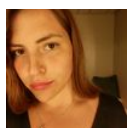



Stay safe, friends. Learn to code from home. Use our free 2,000 hour curriculum.

22 JANUARY 2019 / [#BIG DATA](#)

What to consider for painless Apache Kafka integration



Adi Polak

 Software Developer  Blogger  Speaker  1 of 25 influential women in Software Development



Apache Kafka's real-world adoption is exploding, and it claims to dominate the world of stream data. It has a

keeps on growing. But, it can be painful too. So, just before jumping head first and fully integrating with Apache Kafka, let's check the water and plan ahead for painless integration.



nPhoto by [Helloquence](#) on [Unsplash](#)

What is it?

Apache Kafka is an open source framework for asynchronous messaging and it's a distributed streaming platform. It is TCP based. The messages are persisted in topics. Message producers are called *publishers* and message consumers are called *subscribers*.

Consumers can subscribe to **one or more topics** and consume all the messages in that topic. Messages are written into the topic partitions.

many consumers that subscribe to the data written to it. For each topic Kafka maintains a partition log. Metadata for the partition's logs and topics are usually managed by Zookeeper.

If you would like to learn more about Kafka message delivery semantics — like, *at most once*, *at least once* and *exactly once* — read [here](#).

Many tech companies have already integrated Apache Kafka into their production as a **message broker**, **user activities tracking pipeline**, **metrics gatherer**, **log aggregation mechanism**, **stream processing device** and much more. Apache Kafka is written in Scala and Java.

Why Kafka?

- *Kafka provides **High Availability** and **Fault Tolerance** message logs.* Kafka clusters retain all published records. It is by default **persistent** — If you don't set a limit for Kafka, it will keep records until it runs out of disk space. When **data loss** means awful failure for the product, this is essential for recovery.
- **Multiple Topic Consumers** — when configuring the consumers under multiple consumers groups, it helps to reduce the old bottleneck of sending the data to multiple applications for processing. Kafka is distributed, hence, it can send information to consumers from various physical machines/services instances. Replicating topics to a secondary cluster is also relatively easy using Apache Kafka's mirroring feature, MirrorMaker — see an [example](#) of mirroring data between two HDInsight clusters. **Just remember**, if multiple consumers are

data will be balanced over all the consumers within the group.

- Kafka is **polyglot** – there are many clients in C#, Java, C, python and more. The ecosystem also provides a REST proxy which allows easy integration via HTTP and JSON.
- **Real-Time Handling** – Kafka can handle real-time data pipelines for real time messaging for applications.
- **Scalable** – due to distributed architecture, Kafka can scale out without incurring any downtime.
- and more...

Let's make integration with Kafka painless



Here are 6 things to know before integrating:

