

**Florimond Manca**

Tutorials

Essays

Retrospectives

Sep 20, 2018 | in Tutorials  kafka python Edit

# Building A Streaming Fraud Detection System With Kafka And Python

A thorough tutorial to build your first real-world Kafka app in Docker Compose. Welcome to the land of stream processing!



*Orange and multicolored LEGO toy set. @lh1me via unsplash.com*

---

This is the second article of my series on **building streaming applications with Apache Kafka**. If you missed it, you may read [the opening](#) to know why this series even exists and what to expect.

This time, we will get our hands dirty and **create our first streaming application** backed by Apache Kafka using a Python client. I will try and make it as close as possible to a **real-world** Kafka application.

I wrote this blog post as a **tutorial**. You won't have to know anything about stream processing nor Apache Kafka — I'll explain notions and terminology as we go — and you'll be able to follow along as we build the system.

What we'll build is a **real-time fraud detection system**. We'll do so **from scratch**. We will generate a stream of transactions and process those to detect which ones are potential fraud.

Sounds exciting? I hope so!

You will find the complete code supporting this blog post on [GitHub](#).

## Prerequisites

I've built this post so you can **follow along** and build the various apps with me. However, there will be some prerequisites.

First, we'll need to run a Kafka cluster locally. This is generally not too resource intensive, but you'll probably need a decent development machine.

Next, we'll run the cluster using **Docker Compose** because this will bring us as close as possible to a production system — the *real world*. So make sure you have [Docker](#) and [Docker Compose](#) installed.

Because of this, I also expect you to have a **some knowledge of Docker and Docker Compose**. Although I'll explain the finer details, you should at least know how to read a `Dockerfile` and a `docker-compose.yml` file.

Lastly, because we'll build the app with **Python**, the final prerequisite is to have a basic knowledge of the Python language and ecosystem.

That's pretty much it, so let's get down to business!

# Firing up a local Kafka cluster

Apache Kafka is a piece of software which, as all pieces of software, runs on **actual computers** — your own computer for the sake of this blog post.

First, a bit of **terminology**. Kafka being a **distributed system**, it runs in a **cluster**, i.e. multiple computers (a.k.a. **nodes**) that communicate with one another. In Kafka jargon, nodes are called **brokers**. Your computer will be the only broker here.

What does Kafka need in order to run? Actually, it only needs two things — a Kafka broker and a **Zookeeper** instance.

"What is Zookeeper?!"

Keep calm — you don't really need to know what it is. But in a nutshell, it is a coordination software, distributed as well, used by Apache Kafka to keep track of the cluster state and members. It is an essential component of any Kafka cluster, but we won't talk about it much more than this.

In introduction, I mentioned we were going to use **Docker Compose** to run the cluster. Let's see how that goes.

## Docker Compose configuration

Here's the `docker-compose.yml` file we'll use. It uses Docker images from [Confluent Platform](#) — [Confluent](#) being a major actor in the Apache Kafka community — and was inspired by their

## Kafka single node example.

It has two services: one for the Kafka broker and one for the Zookeeper instance.

```
version: "3"

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

I'll break it down what's happening here — but if you just want to run it, you can also skip to the next part.

The `zookeeper` service is — wait for it — for **Zookeeper**. It uses Confluent Platform's Docker image, `cp-zookeeper`. The two environment variables define which port Zookeeper will be accessible on (which we'll pass to the broker), and its basic unit of time (a required configuration detail).

Second, the `kafka` service is our **Kafka broker**. Among the few environment variables, the two we need to care about are `KAFKA_ZOOKEEPER_CONNECT`, which tells the broker where it can find Zookeeper, and `KAFKA_ADVERTISED_LISTENERS`, which defines where we'll be able to connect to the broker from other applications. You don't need to care about the replication factor; I'll just tell you that it's set to 1 because we have only one broker in the cluster.

Now that we have defined the cluster's Docker Compose configuration, let's just...

## Spin up the cluster!

It's now time to **spin up our local Kafka cluster**. How? If you know about Docker Compose, it's straight-forward. Just go to your project root (where `docker-compose.yml` lives) and run:

```
$ docker-compose up
```

You'll see a huge bunch of logs in the console — don't worry! That's normal. Kafka and Zookeeper are quite verbose when starting up. To know when they're finished initialising, you can run this in a separate terminal. It waits for the `broker` service to output a "started" message.

