

# **Apache Kafka Development**

Rajesh Pasham

# Apache Kafka - Installation Steps

## ▶ Step 1 - Verifying Java Installation

```
$ java -version
```

- If java is successfully installed on your machine, you could see the version of the installed Java.

# Apache Kafka - Installation Steps

- ▶ Step 2 - Apache Kafka Installation
  - Step 2.1 - Download Kafka
    - the latest version i.e., – **kafka\_2.12-3.1.0.tar.gz**
  - Step 2.2 - Extract the tar file

```
$ tar -zxf kafka_2.12-3.1.0.tar.gz
```

```
$ cd kafka_2.12-3.1.0
```

# Apache Kafka - Basic Operations


- ▶ First let us start implementing single node-single broker configuration and we will then migrate our setup to single node-multiple brokers configuration.
- ▶ Before moving to the Kafka Cluster Setup, first you would need to start your ZooKeeper because Kafka Cluster uses ZooKeeper.
- ▶ Start ZooKeeper  
`$bin/zookeeper-server-start.sh config/zookeeper.properties`

# Apache Kafka - Basic Operations

- ▶ To start Kafka Broker, type the following command –  
`$bin/kafka-server-start.sh config/server.properties`
- ▶ After starting Kafka Broker, type the command `jps` on ZooKeeper terminal and you would see the following response  
821 QuorumPeerMain  
928 Kafka  
931 Jps
- ▶ Now you could see two daemons running on the terminal where QuorumPeerMain is ZooKeeper daemon and another one is Kafka daemon.

# Apache Kafka - Basic Operations

## **Single Node-Multiple Brokers Configuration**

- ▶ Before moving on to the multiple brokers cluster setup, first start your ZooKeeper server.
  - ▶ Create Multiple Kafka Brokers - We have one Kafka broker instance already in `config/server.properties`.
  - ▶ Now we need multiple broker instances, so copy the existing `server.properties` file into two new config files and rename it as `server-one.properties` and `server-two.properties`.
- 

# Apache Kafka - Basic Operations

## Single Node-Multiple Brokers Configuration

- ▶ Then edit both new files and assign the following changes –
  - config/server-one.properties

# The id of the broker. This must be set to a unique integer for each broker.  
broker.id=1

# The port the socket server listens on  
port=9093

# A comma separated list of directories under which to store log files  
log.dirs=/tmp/kafka-logs-1

# Apache Kafka - Basic Operations

## Single Node-Multiple Brokers Configuration

### ► config/server-two.properties

# The id of the broker. This must be set to a unique integer for each broker.  
broker.id=2

# The port the socket server listens on  
port=9094

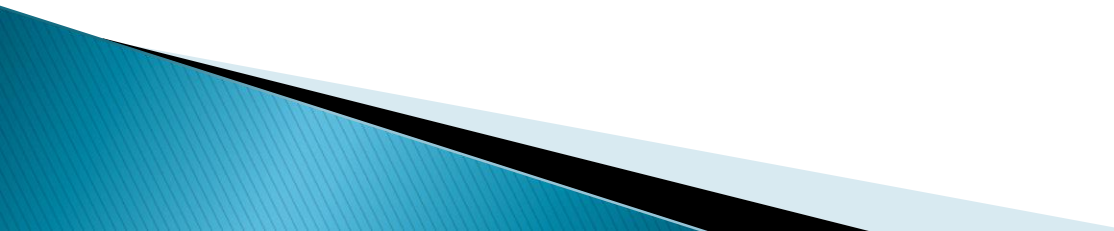
# A comma separated list of directories under which to store log files  
log.dirs=/tmp/kafka-logs-2





# Apache Kafka - Basic Operations

## Single Node-Multiple Brokers Configuration

- ▶ Start Multiple Brokers - After all the changes have been made on three servers then open three new terminals to start each broker one by one.
  - ▶ Broker1 : `$bin/kafka-server-start.sh config/server.properties`
  - ▶ Broker2 : `$bin/kafka-server-start.sh config/server-one.properties`
- 

