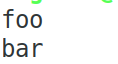
# **[Use Kafka Connect to import/export data](https://kafka.apache.org/documentation.html" \l "quickstart_kafkaconnect)**

First, we'll start by creating some seed data to test with:

$ **echo -e "foo\nbar" > test.txt**

**vagrant@master:~$ echo -e "foo\nbar" > test.txt**

**vagrant@master:~$ cat test.txt**



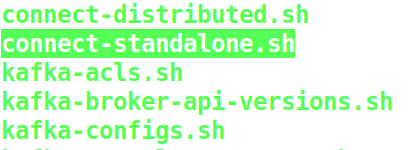
**vagrant@master:~$ cp test.txt bigdata/kafka/**

Next, we'll start **two connectors** running in *standalone* mode, which means they run in a single, local, dedicated process.

**vagrant@master:~$ cd bigdata/kafka/**

**vagrant@master:~/bigdata/kafka$ cd bin**

**vagrant@master:~/bigdata/kafka/bin$ ls**

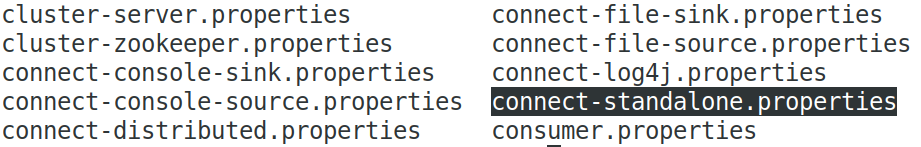


We provide **three configuration files** as parameters.

* The **first** is always the configuration for the Kafka Connect process, containing common configuration such as the Kafka brokers to connect to and the serialization format for data.
* The **remaining configuration files** each specify a connector to create.
  + These files include a unique connector name, the connector class to instantiate, and any other configuration required by the connector.

**vagrant@master:~$ cd /home/vagrant/bigdata/kafka/config/**

**vagrant@master:~/bigdata/kafka/config$ ls**



**vagrant@master:~/bigdata/kafka/config$ gedit connect-standalone.properties**

vagrant@master:~/bigdata/kafka/config$ gedit connect-file-source.properties

**name=local-file-source**

**connector.class=FileStreamSource**

**tasks.max=1**

**file=test.txt**

**topic=connect-test**

vagrant@master:~/bigdata/kafka/config$ gedit connect-file-sink.properties

**name=local-file-sink**

**connector.class=FileStreamSink**

**tasks.max=1**

**file=test.sink.txt**

**topics=connect-test**

**You need to start Kafka server and Zookeeper before running Kafka Connect.**

**vagrant@master:~/bigdata/kafka/config$ cd ..**

**vagrant@master:~/bigdata/kafka$**

**$ bin/zookeeper-server-start.sh config/zookeeper.properties &**

**vagrant@master:~/bigdata/kafka$ jps**

**2169 Jps**

**1899 QuorumPeerMain**

**$ bin/kafka-server-start.sh config/server.properties &**

**vagrant@master:~/bigdata/kafka$ jps**

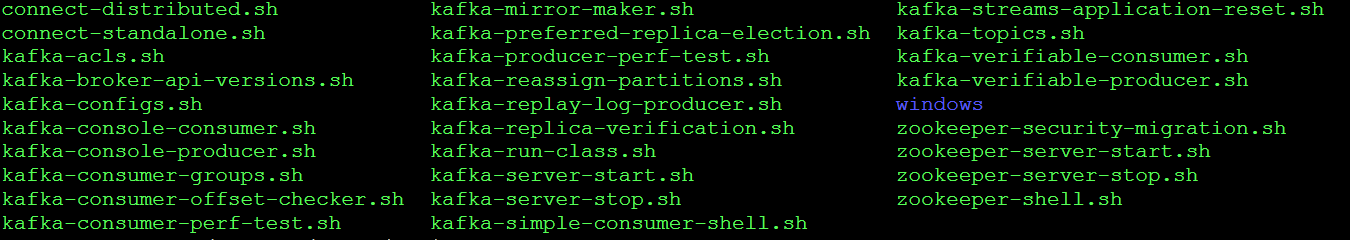
**2179 Kafka**

**2490 Jps**

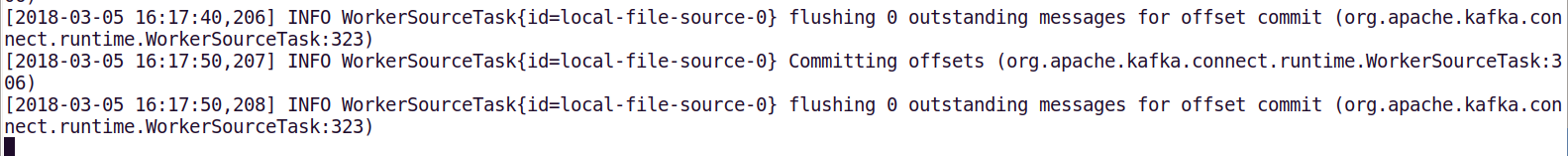
**1899 QuorumPeerMain**

**vagrant@master:~/bigdata/kafka$ kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partition 1 --topic connect-test**

**vagrant@master:~/bigdata/kafka$ ls bin**



$ bin/connect-standalone.sh config/connect-standalone.properties config/connect-file-source.properties config/connect-file-sink.properties



These sample configuration files, included with Kafka,

* use the default local cluster configuration you started earlier and
* **create two connectors**:
  + the first is a source connector that reads lines from an input file and produces each to a Kafka topic and
  + the second is a sink connector that reads messages from a Kafka topic and produces each as a line in an output file.

