

You have **2** free member-only stories left this month. [Upgrade for unlimited access.](#)

Async Messaging With Kotlin, Kafka and Docker

Simple message producer and consumer in Kotlin to interact with the Apache Kafka streaming platform



Nuno Brites

Dec 31, 2020 · 7 min read ★

ASYNC MESSAGING WITH KOTLIN, KAFKA AND DOCKER

A simple message producer and consumer in Kotlin to interact with the Kafka streaming platform

Producer



Consumer

In this article we will create a simple consumer and producer to interact with the Kafka platform, but before we start to write code and dive into exchanging messages let's

have a brief introduction to [Kafka](#).

For this introduction, I will use as the main source of reference the Kafka official documentation to pick some of the concepts we will be using in this article if you want to get more details regarding Kafka you can go through the website [intro](#) and [technical documentation](#). Let's start!

Kafka is an open-source distributed streaming platform. But what does that mean concretely?

Technically speaking, event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Source: [Kafka Documentation — Introduction](#)

This means there is a real-life happening that is represented and stored in the platform in the form of an **event**,

*An **event** records the fact that “something happened” in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers.*

Source: [Kafka Documentation — Introduction](#)

When the event is written into the platform it must be stored concretely “somewhere” so it can be accessed by some other parts at a future point in time. That “somewhere” is called a **topic**.

*Events are organized and durably stored in **topics**. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be “payments”. Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as*

needed — unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Source: [Kafka Documentation — Introduction](#)


To make use of the platform features we have talked about before there needs to be a part that writes and a part that reads from a **topic**, they are called **producer** and **consumer** respectively.

***Producers** are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.*

Source: [Kafka Documentation — Introduction](#)

Putting together all the concepts mentioned above we have a way to establish event-based asynchronous communication.

Kafka is a powerful distributed event streaming platform but it can also work as a message broker, the choice of how you use it should be done according to the scenario you have in your hands. If you need to stream events in real-time and guarantee that those events need to be persisted in a distributed manner definitely go with Kafka, but if you just need to decouple asynchronously from another system maybe you are good to go with a more traditional message broker like RabbitMQ.

 Check [Async Messaging With Kotlin and RabbitMQ](#) for more details on using RabbitMQ message broker with Kotlin.

In this article, we will be using Kafka as an alternative to a traditional message broker and only that. This article will not approach Kafka's potential as a full-fledged distributed streaming platform.

We'll use a Docker image with Kafka to run our Kafka and a Docker image with Zookeeper to deal with the synchronization of the Kafka instance.

We'll use a Docker image with Kafdrop as our web UI layer to inspect what is happening in the running Kafka instance.

Kotlin will be our programming language to implement the logic to produce/consume messages to/from Kafka.

Let's recap the steps and put our hands to work:

- Setup Kafka, Zookeeper, and Kafdrop instances using Docker
- Setup Kotlin project with IntelliJ and Gradle
- Create Kotlin producer and consumer

Setup Kafka and Kafdrop instances using Docker

To set up our environment we'll use docker-compose for defining and running the following containers.

- Kafka —the Kafka broker container
- Zookeeper — container responsible for the synchronization in the Kafka instance
- Kafdrop — the container with the UI to monitor what is happening inside the Kafka instance

Start by creating a file `docker-compose.yml` and add the lines below.

```
1  version: '2'
2  services:
3    zookeeper:
4      image: wurstmeister/zookeeper
5      ports:
6        - "2181:2181"
7      environment:
8        ZOOKEEPER_CLIENT_PORT: 2181
9        ZOOKEEPER_TICK_TIME: 2000
10   kafka:
11     image: wurstmeister/kafka:2.13-2.6.0
12     depends_on:
13       - zookeeper
14     ports:
```

```

15     - "9092:9092"
16     - "9093"
17   environment:
18     KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
19     KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9093,OUTSIDE://localhost:9092
20     KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
21     KAFKA_LISTENERS: INSIDE://:9093,OUTSIDE://:9092
22     KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
23     KAFKA_CREATE_TOPICS: "Topic1:1:1,Topic2:1:1:compact"
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  kafdrop:
    image: obsidiandynamics/kafdrop
    ports:
      - "9000:9000"
    environment:
      KAFKA_BROKERCONNECT: kafka:9093
      JVM_OPTS: "-Xms32M -Xmx64M"
      SERVER_SERVLET_CONTEXTPATH: "/"
    depends_on:
      - kafka

```


docker-compose.yml hosted with ❤ by GitHub

[view raw](#)

Docker Compose

The file above shows the configurations for the three containers we will be using. The main highlights in this file are:

- The Kafka listeners configurations where the Kafka broker container is exposed to the Docker internal network through port 9093 and to the exterior through port 9092.
- Kafdrop (inside the Docker network) will use port 9093 to access the broker and will expose its user interface through port 9000 (accessible through the browser).
- Kotlin producer/consumer (outside the Docker network) will use port 9092 to connect and interact with the broker.
- Two topics are created: `Topic1` will have 1 partition and 1 replica, `Topic2` will also have 1 partition and 1 replica but the cleanup policy will be set to `compact`.

