**Quickstart**

This tutorial assumes you are starting fresh and have no existing Kafka or ZooKeeper data. Since Kafka console scripts are different for Unix-based and Windows platforms, on Windows platforms use bin\windows\ instead of bin/, and change the script extension to .bat.

[**Step 1: Download the code**](http://kafka.apache.org/quickstart#quickstart_download)

[Download](https://www.apache.org/dyn/closer.cgi?path=/kafka/0.10.2.0/kafka_2.11-0.10.2.0.tgz) the 0.10.2.0 release and un-tar it.

> **tar -xzf kafka\_2.11-0.10.2.0.tgz**

> **cd kafka\_2.11-0.10.2.0**

[**Step 2: Start the server**](http://kafka.apache.org/quickstart#quickstart_startserver)

Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

> **bin/zookeeper-server-start.sh config/zookeeper.properties**

[2013-04-22 15:01:37,495] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)

...

Now start the Kafka server:

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

[2013-04-22 15:01:47,028] INFO Verifying properties (kafka.utils.VerifiableProperties)

[2013-04-22 15:01:47,051] INFO Property socket.send.buffer.bytes is overridden to 1048576 (kafka.utils.VerifiableProperties)

...

[**Step 3: Create a topic**](http://kafka.apache.org/quickstart#quickstart_createtopic)

Let's create a topic named "test" with a single partition and only one replica:

> **bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test**

We can now see that topic if we run the list topic command:

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

test

Alternatively, instead of manually creating topics you can also configure your brokers to auto-create topics when a non-existent topic is published to.

[**Step 4: Send some messages**](http://kafka.apache.org/quickstart#quickstart_send)

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default, each line will be sent as a separate message.

Run the producer and then type a few messages into the console to send to the server.

> **bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test**

**This is a message**

**This is another message**

[**Step 5: Start a consumer**](http://kafka.apache.org/quickstart#quickstart_consume)

Kafka also has a command line consumer that will dump out messages to standard output.

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

This is a message

This is another message

If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

All of the command line tools have additional options; running the command with no arguments will display usage information documenting them in more detail.

[**Step 6: Setting up a multi-broker cluster**](http://kafka.apache.org/quickstart#quickstart_multibroker)

So far we have been running against a single broker, but that's no fun. For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our cluster to three nodes (still all on our local machine).

First we make a config file for each of the brokers (on Windows use the copy command instead):

> **cp config/server.properties config/server-1.properties**

> **cp config/server.properties config/server-2.properties**

Now edit these new files and set the following properties:

config/server-1.properties:

broker.id=1

listeners=PLAINTEXT://:9093

log.dir=/tmp/kafka-logs-1

config/server-2.properties:

broker.id=2

listeners=PLAINTEXT://:9094

log.dir=/tmp/kafka-logs-2

The broker.id property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each other's data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:

> **bin/kafka-server-start.sh config/server-1.properties &**

...

> **bin/kafka-server-start.sh config/server-2.properties &**

...

Now create a new topic with a replication factor of three:

> **bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic**

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic**

Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:

Topic: my-replicated-topic Partition: 0 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

* "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
* "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
* "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in my example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test**

Topic:test PartitionCount:1 ReplicationFactor:1 Configs:

Topic: test Partition: 0 Leader: 0 Replicas: 0 Isr: 0

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:

> **bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic**

...

**my test message 1**

**my test message 2**

**^C**

Now let's consume these messages:

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic**

...

my test message 1

my test message 2

**^C**

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

> **ps aux | grep server-1.properties**

*7564* ttys002 0:15.91 /System/Library/Frameworks/JavaVM.framework/Versions/1.8/Home/bin/java...