```
$ docker-compose logs -f broker | grep started  
broker_1 | INFO [KafkaServer id=1] started (kafka.server.KafkaServer)
```

If that happens, congrats! You now have a single-node Kafka cluster up and running. It's probably not very useful by itself. How about we start to stream some data?

## Enter: the fraud detection system

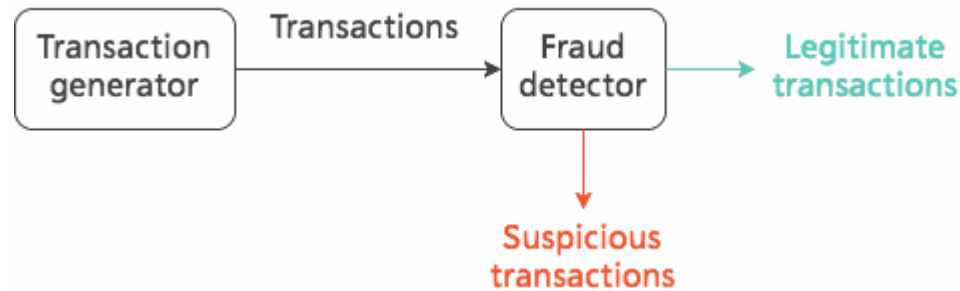
That's right — now is the time we get to build things! 🍰

I could have gone through the making of a "Hello, World!" streaming application, in which you'd type some messages on one end (the producer) and receive them in real-time on the other end (the consumer). But there is already this exact tutorial in [Confluent's Docker Quickstart](#) and elsewhere.

Instead, I want us to build a **real system** solving an **actual business use case**. That's not an easy task while controlling the scope of the post, but we'll see how it goes.

The system we'll build is a simple **fraud detection mechanism** — the same kind of mechanism you may encounter in banking and transaction monitoring systems.

We'll produce fake transactions on one end, and filter and log those that look suspicious on the other end. This will involve a **transaction generator** (simply because we need material to process) and a **fraud detector**. Both applications will run in Docker containers and interact with our Kafka cluster.



*Block diagram of the fraud detection system.*

Now that we've seen the basic idea behind this system, let's start writing some code, shall we?

## Creating app skeletons

We'll start by creating the containerised skeletons for the transaction generator and fraud detector. They will be straight-forward Python applications, so we'll use the same `Dockerfile` in both cases:

```
# Dockerfile
FROM python:3.6

WORKDIR /usr/app

ADD ./requirements.txt ./
RUN pip install -r requirements.txt
ADD ./ ./

CMD ["python", "app.py"]
```



We'll create two folders containing this `Dockerfile` : one for the transaction generator and one for the fraud detector.

We're also going to have one Python dependency — [Kafka Python](#), a Python client for Kafka (among others). We'll use it to build applications that interact with the Kafka cluster.

```
# requirements.txt
kafka-python
```

The actual Python code will live in the `app.py` file in each app. So in the end, this is folder structure we end up with:

```
.
├── docker-compose.yml
├── detector
│   ├── Dockerfile
│   ├── app.py
│   └── requirements.txt
└── generator
    ├── Dockerfile
    ├── app.py
    └── requirements.txt
```

In just a moment, we will integrate these services into our development environment. But before that, let me take you through the process of putting Kafka into its own Docker Compose, which will allow us to use it as we'd use an actual production cluster.

# Isolating the Kafka cluster

In production, **Kafka is always used as a service**. By this, I mean that the cluster runs **independently** of the applications that use it. The latter simply **connect** to the cluster to publish or read events from it.

We're trying to build a **real-world system**, so how about we make our applications interact with an independant, production-like Kafka cluster too?

To do this locally, we're going to move our `zookeeper` and `broker` services to a **separate Docker Compose configuration**. Let's put it in a `docker-compose.kafka.yml` file. Later on, we'll use the regular `docker-compose.yml` for our application services only.