 *For the sake of simplicity, this article doesn't address cleanup policies in Kafka topics. Setting a cleanup policy for a topic would allow, for example, to enforce a size or time after*

which each message would start to be removed from the topic. For more details check the [Apache Kafka Documentation on log compaction](#).

To start our setup run `docker-compose up` and the three containers will start.

After all the containers are up and running you should be able to navigate to `localhost:9000` and see Kafdrop up and running like in the image below.

The screenshot shows the Kafdrop web interface. At the top left is the Kafdrop logo. In the top right corner, there is a 'Star' button and a version/status string: '3.27.0 [2020-06-21T23:16:06.428Z]'. The main heading is 'Kafka Cluster Overview'. Below this, there are two main sections: 'Bootstrap servers' and 'Total topics', 'Total partitions', 'Total preferred partition leader', and 'Total under-replicated partitions'. The 'Brokers' section shows a table with columns: ID, Host, Port, Rack, Controller, and Number of partitions (% of total). The 'Topics' section shows a table with columns: Name, Partitions, % Preferred, # Under-replicated, and Custom Config. There is a search bar for topics and a '+ New' button at the bottom left.

Bootstrap servers		kafka:9093
Total topics	2	
Total partitions	2	
Total preferred partition leader	100%	
Total under-replicated partitions	0	

ID	Host	Port	Rack	Controller	Number of partitions (% of total)
1001	kafka	9093	-	Yes	2 (100%)

Name	Partitions	% Preferred	# Under-replicated	Custom Config
Topic1	1	100%	0	No
Topic2	1	100%	0	Yes

Kafdrop — Kafka Cluster Overview

Now that we have set up our containers let's set up the Kotlin project and create our producer and consumer.


Setup Kotlin project with IntelliJ and Gradle

Let's start by initializing the project in IntelliJ by using the Kotlin plugin and adding the following dependency to the `build.gradle.kts` file.

```
implementation("com.fasterxml.jackson.core:jackson-databind:2.9.6")
implementation("org.apache.kafka:kafka-clients:2.6.0")
```

The first dependency is required for the message exchanging between our producer/consumer and Kafka.

The second dependency is the Kafka client library provided by Apache.

 *Make sure the version of your Kafka client matches the version of your Kafka broker in the container. In this case, we'll use version 2.6.0.*

Now that we have set up our project in IntelliJ let's start by coding our consumer.

Consumer

The consumer code will consume the messages being produced to `Topic1` and write their content to the console.

```
1  import org.apache.kafka.clients.consumer.KafkaConsumer
2  import org.apache.kafka.clients.consumer.Consumer
3  import org.apache.kafka.common.serialization.StringDeserializer
4  import java.time.Duration
5  import java.util.Properties;
6
7  private fun createConsumer(): Consumer<String, String> {
8      val props = Properties()
9      props["bootstrap.servers"] = "localhost:9092"
10     props["group.id"] = "hello-world"
11     props["key.deserializer"] = StringDeserializer::class.java
12     props["value.deserializer"] = StringDeserializer::class.java
13     return KafkaConsumer(props)
14 }
15
16 fun main() {
17     val consumer = createConsumer()
18     consumer.subscribe(listOf("Topic1"))
19
20     while (true) {
21         val records = consumer.poll(Duration.ofSeconds(1))
22         println("Consumed ${records.count()} records")
23
24         records.iterator().forEach {
25             val message = it.value()
26             println("Message: $message")
27         }
28     }
29 }
```

Kotlin — Kafka Consumer

The consumer will subscribe to `Topic1` and will poll every second for new messages. When new messages are found the content is written to the console.

