# Guide to Setting Up Apache Kafka Using Docker

****Docker is one of the most popular container engines**** used in the software industry to ****create, package and deploy applications.****

In this tutorial, we'll learn how to do an [Apache Kafka](https://www.baeldung.com/spring-kafka" \l "overview) setup using Docker.

## **Single Node Setup**

A single node Kafka broker setup would meet most of the ****local development needs****, so let's start by learning this simple setup.

### **docker-compose.yml Configuration**

To start an Apache Kafka server, we'd first need to start a [Zookeeper](https://www.baeldung.com/java-zookeeper" \l "overview) server.

We can ****configure this dependency in a**[docker-compose.yml](https://www.baeldung.com/docker-compose)**file****, which will ensure that the Zookeeper server always starts before the Kafka server and stops after it.

Let's create a simple docker-compose.yml file with two services, namely zookeeper and kafka:

version: '2'services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

ZOOKEEPER\_TICK\_TIME: 2000

ports:

- 22181:2181

kafka:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

ports:

- 29092:29092

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT\_HOST://localhost:29092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

In this setup, our Zookeeper server is listening on port=2181 for the kafka service, which is defined within the same container setup. However, for any client running on the host, it'll be exposed on port 22181.

Similarly, the **kafka**service is exposed to the host applications through port**29092**, but it is actually advertised on port 9092 within the container environment ****configured by the**KAFKA\_ADVERTISED\_LISTENERS**property.

Tóm tắt: Đây là cách để dựng 1 kafka broker trên máy cá nhân sử dụng docker.

# Guide to Setting Up Apache Kafka on linux

Tóm tắt: Đây là cách để dựng 1 kafka broker trên máy cá nhân theo hướng dẫn chính thống trên trang của kafka.

## **[STEP 1: GET KAFKA](https://kafka.apache.org/quickstart" \l "quickstart_download)**

[Download](https://www.apache.org/dyn/closer.cgi?path=/kafka/3.2.0/kafka_2.13-3.2.0.tgz) the latest Kafka release and extract it:

$ tar -xzf kafka\_2.13-3.2.0.tgz

$ cd kafka\_2.13-3.2.0

## **[STEP 2: START THE KAFKA ENVIRONMENT](https://kafka.apache.org/quickstart" \l "quickstart_startserver)**

*NOTE: Your local environment must have Java 8+ installed.*

Run the following commands in order to start all services in the correct order:

# Start the ZooKeeper service# Note: Soon, ZooKeeper will no longer be required by Apache Kafka.

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

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](https://kafka.apache.org/quickstart" \l "quickstart_createtopic)**

Kafka is a distributed event streaming platform that lets you read, write, store, and process [events](https://kafka.apache.org/documentation/" \l "messages) (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](https://kafka.apache.org/documentation/" \l "intro_concepts_and_terms). 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-events --bootstrap-server localhost:9092

All of Kafka's command line tools have additional options: run the kafka-topics.sh command without any arguments to display usage information. For example, it can also show you [details such as the partition count](https://kafka.apache.org/documentation/" \l "intro_concepts_and_terms) of the new topic:

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

Topic:quickstart-events PartitionCount:1 ReplicationFactor:1 Configs:

Topic: quickstart-events Partition: 0 Leader: 0 Replicas: 0 Isr: 0

## **[STEP 4: WRITE SOME EVENTS INTO THE TOPIC](https://kafka.apache.org/quickstart" \l "quickstart_send)**

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 --bootstrap-server localhost:9092

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](https://kafka.apache.org/quickstart" \l "quickstart_consume)**

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

$ bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server localhost:9092

This is my first event

This is my second event

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](https://kafka.apache.org/quickstart" \l "quickstart_kafkaconnect)**

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](https://kafka.apache.org/documentation/" \l "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-3.2.0.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 deployments using absolute paths is always preferable. See [plugin.path](https://kafka.apache.org/quickstart" \l "connectconfigs_plugin.path) for a detailed description of how to set this config.

Edit the config/connect-standalone.properties file, add or change the plugin.path configuration property match the following, and save the file:

> echo "plugin.path=lib/connect-file-3.2.0.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.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:

> 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: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 7: PROCESS YOUR EVENTS WITH KAFKA STREAMS](https://kafka.apache.org/quickstart" \l "quickstart_kafkastreams)**

Once your data is stored in Kafka as events, you can process the data with the [Kafka Streams](https://kafka.apache.org/documentation/streams) client library for Java/Scala. It allows you to implement mission-critical real-time applications and microservices, where the input and/or output data is stored in Kafka topics. Kafka Streams combines the 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("quickstart-events");

KTable<String, Long> wordCounts = textLines

.flatMapValues(line -> Arrays.asList(line.toLowerCase().split(" ")))

.groupBy((keyIgnored, word) -> word)

.count();

wordCounts.toStream().to("output-topic", Produced.with(Serdes.String(), Serdes.Long()));

The [Kafka Streams demo](https://kafka.apache.org/documentation/streams/quickstart) and the [app development tutorial](https://kafka.apache.org/documentation/streams/tutorial) demonstrate how to code and run such a streaming application from start to finish.