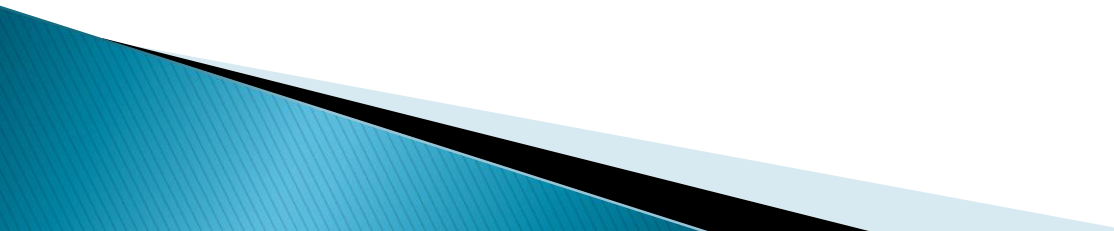
# Apache Kafka - Basic Operations

## Single Node-Multiple Brokers Configuration

- ▶ Broker3 : `$bin/kafka-server-start.sh config/server-two.properties`
- ▶ Now we have three different brokers running on the machine.
- ▶ Try it by yourself to check all the daemons by typing `jps` on the ZooKeeper terminal, then you would see the response.

# Kafka APIs

Kafka has five core APIs for Java and Scala:

- The [Admin API](#) to manage and inspect topics, brokers, and other Kafka objects.
  - The [Producer API](#) to publish (write) a stream of events to one or more Kafka topics.
  - The [Consumer API](#) to subscribe to (read) one or more topics and to process the stream of events produced to them.
  - The [Kafka Streams API](#) to implement stream processing applications and microservices.
  - The [Kafka Connect API](#) to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka.
- 

# Kafka Producer API

- ▶ Let us understand the most important set of Kafka producer API in this section.
- ▶ The central part of the KafkaProducer API is KafkaProducer class.
- ▶ The KafkaProducer class provides an option to connect a Kafka broker in its constructor with the following methods.
  - KafkaProducer class provides send method to send messages asynchronously to a topic.
  - The signature of send() is as follows
    - `producer.send(new ProducerRecord <byte[],byte[]> (topic, partition, key1, value1) , callback);`

# Kafka Producer API

- **ProducerRecord** - The producer manages a buffer of records waiting to be sent.
- **Callback** - A user-supplied callback to execute when the record has been acknowledged by the server (null indicates no callback).
- KafkaProducer class provides a flush method to ensure all previously sent messages have been actually completed.
- Syntax of the flush method is as follows
  - `public void flush()`
- KafkaProducer class provides partitionFor method, which helps in getting the partition metadata for a given topic.
- This can be used for custom partitioning.

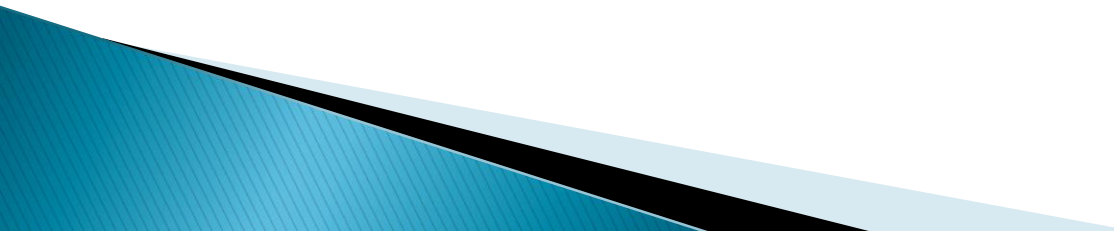
# Kafka Producer API

- The signature of this method is as follows
  - `public Map metrics()`
- It returns the map of internal metrics maintained by the producer.
- `public void close()` - KafkaProducer class provides close method blocks until all previously sent requests are completed.

