How to use Apache Kafka to transform a batch pipeline into a real-time one

Stéphane Maarek Follow
Oct 24, 2017 · 19 min read

In this blog, I will thoroughly explain how to build an end-to-end real-time data pipeline by building four micro-services on top of Apache Kafka. It will give you insights into the Kafka Producer API, Avro and the Confluent Schema Registry, the Kafka Streams High-Level DSL, and Kafka Connect Sinks.

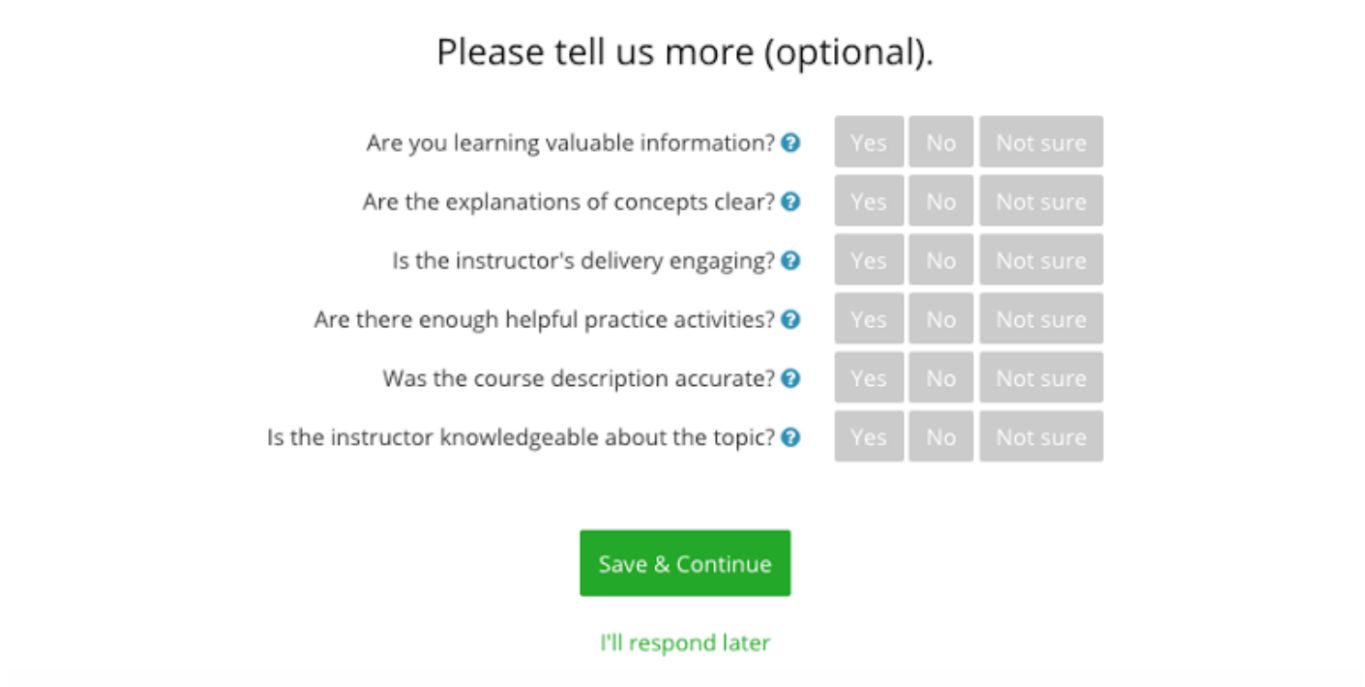
. . .

> If you need to cross the street, would you do it with information that is five minutes old?

. . .

The challenge we'll solve

Aside from my regular job as a data streaming consultant, I am an online instructor on the Udemy online course marketplace. I teach about the technologies that I love, such as Apache Kafka for Beginners, Kafka Connect, Kafka Streams, Kafka Setup & Administration, Confluent Schema Registry & REST Proxy, Apache Kafka Security, and Kafka Monitoring & Operations, Confluent KSQL.

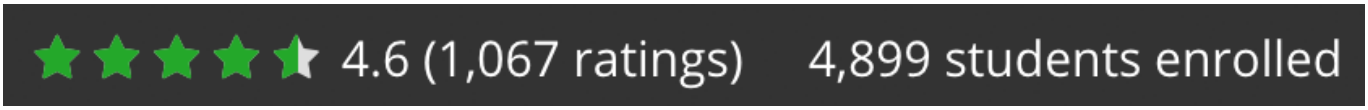


A review prompt on Udemy

On Udemy, students have the opportunity to post reviews on the courses they take in order to provide some feedback to the instructor and the other platform's users.

But these reviews are released to the public every… **24 hours**! I know this because every day at 9 AM PST I receive a **batch** of new reviews.

It can take another **few hours** for a course page to be updated with the new review count and average rating. Sounds like a daily scheduled batch job is running somewhere!