1 — Apache Zookeeper can become a pain point with a Kafka cluster

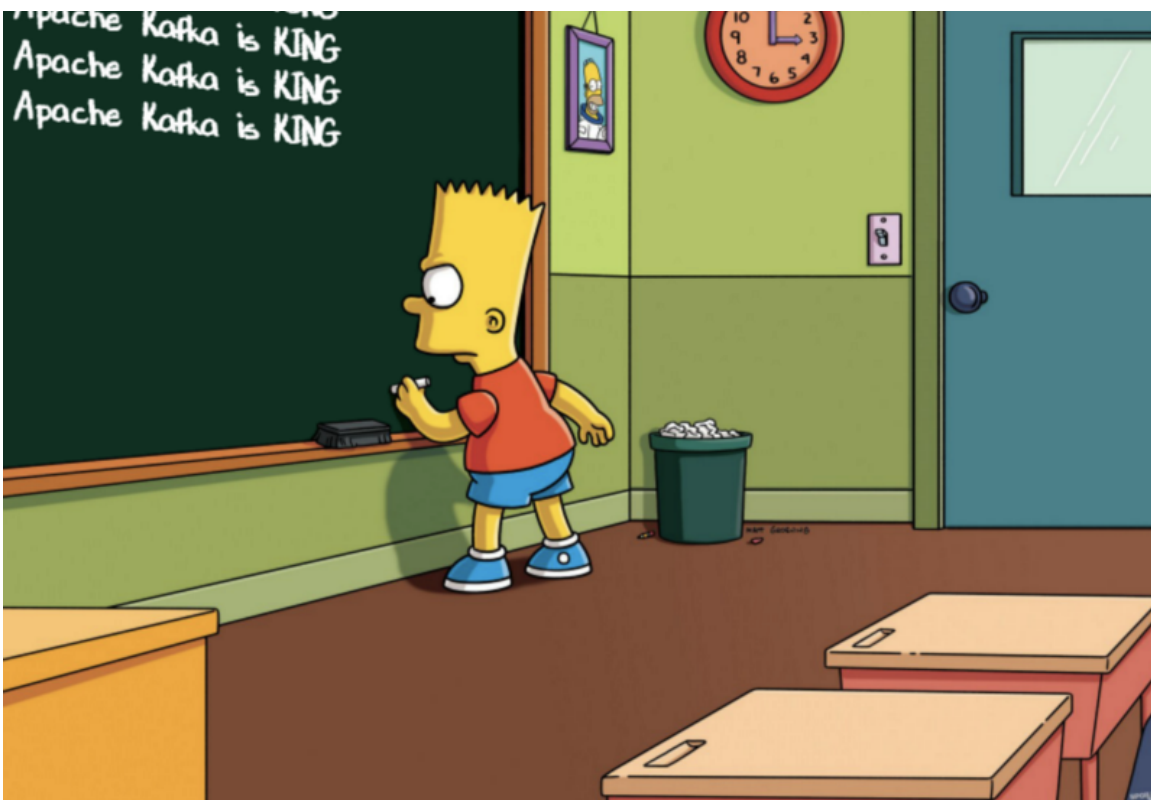
In the past (versions < 0.81) Kafka used Zookeeper to maintain offsets of each topic and partition. Zookeeper used to take part in the read path, where too frequent commits and too many consumers led to sever performance and stability issues.

On top of that, it is better to use commits manually with old Zookeeper-based consumers, since careless auto-commits could lead to data loss.

The newer versions of Kafka offer their own management, where the consumer can use Kafka itself to manage offsets. This means that there is a specific topic that manages the read offsets instead of Zookeeper.

Yet, Kafka still needs a cluster with Zookeeper, even in the later versions 2.+. Zookeeper is used to store Kafka configs (reassigning partitions when needed) and the Kafka topics API, like create topic, add partition, etc.

The load on Kafka is strictly related to the number of consumers, brokers, partitions and frequency of commits from the consumer.



2 – You shouldn't send large messages or payloads through Kafka

According to Apache Kafka, for better throughput, the max message size should be **10KB**. If the messages are larger than this, it is better to check the alternatives or find a way to chop the message into smaller parts before writing to Kafka. Best practice to do so is using a message key to make sure all chopped messages will be written to the same partition.

3 – Apache Kafka can't transform data

Many developers are mistaken and think that they can create Kafka parsers or do a data transformation over Kafka. However, Kafka does not enable transformation of data. If you are using Azure services, there is a great list of [data factories services](#) that you can use to transform the data like [Azure Databricks](#), [HDInsights Spark](#) and others that connects to Kafka.