# Kafka Producer API

- ▶ The producer class provides send method to send messages to either single or multiple topics using the following signatures.
  - `public void send(KeyedMessage <k,v> message)` - sends the data to a single topic, partitioned by key using either sync or async producer.
  - `public void send(List<KeyedMessage<k,v>>messages)` - sends data to multiple topics.
  - `Properties prop = new Properties();`
  - `prop.put(producer.type,"async")`
  - `ProducerConfig config = new ProducerConfig(prop);`

# Kafka Producer API

- ▶ There are two types of producers - Sync and Async.
  - ▶ The same API configuration applies to Sync producer as well.
  - ▶ The difference between them is a sync producer sends messages directly, but Async producer sends messages in background.
  - ▶ Async producer is preferred when you want a higher throughput.
  - ▶ In the previous releases like 0.8, an async producer does not have a callback for `send()` to register error handlers.
  - ▶ This is available only from release of 0.9.
- 



# Kafka Producer API

- ▶ `public void close()`
  - Producer class provides close method to close the producer pool connections to all Kafka brokers.

# Configuration Settings

- ▶ The Producer API's main configuration settings are listed below for better under-standing
  - **client.id**: identifies producer application
  - **producer.type** : either sync or async
  - **Acks** : The acks config controls the criteria under producer requests are considered complete.
  - **Retries** : If producer request fails, then automatically retry with specific value.
  - **bootstrap.servers**: bootstrapping list of brokers.
  - **linger.ms** : if you want to reduce the number of requests you can set linger.ms to something greater than some value.
  - **key.serializer** : Key for the serializer interface.
  - **value.serializer** : value for the serializer interface.

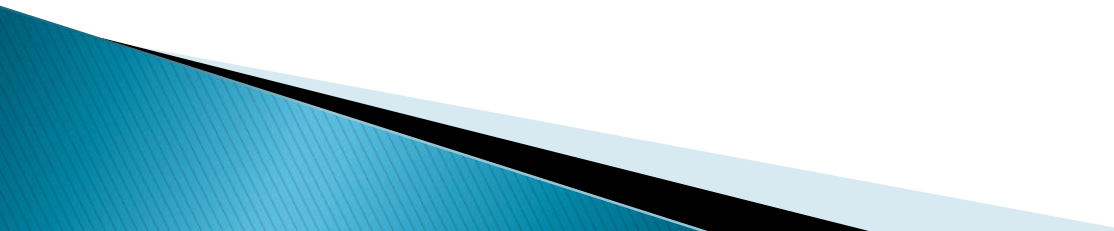
# Configuration Settings

- ▶ The Producer API's main configuration settings are listed below for better understanding
  - **batch.size** : Buffer size.
  - **buffer.memory** : controls the total amount of memory available to the producer for buffering.

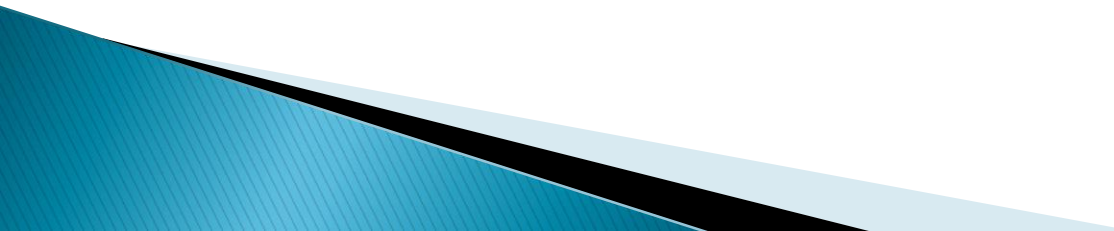
# ProducerRecord API

- ▶ ProducerRecord is a key/value pair that is sent to Kafka cluster.
- ▶ ProducerRecord class constructor for creating a record with partition, key and value pairs using the following signatures.
  - `public ProducerRecord (string topic, int partition, k key, v value)`
  - Topic - user defined topic name that will appended to record.
  - Partition - partition count
  - Key - The key that will be included in the record.
  - Value - Record contents