```
# docker-compose.kafka.yml
version: "3"

services:
  # ... zookeeper and broker as before ...
```

While you may think this is enough, there's something missing. To allow both compositions to access the **same network** (remember each Docker Compose file uses their own private network by default), we'll use an **external Docker network**. We'll call it `kafka-network`. Let's go ahead and create that network now:

```
$ docker network create kafka-network
85351a30f253b0fb8ae197ea07ddeddd0412c8a789e910dd4a3f424057122d30
```

We can now add the `networks` section in `docker-compose.kafka.yml` :

```
# docker-compose.kafka.yml
version: "3"

services:
  # ... zookeeper and broker as before ...

# NEW
networks:
  default:
    external:
      name: kafka-network
```

(You can read more about [networking in Docker Compose](#) if you wish).

Cool, we now have the Kafka cluster in a separate Docker Compose configuration. *So what?*

Well, this means we'll be able to treat the cluster as a **completely isolated system**. You can spin it up or tear it down independently from the rest of your Docker applications — just like what happens in the real world.

In practice, manipulating this Docker Compose will require us to use the `-f` flag to specify which configuration we use.

For example, to spin up the Kafka cluster, you'll now use:

```
docker-compose -f docker-compose.kafka.yml up
```

And if you want to access logs:

```
docker-compose -f docker-compose.kafka.yml logs broker
```

You get the gist.

Now that this is done, we can create an empty `docker-compose.yml` with the same network configuration, to which we'll add the transaction generator and the fraud detection services later on:

```
# docker-compose.yml
version: "3"

services:
  # TODO: generator and detector coming soon

# Give this composition access to the Kafka network
networks:
  default:
    external:
      name: kafka-network
```

That's it for the diversion — I think we're all set! Let's start building the **streaming fraud detection system**.

# Building the transaction generator

To do stream processing, we need a stream to begin with. This is when the transaction generator comes in. We'll use it to create a **stream of transactions**.

In Kafka, there are a few different types of applications you can build. One category of applications is **producers**.

Producers do just what we need — they **publish messages** to the Kafka cluster. These messages will be stored on our broker in what we call a **topic** — a named on-disk store of messages.



*How a Kafka producer works. Simple, right?*

Now, I have some good news. The Python client we use (Kafka Python) allows us to build producers. 🐍

So let's use [Kafka Python's producer API](#) to send messages into a `transactions` topic.

It's going to be hard for me not to copy-paste some code here. This is not a tutorial about the Kafka Python client, so I'll just take you through the steps. If you want more details, we can simply refer to the [Kafka Python docs](#).

First, let's start by grabbing the `KafkaProducer` from the Kafka Python library.

```
from kafka import KafkaProducer
```

We then grab the URL of our broker (from an environment variable — best practices still apply). Producers use that URL to bootstrap their connection to the Kafka cluster.

```
import os

# ...
KAFKA_BROKER_URL = os.environ.get("KAFKA_BROKER_URL")
```

Then, we can instantiate the actual producer. It exposes a simple API (detailed in the [docs](#)) to send messages to a Kafka topic.

