): key for record (type K), can be null
- V value(): record value (type V - String / Integer ..etc)
- String topic(): Topic where this record came from
- int partition(): partition number
- long offset(): long offset in

```
ConsumerRecords < Integer, String > records = consumer.poll(Duration.ofMillis(1000));
  for (ConsumerRecord < Integer, String > record : records) {
     System.out.printf("topic = %s, partition = %d, offset = %d,
            key= %s, value = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
  }
```

LearnQuest

Configuring Consumers

- max.partition.fetch.bytes (default : 1048576 (1M))
  - ✓ Max message size to fetch. Also see message.max.bytes broker config
- session.timeout.ms (default : 30000 (30 secs))
  - ✓ If no heartbeats are not received by this window, consumer will be deemed dead and a partition rebalance will be triggered

```
Properties props = new Properties(); // ** 1 **

...
props.put("session.timeout.ms", 30000); // 30 secs
props.put("max.partition.fetch.bytes", 5 * 1024 * 1024); // 5 M

KafkaConsumer < Integer, String > consumer = new KafkaConsumer<>(props);
```

Clean Shutdown Of Consumers

- Consumers poll in a tight, infinite loop
- Call ' consumer.wakeup () ' from another thread
- This will cause the poll loop to exit with ‘ ‘WakeupException '

```
try {
  while (true) {
   ConsumerRecords < Integer, String > records = consumer.poll(100);
    // handle events
 }
}
catch (WakeupException ex) {
   // no special handling needed, just exit the poll loop
}
finally {
   // close will commit the offsets
   consumer.close();
}
```

LearnQuest

Signaling Consumer To Shutdown

- Can be done from another thread or shutdown hook
- ' consumer.wakeup () ' is safe to call from another thread

```
Runtime.getRuntime().addShutdownHook(new Thread() {
   public void run() {
      System.out.println("Starting exit...");
      consumer.wakeup(); // signal poll loop to exit
      try {
           mainThread.join(); // wait for threads to shutdown
      } catch (InterruptedException e) {
           e.printStackTrace();
  }
 }
}
```

Questions