# ProducerRecord API

- `public ProducerRecord (string topic, k key, v value)`
  - This constructor is used to create a record with key, value pairs and without partition.
  - Topic - Create a topic to assign record.
  - Key - key for the record.
  - Value - record contents.
- 

# ProducerRecord API

- `public ProducerRecord (string topic, v value)`
  - `ProducerRecord` class creates a record without partition and key.
  - **Topic** - create a topic.
  - **Value** - record contents.
- 

# ProducerRecord API

- ▶ The ProducerRecord class methods are listed below
  - **public String topic():** Topic will append to the record.
  - **public K key():** Key that will be included in the record. If no such key, null will be returned here.
  - **public V value():** Record contents.
  - **partition():** Partition count for the record

# Simple Producer Example

## **SimpleProducer application**

- ▶ Before creating the application, first start ZooKeeper and Kafka broker then create your own topic in Kafka broker using create topic command.
- ▶ After that create a java class named SimpleProducer.java and type in the following coding.

```
//import util.properties packages  
import java.util.Properties;
```



# Simple Producer Example

```
//import simple producer packages  
import org.apache.kafka.clients.producer.Producer;
```

```
//import KafkaProducer packages  
import org.apache.kafka.clients.producer.KafkaProducer;
```

```
//import ProducerRecord packages  
import org.apache.kafka.clients.producer.ProducerRecord;
```

```
//Create java class named “SimpleProducer”  
public class SimpleProducer {
```



# Simple Producer Example

```
public static void main(String[] args) throws Exception{
```

```
    // Check arguments length value
```

```
    if(args.length == 0){
```

```
        System.out.println("Enter topic name");
```

```
        return;
```

```
    }
```

```
    //Assign topicName to string variable
```

```
        String topicName = args[0].toString();
```

# Simple Producer Example

```
// create instance for properties to access producer configs
```

```
Properties props = new Properties();
```

```
//Assign localhost id
```

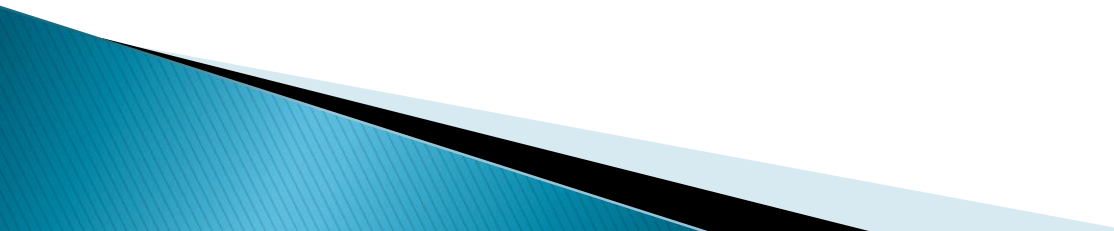
```
props.put("bootstrap.servers", "localhost:9092");
```

```
//Set acknowledgements for producer requests.
```

```
props.put("acks", "all");
```

```
//If the request fails, the producer can automatically retry,
```

```
props.put("retries", 0);
```



# Simple Producer Example

```
//Specify buffer size in config  
props.put("batch.size", 16384);
```

```
//Reduce the no of requests less than 0  
props.put("linger.ms", 1);
```

//The buffer.memory controls the total amount of memory available to the producer for buffering.

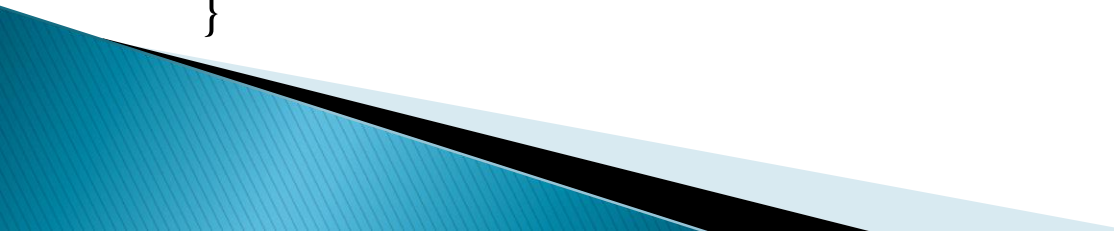
```
props.put("buffer.memory", 33554432);  
props.put("key.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");
```