```
producer = KafkaProducer(bootstrap_servers=KAFKA_BROKER_URL)
```

The next step is — what do we do with the producer? Well, we can use it to send messages. What messages? Good question! Let's skip that for the moment, and say we send empty strings.

Now, the aim of the transaction generator is to *fake* an **infinite stream of transactions** flowing through our system. So we'll simply run an infinite loop that produces those messages to a topic.

Side note — I like to use recommendations from [How to paint a bike shed: Kafka topic naming conventions](#) to name topics. This is why I'll use `queueing.transactions` for the transactions topic

name, which effectively represents a queue of transactions to be processed.

Here's how we can implement such a loop:

```
from time import sleep

# ...
while True:
    message = "" # TODO
    # Kafka messages are plain bytes => need to `.encode()` the string message
    producer.send("queueing.transactions", value=message.encode())
    sleep(1) # Sleep for one second before producing the next transaction
```

In the end, here's the Python file we end up with:

```
# generator/app.py
import os
from time import sleep
from kafka import KafkaProducer

KAFKA_BROKER_URL = os.environ.get("KAFKA_BROKER_URL")

if __name__ == "__main__":
    producer = KafkaProducer(bootstrap_servers=KAFKA_BROKER_URL)
    while True:
        message = "" # TODO
        producer.send("queueing.transactions", value=message.encode())
        sleep(1)
```

Now, let's add the `generator` app to our `docker-compose.yml` configuration:

```
# docker-compose.yml
services:
  # ...
  generator:
    build: ./generator
    environment:
      KAFKA_BROKER_URL: broker:9092
```

Before we start this app, we need to populate the message with an **actual value representing a transaction**.

To do this, I have built a utility `create_random_transaction()` function which we can use to create randomised transactions. These will have a `source` account, a `target` account, an `amount` and a `currency` .

It's pretty straight-forward, so I'm just copy-pasting the `transactions.py` file here:

```
# generator/transactions.py
from random import choices, randint
from string import ascii_letters, digits

account_chars: str = digits + ascii_letters

def _random_account_id() -> str:
    """Return a random account number made of 12 characters."""
    return "".join(choices(account_chars, k=12))
```



```
def _random_amount() -> float:
    """Return a random amount between 1.00 and 1000.00."""
    return randint(100, 1000000) / 100

def create_random_transaction() -> dict:
    """Create a fake, randomised transaction."""
    return {
        "source": _random_account_id(),
        "target": _random_account_id(),
        "amount": _random_amount(),
        # Keep it simple: it's all euros
        "currency": "EUR",
    }
```

Nice! We can now replace the empty message with a fake transaction:

```
from transactions import create_random_transaction

# ...
transaction: dict = create_random_transaction()
message: str = json.dumps(transaction)
producer.send("queueing.transactions", value=message.encode())
# ...
```

Let's add the final touches by making our generator app more configurable — which is always a good practice. Let's:

- Extract the `queueing.transactions` topic into a `KAFKA_TRANSACTIONS_TOPIC` environment variable
- Make the sleep time configurable through a `TRANSACTIONS_PER_SECOND` environment variable.
- Make use of `KafkaProducer`'s `value_serializer` argument to automatically serialize messages from a dict to JSON bytes.

Here's the **final version of the transaction generator**:

```
# generator/app.py
import os
import json
from time import sleep
from kafka import KafkaProducer
from transactions import create_random_transaction

KAFKA_BROKER_URL = os.environ.get("KAFKA_BROKER_URL")
TRANSACTIONS_TOPIC = os.environ.get("TRANSACTIONS_TOPIC")
TRANSACTIONS_PER_SECOND = float(os.environ.get("TRANSACTIONS_PER_SECOND"))
SLEEP_TIME = 1 / TRANSACTIONS_PER_SECOND

if __name__ == "__main__":
    producer = KafkaProducer(
        bootstrap_servers=KAFKA_BROKER_URL,
        # Encode all values as JSON
        value_serializer=lambda value: json.dumps(value).encode(),
    )
    while True:
        transaction: dict = create_random_transaction()
        producer.send(TRANSACTIONS_TOPIC, value=transaction)
        print(transaction) # DEBUG
        sleep(SLEEP_TIME)
```

And here's what the Docker Compose service looks like:

```
# docker-compose.yml
services:
  generator:
    build: ./generator
    environment:
      KAFKA_BROKER_URL: broker:9092
      TRANSACTIONS_TOPIC: queueing.transactions
      TRANSACTIONS_PER_SECOND: 1000
```

## Trying out the transaction generator

At this point, it's probably a good idea to get things running and check that everything is going well. After all, who am I to tell you this works, if you can't verify it for yourself?

To do this, let's begin by making sure our local Kafka cluster is up, then start the main Docker Compose:

```
$ docker-compose up
```

This will start the `generator` 's producer.

Our debug `print s` will output the transactions to the console from Python, but this does not guarantee messages are effectively produced to the topic.

To verify this is the case, we can use a script built into Confluent Platform and present in the `confluent/kafka-cp` Docker image. This script is called `kafka-console-consumer`. It allows you to read and print the contents of a topic to the console.

The command needs to be run inside the `broker` container, which makes it a bit obscure, but here it is (you'll need to run it in a separate terminal):

```
docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bootstrap-server localhost:9092 --zookeeper-quorum
```

The `--from-beginning` flag means the command will read the whole topic, instead of only reading new messages.

You can let it run for a while, then press `ctrl+c` to stop the `kafka-console-consumer` and see the total number of read messages:

```
...
{"source": "Jpe2zq1QDYTn", "target": "lLbLn10FSr7T", "amount": 254.83, "currency": "EUR"}
{"source": "mfGWAdLKW8jh", "target": "ede8FQpjBeej", "amount": 785.22, "currency": "EUR"}
{"source": "GufdRdi1TElh", "target": "u0ANirFVu76B", "amount": 407.64, "currency": "EUR"}
^C
Processed a total of 4783 messages
```

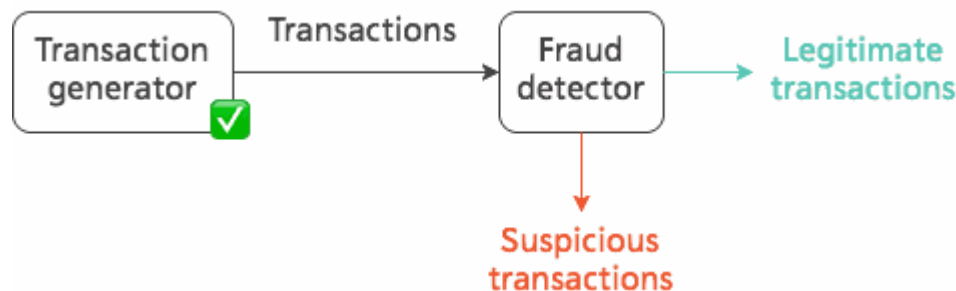
Wonderful! It works — fake transactions are effectively produced to the topic.

You can stop the generator for now: `ctrl+c` OR `$ docker-compose down`.

So, what have we got so far?

- A local Kafka cluster up and running 🚀
- A producer-based application that generates fake transactions and publishes them to a Kafka topic 📝

Here's the architecture diagram again:



*Transaction generator: check!*

We're just missing one thing — the actual **fraud detector** to process the stream of transactions and detect those that are suspicious!

## Building the fraud detector

The fraud detector is a typical example of a **stream processing application**.

It takes a stream of transactions as an input, performs some kind of filtering, then outputs the result into two separate streams — those that are legitimate, and those that are suspicious, an operation also known as **branching**.



*The detector branches out the transaction stream in two separate streams. ✂*

Beyond producers that publish messages into a topic, there exists another type of Kafka applications to read messages from a topic. They are called **consumers**. Actually, we've already used one via the `kafka-console-consumer` script above!

Consumers allow to read messages from one or more topic. In practice, they **subscribe to those topics** and Kafka will broadcast the messages to them as they are being published. This is a feature at the core of the **reactiveness** of streaming applications made with Kafka.

Again, we're lucky! The Kafka Python client allows us to build consumers in Python. We'll use [Kafka Python's Consumer API](#) for this.

The fraud detector will not be a plain consumer, though. It is a **streaming application**. As such, it uses a consumer to read messages, then does its own processing on those messages and produces messages back into one of the two output topics. So we'll need **a consumer and a producer**.

First, let's write the consumer part. Again, I'll go step by step and eventually show you the end result.

Let's begin by grabbing the `KafkaConsumer` :