> **kill -9 7564**

On Windows use:

> **wmic process get processid,caption,commandline | find "java.exe" | find "server-1.properties"**

java.exe java -Xmx1G -Xms1G -server -XX:+UseG1GC ... build\libs\kafka\_2.10-0.10.2.0.jar" kafka.Kafka config\server-1.properties *644*

> **taskkill /pid 644 /f**

Leadership has switched to one of the slaves and node 1 is no longer in the in-sync replica set:

> **bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic my-replicated-topic**

Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:

Topic: my-replicated-topic Partition: 0 Leader: 2 Replicas: 1,2,0 Isr: 2,0

But the messages are still available for consumption even though the leader that took the writes originally is down:

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic**

...

my test message 1

my test message 2

**^C**

[**Step 7: Use Kafka Connect to import/export data**](http://kafka.apache.org/quickstart#quickstart_kafkaconnect)

Writing data from the console and writing it back to the console is a convenient place to start, but you'll probably want to use data from other sources or export data from Kafka to other systems. For many systems, instead of writing custom integration code you can use Kafka Connect to import or export data.

Kafka Connect is a tool included with Kafka that imports and exports data to Kafka. It is an extensible tool that runs *connectors*, which implement the custom logic for interacting with an external system. In this quickstart we'll see how to run Kafka Connect with simple connectors that import data from a file to a Kafka topic and export data from a Kafka topic to a file.

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

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

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

> **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-testand 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:

> **cat test.sink.txt**

foo

bar

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"}

...

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

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

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

[**Step 8: Use Kafka Streams to process data**](http://kafka.apache.org/quickstart#quickstart_kafkastreams)

Kafka Streams is a client library of Kafka for real-time stream processing and analyzing data stored in Kafka brokers. This quickstart example will demonstrate how to run a streaming application coded in this library. Here is the gist of the [WordCountDemo](https://github.com/apache/kafka/blob/0.10.2/streams/examples/src/main/java/org/apache/kafka/streams/examples/wordcount/WordCountDemo.java) example code (converted to use Java 8 lambda expressions for easy reading).

// Serializers/deserializers (serde) for String and Long types

final Serde<String> stringSerde = Serdes.String();

final Serde<Long> longSerde = Serdes.Long();

// Construct a `KStream` from the input topic ""streams-file-input", where message values

// represent lines of text (for the sake of this example, we ignore whatever may be stored

// in the message keys).

KStream<String, String> textLines = builder.stream(stringSerde, stringSerde, "streams-file-input");

KTable<String, Long> wordCounts = textLines

// Split each text line, by whitespace, into words.

.flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

// Group the text words as message keys

.groupBy((key, value) -> value)

// Count the occurrences of each word (message key).

.count("Counts")

// Store the running counts as a changelog stream to the output topic.

wordCounts.to(stringSerde, longSerde, "streams-wordcount-output");

It implements the WordCount algorithm, which computes a word occurrence histogram from the input text. However, unlike other WordCount examples you might have seen before that operate on bounded data, the WordCount demo application behaves slightly differently because it is designed to operate on an **infinite, unbounded stream** of data. Similar to the bounded variant, it is a stateful algorithm that tracks and updates the counts of words. However, since it must assume potentially unbounded input data, it will periodically output its current state and results while continuing to process more data because it cannot know when it has processed "all" the input data.

As the first step, we will prepare input data to a Kafka topic, which will subsequently be processed by a Kafka Streams application.

> **echo -e "all streams lead to kafka\nhello kafka streams\njoin kafka summit" > file-input.txt**

Or on Windows:

> **echo all streams lead to kafka> file-input.txt**

> **echo hello kafka streams>> file-input.txt**

> **echo|set /p=join kafka summit>> file-input.txt**

Next, we send this input data to the input topic named **streams-file-input** using the console producer, which reads the data from STDIN line-by-line, and publishes each line as a separate Kafka message with null key and value encoded a string to the topic (in practice, stream data will likely be flowing continuously into Kafka where the application will be up and running):