During startup you'll see a number of log messages, including some indicating that the connectors are being instantiated.

Once the Kafka Connect process has started,

* the source connector should start reading lines from test.txt and producing them to the topic connect-test, and
* the sink connector should start reading messages from the topic connect-test and write them to the file test.sink.txt.

We can verify the data has been delivered through the entire pipeline by examining the contents of the output file:

**Open Another Terminal**

**vagrant@master:~/bigdata/kafka$ ls**

$ **cat test.sink.txt**

foo

bar

**Open Another Terminal**

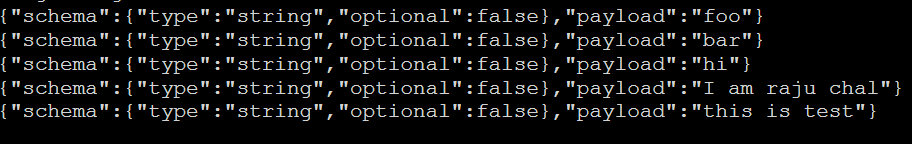
Note that the data is being stored in the Kafka topic **connect-test,** so we can also run a console consumer to see the data in the topic (or use custom consumer code to process it):

$ **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic connect-test --from-beginning**

{"schema":{"type":"string","optional":false},"payload":"foo"}

{"schema":{"type":"string","optional":false},"payload":"bar"}

...



**[**<https://www.confluent.io/blog/kafka-connect-deep-dive-converters-serialization-explained/>]

**Open Another Terminal**

The connectors continue to process data, so we can add data to the file and see it move through the pipeline:

Add some more data in the file “**test.txt**”.

> **echo "Another line" >> test.txt**

**vagrant@master:~/bigdata/kafka$ cat test.sink.txt**

foo

bar

**vagrant@master:~/bigdata/kafka$ cat >>test.txt**

hi

I am Karthik

this is test

^C

**vagrant@master:~/bigdata/kafka$ cat test.sink.txt**

foo

bar

hi

I am Karthik

this is test

**You should see the line appear in the console consumer output and in the sink file.**

# [**Custom**](https://kafka.apache.org/documentation.html#quickstart_kafkastreams) **Producer & Consumer**

**vagrant@master:~/bigdata/kafka$ bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic custom-topic1**

Created topic "custom-topic1".

**vagrant@master:~/bigdata/kafka$ bin/kafka-topics.sh --list --zookeeper localhost:2181**

**vagrant@master:~/bigdata/Dataset/Kafka-Progs$ gedit SimpleProducer.java**

//import util.properties packages

import java.util.Properties;

//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 {

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();

// 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);

//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");

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

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

for(int i = 0; i < 100; i++)

producer.send(new ProducerRecord<String, String>(topicName,Integer.toString(i), Integer.toString(i)));

System.out.println("Message sent successfully");

producer.close();

}

}

**Compilation** − The application can be compiled using the following command.

**vagrant@master:~/** **/bigdata/Dataset/Kafka-Progs$ javac -cp "/home/vagrant/bigdata/kafka/libs/\*" SimpleProducer.java**

**Execution** − The application can be executed using the following command.

**vagrant@master:~/bigdata/Dataset/Kafka-Progs$ java -cp "/home/vagrant/bigdata/kafka/libs/\*:." SimpleProducer custom-topic1**

**Run the Console Consumer**

**vagrant@master:~/bigdata/kafka$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic custom-topic1 --from-beginning**

# **Custom Consumer Application**

**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)

**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**

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)**

**TopicPartition** − Return a map of partition for a particular topic.

**Records** − Return list of ConsumerRecord.

**vagrant@master:~/bigdata/Dataset/Kafka-Progs$ gedit SimpleConsumer.java**

**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;**

**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();**

**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));**

**//print the topic name**

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

**int i = 0;**

**while (true) {**

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

//Kafka Consumer Poll method. The poll method returns fetched records based on current partition offset. The //poll method is a blocking method waiting for **specified** time in seconds.

**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());**

**}**

**}**

**}**

**Compile the program**

**vagrant@master:~/bigdata/Dataset/Kafka-Progs$ javac -cp "/home/vagrant/bigdata/kafka/libs/\*" SimpleConsumer.java**

**Run the Consumer Program**

**vagrant@master:~/bigdata/Dataset/Kafka-Progs$ java -cp "/home/vagrant/bigdata/kafka/libs/\*:." SimpleConsumer custom-topic1**