```
# detector/app.py
from kafka import KafkaConsumer
```

Let's grab the broker URL and the transaction topic from environment variables:

```
import os

# ...
KAFKA_BROKER_URL = os.environ.get("KAFKA_BROKER_URL")
TRANSACTIONS_TOPIC = os.environ.get("TRANSACTIONS_TOPIC")
```

We can now instantiate the consumer. We know the transactions topic contains JSON-encoded values (as produced by the transactions generator), so we can use `KafkaConsumer`'s `value_deserializer` to automatically load the raw bytes into a Python dictionary. This is similar to `KafkaProducer`'s `value_serializer` we have used previously.

```
import json

# ...
consumer = KafkaConsumer(
    TRANSACTIONS_TOPIC,
    bootstrap_servers=KAFKA_BROKER_URL,
    value_deserializer=json.loads,
)
```

We can now process the stream of messages by iterating over the consumer:

```
for message in consumer:
    transaction: dict = message.value
    # TODO: determine if transaction is suspicious
    # TODO: output transaction in the correct topic (legit or fraud)
```

Now an interesting question — how do we determine whether a transaction is fraud?

There are a lot of options, and it depends a lot on the business use case.

In the real world, **detecting fraud is a difficult problem** because there are a lot of ways in which a transaction could indicate fraud. For example, if someone receives 50 transactions of exactly 1000€ each from 50 different accounts within 24 hours, that's probably suspicious. You could even think of adding an AI layer to further improve the predictions.

Although it is feasible, that kind of rule is harder to implement with Kafka than what we can cover here. All we're interested in here is how Kafka can be used to **bring the stream processing approach to life**.

To keep it simple, we'll have to cheat a bit. Imagine we deal with a fictional banking system in which it is illegal to send more than 900€ at a time. As a result, any transaction whose `amount` is greater than 900 can be considered as fraud.

Here is the corresponding function that will determine whether a transaction is suspicious:

```
def is_suspicious(transaction: dict) -> bool:
    return transaction["amount"] >= 900
```



Guess what — we now have a way to decide which topic to redirect the transaction to!

We're now ready to implement the meat of the detector. To produce the messages into the topics, we'll need a `KafkaProducer`. Let's instantiate it — just like we did for the transactions generator.

```
from kafka import KafkaProducer

# ...
producer = KafkaProducer(
    bootstrap_servers=KAFKA_BROKER_URL,
    value_serializer=lambda value: json.dumps(value).encode(),
)
```

We can now make use of `is_suspicious()` and the producer to determine which topic the transaction should go to, and then produce the transaction into that topic:

```
import os

# ...
LEGIT_TOPIC = os.environ.get("LEGIT_TOPIC")
FRAUD_TOPIC = os.environ.get("FRAUD_TOPIC")
# ...
for message in consumer:
    transaction: dict = message.value
    topic = FRAUD_TOPIC if is_suspicious(transaction) else LEGIT_TOPIC
    producer.send(topic, value=transaction)
```

That's it! We have plugged in all the parts together.

Before we try this out, here is the **final version of the fraud detector**:

```
# detector/app.py
import os
import json
from kafka import KafkaConsumer, KafkaProducer