# Simple Producer Example

```
props.put("value.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");
```

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

```
for(int i = 0; i < 10; i++)  
    producer.send(new ProducerRecord<String, String>(topicName,  
        Integer.toString(i), Integer.toString(i)));  
    System.out.println("Message sent successfully");  
    producer.close();  
}  
}
```



# Simple Producer Example

- ▶ Compilation - The application can be compiled using the following command.
  - `javac -cp "/path/to/kafka/kafka_2.12-3.1.0/lib/*" *.java`
- ▶ Execution - The application can be executed using the following command.
  - `java -cp "/path/to/kafka/kafka_2.12-3.1.0/lib/*" SimpleProducer <topic-name>`
  - Output  
Message sent successfully

# Simple Producer Example

- ▶ To check the above output open new terminal and type Consumer CLI command to receive messages.
  - `bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic <topic-name> --from-beginning`

1

2

3

4

5

6

7

8

9

10

# KafkaConsumer API

- ▶ KafkaConsumer API is used to consume messages from the Kafka cluster.
- ▶ KafkaConsumer class constructor is defined below.
  - `Public KafkaConsumer(java.util.Map<java.lang.String,java.lang.Object> configs)`
  - configs - Return a map of consumer configs.
- ▶ KafkaConsumer class has the following significant methods that are listed below.
  - **`public java.util.Set<TopicPartition> assignment()`** : Get the set of partitions currently assigned by the consumer.
  - **`public string subscription()`** : Subscribe to the given list of topics to get dynamically assigned partitions.



# KafkaConsumer API

- ▶ KafkaConsumer class has the following significant methods that are listed below.
  - **public void subscribe(java.util.List <java.lang.String> topics, ConsumerRebalanceListener listener):** Subscribe to the given list of topics to get dynamically assigned partitions.
  - **public void unsubscribe():** Unsubscribe the topics from the given list of partitions.
  - **public void subscribe(java.util.List <java.lang.String> topics) :** Subscribe to the given list of topics to get dynamically assigned partitions.
  - If the given list of topics is empty, it is treated the same as unsubscribe().

# KafkaConsumer API

- ▶ KafkaConsumer class has the following significant methods that are listed below.
  - **public void subscribe(java.util.regex.Pattern pattern, ConsumerRebalanceListener listener) :** The argument pattern refers to the subscribing pattern in the format of regular expression and the listener argument gets notifications from the subscribing pattern.
  - **public void assign(java.util.List <TopicPartition> partitions) :** Manually assign a list of partitions to the customer.
  - **poll() :**Fetch data for the topics or partitions specified using one of the subscribe/assign APIs. This will return error, if the topics are not subscribed before the polling for data.
  - **public void commitSync() :** Commit offsets returned on the last poll() for all the subscribed list of topics and partitions. The same operation is applied to commitAsyn().

# KafkaConsumer API

- ▶ KafkaConsumer class has the following significant methods that are listed below.
  - **public void seek(TopicPartition partition, long offset) :** Fetch the current offset value that consumer will use on the next poll() method.
  - **public void resume() :** Resume the paused partitions.
  - **public void wakeup() :** Wakeup the consumer.