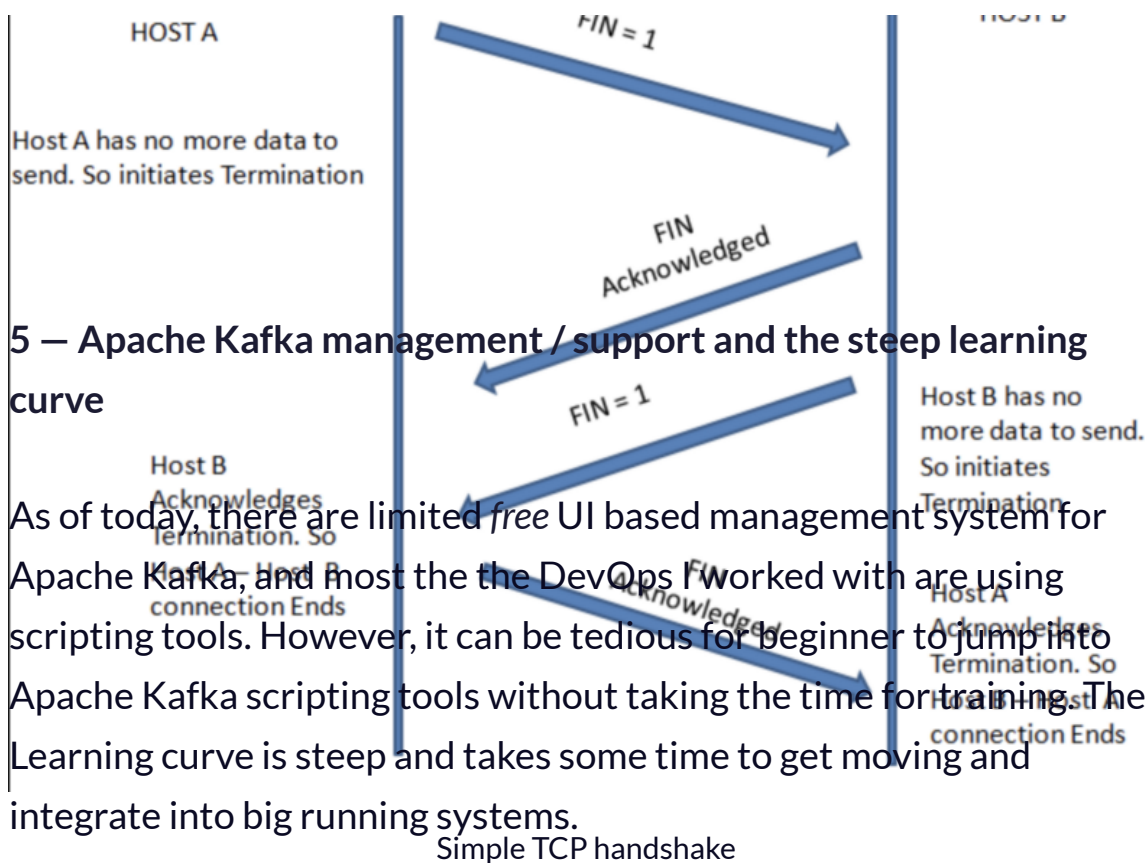
API that is build on top of Kafka's producer and consumer clients. It's significantly more powerful and also more expressive than the Kafka consumer client.

The KafkaStreams client allows us to perform continuous computation on input coming from one or more input topics and sends output to zero, one, or more output topics. Internally a KafkaStreams instance contains a normal KafkaProducer and KafkaConsumer instance that is used for reading input and writing output.

Another option is using Flink, check it out [here](#).

4 – Apache Kafka supports a binary protocol over TCP

Apache Kafka communication protocol is TCP based. It doesn't support MQTT or JMS or other non-based TCP protocols out of the box. However, many users have written adaptors to read data from those protocols and write to Apache Kafka. For example [kafka-jms-client](#).



As of today, there are limited free UI based management system for Apache Kafka, and most the the DevOps I worked with are using scripting tools. However, it can be tedious for beginner to jump into Apache Kafka scripting tools without taking the time for training. The Learning curve is steep and takes some time to get moving and integrate into big running systems.

For experienced DevOps/ developers it might take a few months (2+) to fully understand how to integrate, support and work with Apache Kafka. It is important to learn how Kafka works in order to use the configuration in the way that will best suit the system's needs.

Here's a list of management tools that you can use for **almost free** (some are restricted to personal/community use):

- KafkaTool – GUI application for managing and using **Apache Kafka** clusters.
- Confluent platform – full enterprise streaming platform solution.
- KafDrop – tool for displaying information such as brokers, topics, partitions, and even lets you view messages. It is a lightweight application that runs on Spring Boot and requires very little configuration.

it offers much less than the rest.