KAFKA_BROKER_URL = os.environ.get("KAFKA_BROKER_URL")
TRANSACTIONS_TOPIC = os.environ.get("TRANSACTIONS_TOPIC")
LEGIT_TOPIC = os.environ.get("LEGIT_TOPIC")
FRAUD_TOPIC = os.environ.get("FRAUD_TOPIC")

def is_suspicious(transaction: dict) -> bool:
    """Determine whether a transaction is suspicious."""
    return transaction["amount"] >= 900

if __name__ == "__main__":
    consumer = KafkaConsumer(
        TRANSACTIONS_TOPIC,
        bootstrap_servers=KAFKA_BROKER_URL,
        value_deserializer=lambda value: json.loads(value),
    )
    producer = KafkaProducer(
        bootstrap_servers=KAFKA_BROKER_URL,
        value_serializer=lambda value: json.dumps(value).encode(),
    )
    for message in consumer:
        transaction: dict = message.value
```

```
topic = FRAUD_TOPIC if is_suspicious(transaction) else LEGIT_TOPIC
producer.send(topic, value=transaction)
print(topic, transaction) # DEBUG
```

And the corresponding service in our `docker-compose.yml` configuration:

```
# docker-compose.yml
services:
  # ... generator here ...
  detector:
    build: ./detector
    environment:
      KAFKA_BROKER_URL: broker:9092
      TRANSACTIONS_TOPIC: queueing.transactions
      LEGIT_TOPIC: streaming.transactions.legit
      FRAUD_TOPIC: streaming.transactions.fraud
```

## Trying out the fraud detector

Let's try things out! As before, let's start by launching Docker Compose, which will this time start both the transaction generator and the fraud detector:

```
$ docker-compose up
```

As we did when trying out the generator, we can verify that the detector correctly consumes, processes and produces the transactions by starting two `kafka-console-consumer` S — one for each output topic.

Here are the commands to do so:

```
docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bootstrap-server localhost:9092 --zookeeper-quorum=localhost:2181 --topic streaming.transactions.legit --from-beginning
docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bootstrap-server localhost:9092 --zookeeper-quorum=localhost:2181 --topic streaming.transactions.suspicious --from-beginning
```

Let's inspect the contents of these topics!

- The `streaming.transactions.legit` contains the legit transactions (amount below 900€), as expected:

```
...
{"source": "v16x080j7s4u", "target": "QgvC9nMnBQM0", "amount": 231.67, "currency": "EUR"}
{"source": "wqduJdB5focL", "target": "sxKNYd1SQqQ1", "amount": 229.57, "currency": "EUR"}
{"source": "VNWKUvsm00WA", "target": "7wWynS1nuYpP", "amount": 752.4, "currency": "EUR"}
{"source": "bjt8hci7q8Wm", "target": "LobJtJJB3RDt", "amount": 410.42, "currency": "EUR"}
{"source": "E6NpRQcSJ3es", "target": "E9QeU7qfekKB", "amount": 659.23, "currency": "EUR"}
{"source": "IijVKIGtiGrc", "target": "eWVPvLGm8JLi", "amount": 846.07, "currency": "EUR"}
{"source": "yGfZ1Xa6k1r0", "target": "N5RvY7R05sQF", "amount": 217.46, "currency": "EUR"}
...
^C
Processed a total of 13593 messages
```

- The `streaming.transactions.fraud` only contains suspicious transactions (amount above 900€), as expected too!