## **[STEP 8: TERMINATE THE KAFKA ENVIRONMENT](https://kafka.apache.org/quickstart" \l "quickstart_kafkaterminate)**

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.

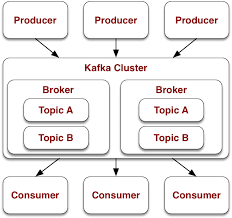
If you also want to delete any data of your local Kafka environment including any events you have created along the way, run the command:

$ rm -rf /tmp/kafka-logs /tmp/zookeeper

# Intro to Apache Kafka with Spring

## Basic concept

Nói 1 cách dễ hiểu thì đây là 1 cái queue để xử lý bất đồng bộ giữa các hệ thống, phân biệt với nhau bằng topic



## Producer

### **Producer Configuration**

@Configurationpublic class KafkaProducerConfig {

@Bean

public ProducerFactory<String, String> producerFactory() {

Map<String, Object> configProps = new HashMap<>();

configProps.put(

ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG,

bootstrapAddress);

configProps.put(

ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG,

StringSerializer.class);

configProps.put(

ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,

StringSerializer.class);

return new DefaultKafkaProducerFactory<>(configProps);

}

@Bean

public KafkaTemplate<String, String> kafkaTemplate() {

return new KafkaTemplate<>(producerFactory());

}

}

### **Publishing Messages**

We can send messages using the KafkaTemplate class:

@Autowiredprivate KafkaTemplate<String, String> kafkaTemplate;

public void sendMessage(String msg) {

kafkaTemplate.send(topicName, msg);

}

****The**send**API returns a**ListenableFuture**object.**** If we want to block the sending thread and get the result about the sent message, we can call the get API of the ListenableFuture object. The thread will wait for the result, but it will slow down the producer.

Kafka is a fast stream processing platform. Therefore, it's better to handle the results asynchronously so that the subsequent messages do not wait for the result of the previous message.

We can do this through a callback:

public void sendMessage(String message) {

ListenableFuture<SendResult<String, String>> future =

kafkaTemplate.send(topicName, message);

future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {

@Override

public void onSuccess(SendResult<String, String> result) {

System.out.println("Sent message=[" + message +

"] with offset=[" + result.getRecordMetadata().offset() + "]");

}

@Override

public void onFailure(Throwable ex) {

System.out.println("Unable to send message=["

+ message + "] due to : " + ex.getMessage());

}

});

}

## Consumer

### Consumer Configuration

For consuming messages, we need to configure a [ConsumerFactory](https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/core/ConsumerFactory.html) and a [KafkaListenerContainerFactory](https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/config/KafkaListenerContainerFactory.html). Once these beans are available in the Spring bean factory, POJO-based consumers can be configured using [@KafkaListener](https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/KafkaListener.html) annotation.

**[@EnableKafka](https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/EnableKafka.html)**annotation is required on the configuration class to enable detection of**@KafkaListener**annotation on spring-managed beans****:

@EnableKafka@Configurationpublic class KafkaConsumerConfig {

@Bean

public ConsumerFactory<String, String> consumerFactory() {

Map<String, Object> props = new HashMap<>();

props.put(

ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG,

bootstrapAddress);

props.put(

ConsumerConfig.GROUP\_ID\_CONFIG,

groupId);

props.put(

ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG,

StringDeserializer.class);

props.put(

ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG,

StringDeserializer.class);

return new DefaultKafkaConsumerFactory<>(props);

}

@Bean

public ConcurrentKafkaListenerContainerFactory<String, String>

kafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, String> factory =

new ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(consumerFactory());

return factory;

}

}

### Consuming Messages

@KafkaListener(topics = "topicName", groupId = "foo")public void listenGroupFoo(String message) {

System.out.println("Received Message in group foo: " + message);

}

****We can implement multiple listeners for a topic****, each with a different group Id. Furthermore, one consumer can listen for messages from various topics:

@KafkaListener(topics = "topic1, topic2", groupId = "foo")

Spring also supports retrieval of one or more message headers using the [@Header](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/messaging/handler/annotation/Header.html) annotation in the listener:

@KafkaListener(topics = "topicName")public void listenWithHeaders(

@Payload String message,

@Header(KafkaHeaders.RECEIVED\_PARTITION\_ID) int partition) {

System.out.println(

"Received Message: " + message"

+ "from partition: " + partition);

}

### Consuming Messages from a Specific Partition

Notice that we created the topic baeldung with only one partition.

