APACHE KAFKA QUICKSTART

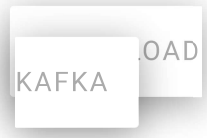
Everything you need to know about Kafka in 10 minutes
(clicking the image will load a video from YouTube)

STEP 1: GET KAFKA

Download the latest Kafka release and extract it:

```
$ tar -xzf kafka_2.13-3.2.1.tgz
$ cd kafka_2.13-3.2.1
```

STEP 2: START THE KAFKA ENVIRONMENT




```
# Start the ZooKeeper service
# Note: Soon, ZooKeeper will no longer be required by A
$ bin/zookeeper-server-start.sh config/zookeeper.proper
```

Open another terminal session and run:

```
# Start the Kafka broker service
$ bin/kafka-server-start.sh config/server.properties
```

Once all services have successfully launched, you will have a basic Kafka environment running and ready to use.

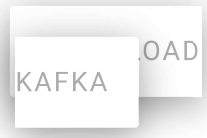
STEP 3: CREATE A TOPIC TO STORE YOUR EVENTS

Kafka is a distributed *event streaming platform* that lets you read, write, store, and process *events* (also called *records* or *messages* in the documentation) across many machines.

Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in *topics*. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder.

So before you can write your first events, you must create a topic. Open another terminal session and run:

```
$ bin/kafka-topics.sh --create --topic quickstart-event
```




```
$ bin/kafka-topics.sh --describe --topic quickstart-eve
Topic:quickstart-events  PartitionCount:1    Replicatio
    Topic: quickstart-events Partition: 0    Leader: 0
```

STEP 4: WRITE SOME EVENTS INTO THE TOPIC

A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever.

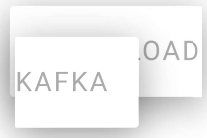
Run the console producer client to write a few events into your topic. By default, each line you enter will result in a separate event being written to the topic.

```
bin/kafka-console-producer.sh --topic quickstart-events --boot
This is my first event
This is my second event
```

You can stop the producer client with `Ctrl-C` at any time.

STEP 5: READ THE EVENTS

Open another terminal session and run the console consumer client to read the events you just created:



You can stop the consumer client with `Ctrl-C` at any time.

Feel free to experiment: for example, switch back to your producer terminal (previous step) to write additional events, and see how the events immediately show up in your consumer terminal.

Because events are durably stored in Kafka, they can be read as many times and by as many consumers as you want. You can easily verify this by opening yet another terminal session and re-running the previous command again.

STEP 6: IMPORT/EXPORT YOUR DATA AS STREAMS OF EVENTS WITH KAFKA CONNECT

You probably have lots of data in existing systems like relational databases or traditional messaging systems, along with many applications that already use these systems. Kafka Connect allows you to continuously ingest data from external systems into Kafka, and vice versa. It is an extensible tool that runs *connectors*, which implement the custom logic for interacting with an external system. It is thus very easy to integrate existing systems with Kafka. To make this process even easier, there are hundreds of such connectors readily available.

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, make sure to add `connect-file-3.2.1.jar` to the `plugin.path` property in the Connect worker's configuration. For the purpose of this quickstart we'll use a relative path and consider the connectors' package as an uber jar, which works when the quickstart commands are run from the installation directory. However, it's worth noting that for production

`plugin.path` configuration property match the following, and save the file:

```
> echo "plugin.path=libs/connect-file-3.2.1.jar"
```

Then, start by creating some seed data to test with:

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

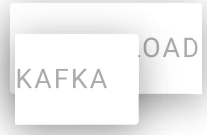
Or on Windows:

```
> echo foo> test.txt
> echo bar>> 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.proper
```

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.



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:

```
> more 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
{"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 7: PROCESS YOUR EVENTS WITH KAFKA STREAMS

simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, and distributed. The library supports exactly-once processing, stateful operations and aggregations, windowing, joins, processing based on event-time, and much more.

To give you a first taste, here's how one would implement the popular `WordCount` algorithm:

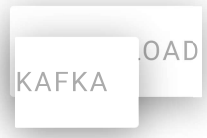
```
KStream<String, String> textLines = builder.stream("qui

KTable<String, Long> wordCounts = textLines
            .flatMapValues(line -> Arrays.asList(line.t
            .groupBy((keyIgnored, word) -> word)
            .count();

wordCounts.toStream().to("output-topic", Produced.with(
```

The Kafka Streams demo and the app development tutorial demonstrate how to code and run such a streaming application from start to finish.

STEP 8: TERMINATE THE KAFKA ENVIRONMENT

Now that you reached the end of the quickstart, feel free to tear down the Kafka environment—or continue playing around.

1. Stop the producer and consumer clients with `Ctrl-C`, if you haven't done so already.
2. Stop the Kafka broker with `Ctrl-C`.
3. Lastly, stop the ZooKeeper server with `Ctrl-C`.



CONGRATULATIONS!

You have successfully finished the Apache Kafka quickstart.

To learn more, we suggest the following next steps:

- Read through the brief Introduction to learn how Kafka works at a high level, its main concepts, and how it compares to other technologies. To understand Kafka in more detail, head over to the Documentation.
- Browse through the Use Cases to learn how other users in our world-wide community are getting value out of Kafka.
- Join a local Kafka meetup group and watch talks from Kafka Summit, the main conference of the Kafka community.