Producer

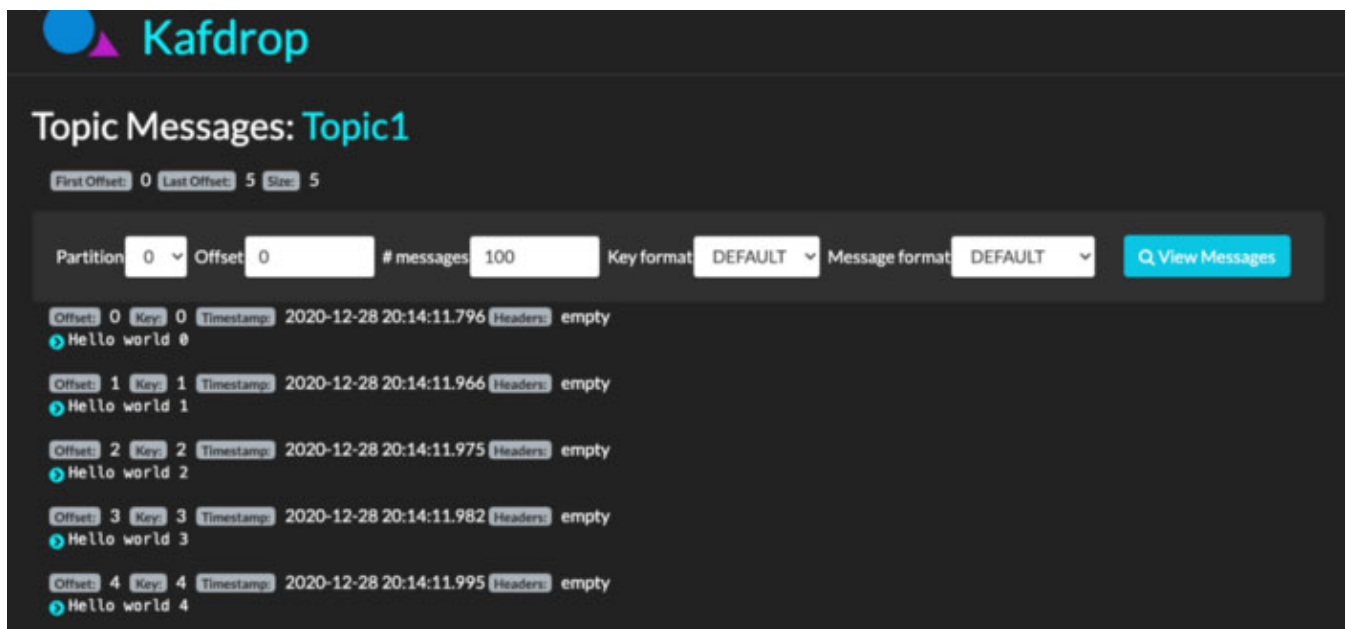
Since we now have the consumer waiting for messages to be written, let's create the producer and add a for loop that will write 5 messages to the Kafka topic with the name `Topic1`.

```
1  import org.apache.kafka.clients.producer.KafkaProducer
2  import org.apache.kafka.clients.producer.Producer
3  import org.apache.kafka.clients.producer.ProducerRecord
4  import org.apache.kafka.common.serialization.StringSerializer
5  import java.util.Properties;
6
7  private fun createProducer(): Producer<String, String> {
8      val props = Properties()
9      props["bootstrap.servers"] = "localhost:9092"
10     props["key.serializer"] = StringSerializer::class.java
11     props["value.serializer"] = StringSerializer::class.java
12     return KafkaProducer<String, String>(props)
13 }
14
15 fun main() {
16     val producer = createProducer()
17     for (i in 0..4) {
18         val future = producer.send(ProducerRecord("Topic1", i.toString(), "Hello world $i"))
19         future.get()
20     }
21 }
```

Kotlin — Kafka Producer

Run the producer code and check Kafdrop, go to the `Topic1` [messages](#) page, and click on the button 'View Messages'. You should see something similar to the image below where the 5 messages are listed.





Kafdrop — Topic 1 messages

The same messages should also appear in the consumer console.

Wrapping Up

After experimenting with the code above, here are some points worth mentioning:

- Even though Kafka main purpose is to be an event streaming platform it can also be used as a message broker.
- The Docker networking can be a bit tricky to set up if you're new to Docker due to the inside/outside network configuration for Kafka listeners.
- With a few lines of code, we can write a producer and a consumer in Kotlin that makes use of the Apache Kafka Client library and interacts with Kafka.
- The provided producer and consumer are just the first steps for writing/reading messages to/from Kafka; much more can be done and explored.

For more details related to the topics in this article check the following books:

- [Kotlin in Action](#)
- [Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale](#)
- [Docker: Up & Running: Shipping Reliable Containers in Production](#)

In case you are more of an audiobook person check [Audible](#) .

If you have any suggestions or contributions feel free to comment below. Happy coding!

Sign up for Top 10 Stories

By The Startup

Get smarter at building your thing. Subscribe to receive The Startup's top 10 most read stories — delivered straight into your inbox, once a week. [Take a look.](#)

Get this newsletter

Emails will be sent to enrico.alberti.18@gmail.com.
[Not you?](#)

[Kotlin](#) [Kafka](#) [Asynchronous](#) [Messaging](#) [Programming](#)

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