Supporting Managed Kafka on the cloud

Today almost all clouds support Kafka, if it is fully managed or using integration with Confluent from the cloud store up to just purchasing Kafka machines:

- [Confluent Cloud- Kafka as a Service](#)
- [Azure Event Hub- fully managed Kafka](#)
- [Managed Kafka on HDInsight — Azure](#)
- [Kafka Machine on Google cloud](#)
- [Kafka on AWS using Confluent solution](#)
- ... many more

6 — Kafka is no magic — There is still a possibility of data loss

Apache Kafka is probably the most popular tool for distributed asynchronous messaging. This is mainly due to his **high throughput, low latency, scalability, centralised and real time** abilities. Most of this is due to using data replicas which in Kafka are called partitions.

However, with misconfiguration there is a high chance of *data loss* when machines/processes are failing, and they will fail. Therefore, it's important to understand how Kafka works and what the product/system requirements are.

To assist you in finding the right configuration, the Kafka team created Trogdor. Trogdor is a failure testing framework.

How it works

- Configure Kafka the way you would in production
- Create a producer that generates messages with sequence 1... X million.
- Run the producer
- Run the consumer
- Create failure by crashing and/or hanging broker.
- Test and check that every event produced was consumed.
- ... if that's not the case, it is better to go back and update the configuration accordingly!

On top of that, it is important to remember that Apache Kafka ...

- Is *not a RPC* — Apache Kafka is a messaging system. For RPC, service X needs to be aware of Service Y and the call signature. For example, in Kafka, if you send a message it doesn't mean that someone will consume it, ever. In RPC, there is always a consumer since the service itself is aware of the consumer Y and creates a call to its signature/function.
- It is *not a Database* — it's not a good place to save messages since you can't jump between them or create a search without an expensive full scan.

An interesting library brought to us by the Confluent Community is [KSQL](#). It is build on top of Kafka stream. KSQL is a completely interactive SQL interface. You can use it without writing any code. KSQL is under the Confluent Community licensing.

TL;DR

Apache Kafka has many benefits, yet before adding it in production, one should be aware that:

- It has a steep learning curve — make time to learn the bits and bits of Kafka
- You must manage cluster resources — be aware of the requirements like Zookeeper
- You can still lose data with Apache Kafka
- Most clouds provide managed Apache Kafka
- It won't transform data
- It's not a Database
- It support binary protocol over TCP protocol
- At the moment, you can't sent large messages using Kafka
- You should use Trogdor for fault testing of your system

All that being said, Apache Kafka is probably the best tool for messaging and streaming tasks.

Thank you [Gwen Shapira](#) for your input and guidance along the way.

If you enjoyed this story, please click the ? button. Feel free to leave a



[Follow me](#) here, or [here](#) for more posts about Scala, Kotlin, Big data, clean code and software engineers nonsense. Cheers!

If you read this far, tweet to the author to show them you care.

[Tweet a thanks](#)

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

[Get started](#)

Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can [make a tax-deductible donation here](#).

Our Nonprofit

[About](#)

[Alumni Network](#)

[Open Source](#)

[Shop](#)

[Support](#)

[Sponsors](#)

[Academic Honesty](#)

[Code of Conduct](#)

[Privacy Policy](#)

[Terms of Service](#)

[Copyright Policy](#)

Trending Guides

[2019 Web Developer Roadmap](#)

[Python Tutorial](#)

[CSS Flexbox Guide](#)

[JavaScript Tutorial](#)

[Python Example](#)

[HTML Tutorial](#)

[Linux Command Line Guide](#)

[JavaScript Example](#)

[Git Tutorial](#)

[React Tutorial](#)

[Java Tutorial](#)

[Linux Tutorial](#)

[CSS Tutorial](#)

[jQuery Example](#)

[SQL Tutorial](#)

[CSS Example](#)

[React Example](#)

[Bootstrap Example](#)

[How to Set Up SSH Keys](#)

[WordPress Tutorial](#)

[PHP Example](#)