```
...
{"source": "jwRMZyzGAsmG", "target": "i33smnlpnYtd", "amount": 980.05, "currency": "EUR"}
{"source": "WcmydQNdDxya", "target": "ykFPmUSysJuJ", "amount": 914.81, "currency": "EUR"}
{"source": "UuC0tfIqc4oU", "target": "PrD271ASM3mZ", "amount": 911.9, "currency": "EUR"}
{"source": "ZHdwSxcIi1gv", "target": "CNLyxBAHT2lW", "amount": 925.55, "currency": "EUR"}
{"source": "gQUtarjghjWx", "target": "ABKEhcJuYiEo", "amount": 945.14, "currency": "EUR"}
{"source": "NS2HA0HoMQPt", "target": "h989MZ4lcDqg", "amount": 901.25, "currency": "EUR"}
{"source": "0w22LoewVfYW", "target": "bUejK0tbzLT2", "amount": 955.0, "currency": "EUR"}
...
^C
Processed a total of 1519 messages
```

As you can see, about 1 in 10 messages has been flagged as fraud. We'd expect 10% considering the transactions range between 0-1000€ with a trigger at 900€, so it all seems to work fine!

## Taking a step back

That's it — we now have a **working fraud detection pipeline**:

- A local Kafka cluster that acts as a **centralised streaming platform**.
- A **transaction generator** which produces transactions to the cluster.
- A **fraud detector** which processes transactions, detects potentially fraudulent ones and produces the results in two separate topics.

You might be wondering — **what's next?**

There are many ways in which this system could be improved or extended. Here are just a few ideas:

- Implement a better fraud detection algorithm
- Send alerts or reports about fraudulent transactions somewhere: email, notifications, a Slack bot — anything!

You may also be wondering — what about **deployment**? Well, I skipped this part intentionnally. While getting a set of Kafka applications in production should not be any different from your usual workflow, it may require its own best practices. On the other hand, putting a Kafka cluster in production is a wholly different topic (pun intended).

Although we've achieved to build a **streaming system**, we haven't even scratched the surface of what the **Kafka ecosystem** can do.

Those among you that are already familiar with the Kafka ecosystem may be thinking: "What about KSQL?", "What about Kafka Streams?", "What about the Schema Registry?". Well, yes, there's all that, and we might be able to peek into that later on.

What I wanted to show you in this tutorial was **the core of Apache Kafka** — topics, producers and consumers, and how one can arrange them to build real-world streaming apps.

## Streaming all the things

If you've read and built this fraud detection system with me — congrats! You have just built **your first real-world streaming application with Apache Kafka and Python**.

We can be proud of what we've achieved. We combined Apache Kafka, Python and Docker to solve an actual business problem.

This was just an introductory example, yet I hope you begin to envision how **a streaming platform like Apache Kafka enables a whole new world of systems and applications**. Just think of how hard it would have been to build this system with REST APIs and good old HTTP calls. My guess is: impossible.

If you're excited to see what's next, I have good news: we're just getting started! 🙌

In the next article, we'll get back to Kafka core ideas. We'll try to perform a **paradigm shift**. Apache Kafka is all about **streaming data**, so we'll need to shift from thinking in requests to **thinking in events**.

Until then, you can find the complete code on [GitHub](#), so keep playing around with the fraud detector and stay tuned for more Kafka material! 📖

4 Comments - powered by utteranc.es

kapilbk1996 commented on Jul 25, 2020

Great article with real world example. Thanks a lot.

yonash2 commented on Jul 27, 2020

Hi,

Really cool article, thanks a lot! 🙏

One small error I found:

As we did when trying out the generator, we can verify that the detector correctly consumes, processes and produces the transactions by starting two kafka-console-consumers — one for each output topic.

Here are the commands to do so:

```
docker-compose -f docker-compose.kafka.yml exec detector kafka-console-consumer --  
docker-compose -f docker-compose.kafka.yml exec detector kafka-console-consumer --
```

It should be executed on the broker:

```
docker-compose -f docker-compose.kafka.yml exec broker ...
```

```
docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bc  
docker-compose -f docker-compose.kafka.yml exec broker kafka-console-consumer --bc
```

florimondmanca commented on Jul 27, 2020

Owner

Thanks @yonash2! Good catch. ~~I don't have access to a workstation right now, but there's an "Edit" link at the top of the page in case you'd be up for submitting a PR. :-)~~ Edit: nevermind, [fixed](#)! Thanks.

Nada-Bu commented on Mar 4, 2021

Thanks for this detailed article.



1




1



Write

Preview

Sign in to comment

 Styling with Markdown is supported

Sign in with GitHub

## Links

[GitHub](#) [Twitter](#) [LinkedIn](#) [DEV](#)

© 2021 Florimond Manca · [Source code](#)