> **bin/kafka-topics.sh --create \**

**--zookeeper localhost:2181 \**

**--replication-factor 1 \**

**--partitions 1 \**

**--topic streams-file-input**

> **bin/kafka-console-producer.sh --broker-list localhost:9092 --topic streams-file-input < file-input.txt**

We can now run the WordCount demo application to process the input data:

> **bin/kafka-run-class.sh org.apache.kafka.streams.examples.wordcount.WordCountDemo**

The demo application will read from the input topic **streams-file-input**, perform the computations of the WordCount algorithm on each of the read messages, and continuously write its current results to the output topic **streams-wordcount-output**. Hence there won't be any STDOUT output except log entries as the results are written back into in Kafka. The demo will run for a few seconds and then, unlike typical stream processing applications, terminate automatically.

We can now inspect the output of the WordCount demo application by reading from its output topic:

> **bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \**

**--topic streams-wordcount-output \**

**--from-beginning \**

**--formatter kafka.tools.DefaultMessageFormatter \**

**--property print.key=true \**

**--property print.value=true \**

**--property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \**

**--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer**

with the following output data being printed to the console:

all 1

lead 1

to 1

hello 1

streams 2

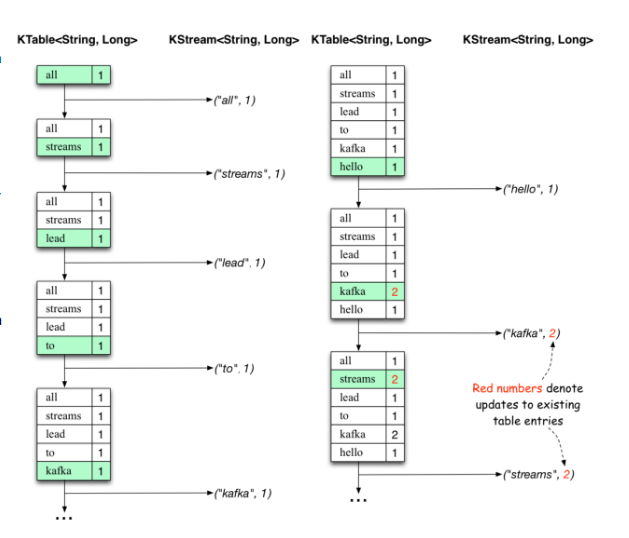
join 1

kafka 3

summit 1

Here, the first column is the Kafka message key in java.lang.String format, and the second column is the message value in java.lang.Long format. Note that the output is actually a continuous stream of updates, where each data record (i.e. each line in the original output above) is an updated count of a single word, aka record key such as "kafka". For multiple records with the same key, each later record is an update of the previous one.

The two diagrams below illustrate what is essentially happening behind the scenes. The first column shows the evolution of the current state of the KTable<String, Long> that is counting word occurrences for count. The second column shows the change records that result from state updates to the KTable and that are being sent to the output Kafka topic **streams-wordcount-output**.



First the text line “all streams lead to kafka” is being processed. The KTable is being built up as each new word results in a new table entry (highlighted with a green background), and a corresponding change record is sent to the downstream KStream.

When the second text line “hello kafka streams” is processed, we observe, for the first time, that existing entries in the KTable are being updated (here: for the words “kafka” and for “streams”). And again, change records are being sent to the output topic.

And so on (we skip the illustration of how the third line is being processed). This explains why the output topic has the contents we showed above, because it contains the full record of changes.

Looking beyond the scope of this concrete example, what Kafka Streams is doing here is to leverage the duality between a table and a changelog stream (here: table = the KTable, changelog stream = the downstream KStream): you can publish every change of the table to a stream, and if you consume the entire changelog stream from beginning to end, you can reconstruct the contents of the table.

Now you can write more input messages to the **streams-file-input** topic and observe additional messages added to **streams-wordcount-output** topic, reflecting updated word counts (e.g., using the console producer and the console consumer, as described above).

You can stop the console consumer via **Ctrl-C**.