# ConsumerRecord API

- ▶ The ConsumerRecord API is used to receive records from the Kafka cluster.
- ▶ This API consists of a topic name, partition number, from which the record is being received and an offset that points to the record in a Kafka partition.
- ▶ *ConsumerRecord* class is used to create a consumer record with specific topic name, partition count and <key, value> pairs. It has the following signature.
  - `public ConsumerRecord(string topic,int partition, long offset, K key, V value)`

# ConsumerRecord API

- Topic - The topic name for consumer record received from the Kafka cluster.
  - Partition - Partition for the topic.
  - Key - The key of the record, if no key exists null will be returned.
  - Value - Record contents.
- 
- ▶ ConsumerRecords API acts as a container for ConsumerRecord.
  - ▶ This API is used to keep the list of ConsumerRecord per partition for a particular topic.
  - ▶ Its Constructor is defined below.
    - `public ConsumerRecords(java.util.Map <TopicPartition,java.util.List <Consumer-Record> K, V>>> records)`

# ConsumerRecord API

- TopicPartition - Return a map of partition for a particular topic.
- Records - Return list of ConsumerRecord.
- ▶ ConsumerRecords class has the following methods defined.
  - **public int count():** The number of records for all the topics.
  - **public Set partitions():** The set of partitions with data in this record set (if no data was returned then the set is empty).
  - **public Iterator iterator() :** Iterator enables you to cycle through a collection, obtaining or re-moving elements.
  - **public List records() :** Get list of records for the given partition.

# ConsumerRecord API

## Configuration Settings

- ▶ The configuration settings for the Consumer client API main configuration settings are listed below
  - **bootstrap.servers**: Bootstrapping list of brokers.
  - **group.id**: Assigns an individual consumer to a group.
  - **enable.auto.commit** : Enable auto commit for offsets if the value is true, otherwise not committed.
  - **auto.commit.interval.ms** : Return how often updated consumed offsets are written to ZooKeeper.
  - **session.timeout.ms** : Indicates how many milliseconds Kafka will wait for the ZooKeeper to respond to a request (read or write) before giving up and continuing to consume messages.

# Simple Consumer Example

- ▶ First, start your ZooKeeper and Kafka broker. Then create a SimpleConsumer application with the java class named SimpleConsumer.java and type the following code.

```
import java.util.Properties;  
import java.util.Arrays;  
import org.apache.kafka.clients.consumer.KafkaConsumer;  
import org.apache.kafka.clients.consumer.ConsumerRecords;  
import org.apache.kafka.clients.consumer.ConsumerRecord;
```



# Simple Consumer Example

```
public class SimpleConsumer {  
    public static void main(String[] args) throws Exception {  
        if(args.length == 0){  
            System.out.println("Enter topic name");  
            return;  
        }  
  
        //Kafka consumer configuration settings  
        String topicName = args[0].toString();  
        Properties props = new Properties();
```

# Simple Consumer Example

```
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test");
props.put("enable.auto.commit", "true");
props.put("auto.commit.interval.ms", "1000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer
    <String, String>(props);
//Kafka Consumer subscribes list of topics here.
consumer.subscribe(Arrays.asList(topicName))
```

# Simple Consumer Example

```
//print the topic name
```

```
System.out.println("Subscribed to topic " + topicName);
```

```
int i = 0;
```

```
while (true) {
```

```
    ConsumerRecords<String, String> records = consumer.poll(100);
```

```
    for (ConsumerRecord<String, String> record : records)
```

```
        // print the offset,key and value for the consumer records.
```

```
        System.out.printf("offset = %d, key = %s, value = %s\n",
```

```
            record.offset(), record.key(), record.value());
```

```
    }
```

```
}
```

```
}
```



# Simple Consumer Example

- ▶ **Compilation** - The application can be compiled using the following command.
  - `javac -cp "/path/to/kafka/kafka_2.12-3.1.0/lib/*" *.java`
- ▶ **Execution** - The application can be executed using the following command
  - `java -cp "/path/to/kafka/kafka_2.12-3.1.0/lib/*" SimpleConsumer <topic-name>`
- ▶ **Input** - Open the producer CLI and send some messages to the topic. You can put the simple input as 'Hello Consumer'.
- ▶ **Output** - Following will be the output.
  - Subscribed to topic Hello-Kafka
  - `offset = 3, key = null, value = Hello Consumer`

# Exercise 1

- ▶ Create kafka Brokers
  - ▶ Create topic
  - ▶ Create Producer Class
  - ▶ Create Consumer Class
- 