For a topic with multiple partitions, however, a @KafkaListener can explicitly subscribe to a particular partition of a topic with an initial offset:

@KafkaListener(

topicPartitions = @TopicPartition(topic = "topicName",

partitionOffsets = {

@PartitionOffset(partition = "0", initialOffset = "0"),

@PartitionOffset(partition = "3", initialOffset = "0")}),

containerFactory = "partitionsKafkaListenerContainerFactory")public void listenToPartition(

@Payload String message,

@Header(KafkaHeaders.RECEIVED\_PARTITION\_ID) int partition) {

System.out.println(

"Received Message: " + message"

+ "from partition: " + partition);

}

Since the initialOffset has been set to 0 in this listener, all the previously consumed messages from partitions 0 and 3 will be re-consumed every time this listener is initialized.

If we don't need to set the offset, we can use the partitions property of @TopicPartition annotation to set only the partitions without the offset:

@KafkaListener(topicPartitions

= @TopicPartition(topic = "topicName", partitions = { "0", "1" }))

### Adding Message Filter for Listeners

We can configure listeners to consume specific types of messages by adding a custom filter. This can be done by setting a [RecordFilterStrategy](https://docs.spring.io/spring-kafka/api/org/springframework/kafka/listener/adapter/RecordFilterStrategy.html) to the KafkaListenerContainerFactory:

@Beanpublic ConcurrentKafkaListenerContainerFactory<String, String>

filterKafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, String> factory =

new ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(consumerFactory());

factory.setRecordFilterStrategy(

record -> record.value().contains("World"));

return factory;

}

We can then configure a listener to use this container factory:

@KafkaListener(

topics = "topicName",

containerFactory = "filterKafkaListenerContainerFactory")public void listenWithFilter(String message) {

System.out.println("Received Message in filtered listener: " + message);

}

In this listener, all the ****messages matching the filter will be discarded.****

### Custom Message Converters

So far, we have only covered sending and receiving Strings as messages. However, we can also send and receive custom Java objects. This requires configuring appropriate serializer in ProducerFactory and deserializer in ConsumerFactory.

Let's look at a simple bean class, which we will send as messages:

public class Greeting {

private String msg;

private String name;

// standard getters, setters and constructor

}

#### Producing Custom Messages

In this example, we will use [JsonSerializer](https://docs.spring.io/spring-kafka/api/org/springframework/kafka/support/serializer/JsonSerializer.html).

Let's look at the code for ProducerFactory and KafkaTemplate:

@Beanpublic ProducerFactory<String, Greeting> greetingProducerFactory() {

// ...

configProps.put(

ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,

JsonSerializer.class);

return new DefaultKafkaProducerFactory<>(configProps);

}

@Beanpublic KafkaTemplate<String, Greeting> greetingKafkaTemplate() {

return new KafkaTemplate<>(greetingProducerFactory());

}

We can use this new KafkaTemplate to send the Greeting message:

kafkaTemplate.send(topicName, new Greeting("Hello", "World"));

#### Consuming Custom Messages

Similarly, let's modify the ConsumerFactory and KafkaListenerContainerFactory to deserialize the Greeting message correctly:

@Beanpublic ConsumerFactory<String, Greeting> greetingConsumerFactory() {

// ...

return new DefaultKafkaConsumerFactory<>(

props,

new StringDeserializer(),

new JsonDeserializer<>(Greeting.class));

}

@Beanpublic ConcurrentKafkaListenerContainerFactory<String, Greeting>

greetingKafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, Greeting> factory =

new ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(greetingConsumerFactory());

return factory;

}

The spring-kafka JSON serializer and deserializer uses the [Jackson](https://www.baeldung.com/jackson) library, which is also an optional Maven dependency for the spring-kafka project.

So, let's add it to our pom.xml:

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.7</version>

</dependency>

Instead of using the latest version of Jackson, it's recommended to use the version that is added to the pom.xml of spring-kafka.

Finally, we need to write a listener to consume Greeting messages:

@KafkaListener(

topics = "topicName",

containerFactory = "greetingKafkaListenerContainerFactory")public void greetingListener(Greeting greeting) {

// process greeting message

}