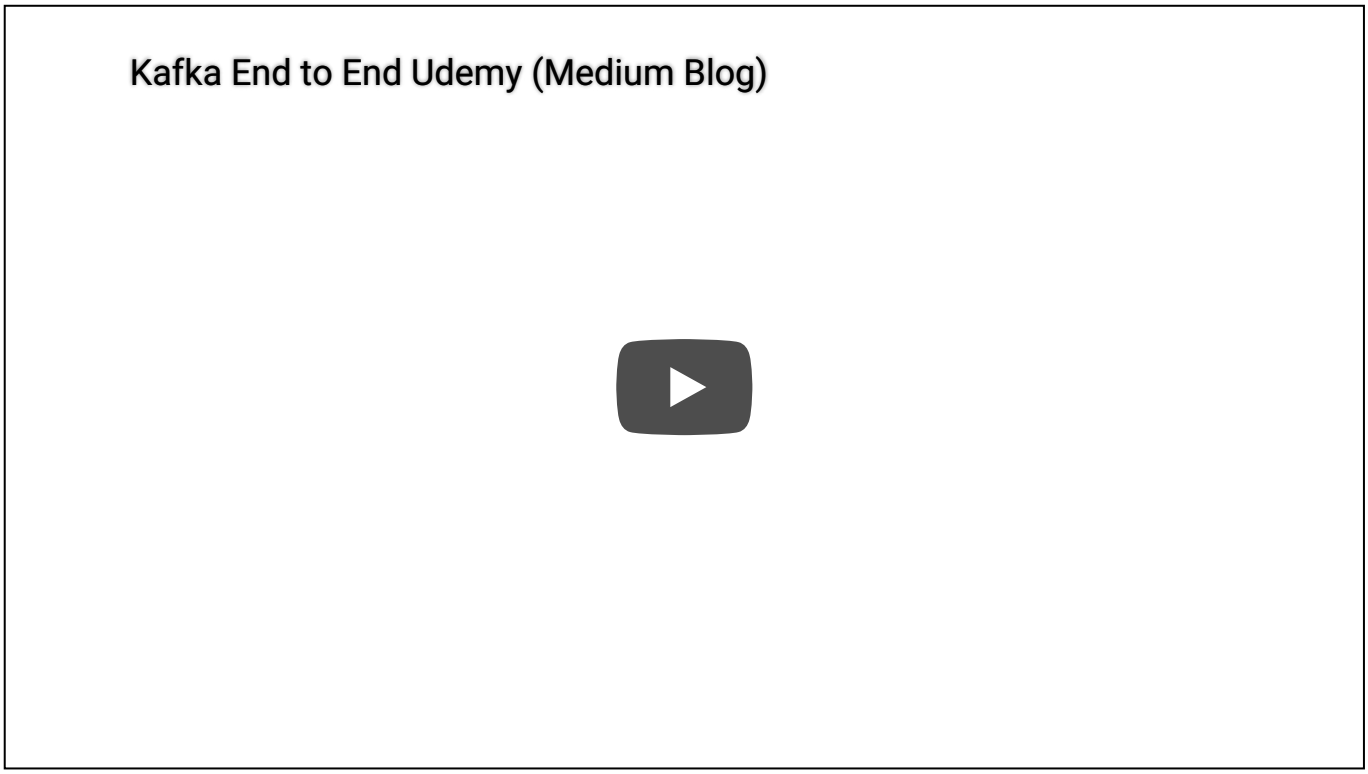


Screenshot of the statistics for the Apache Kafka for Beginners course

In this blog, I'll show you how to transform this batch pipeline into a real-time one using Apache Kafka by building a few micro-services.

All the source code is available here: https://github.com/simplesteph/medium-blog-kafka-udemy

And for the lazy, you can see me running all the code in this video:



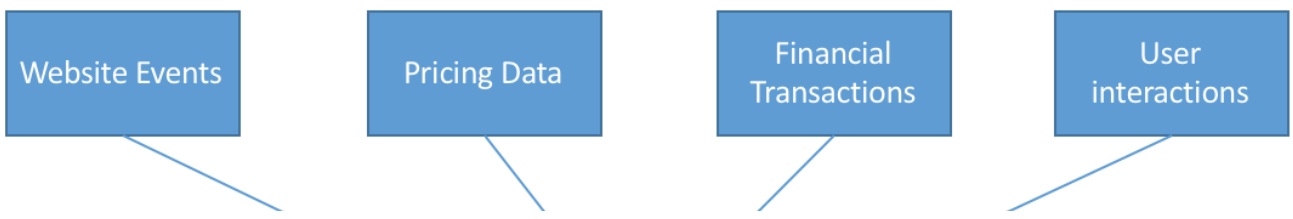
Video to see the code running (for the lazy)

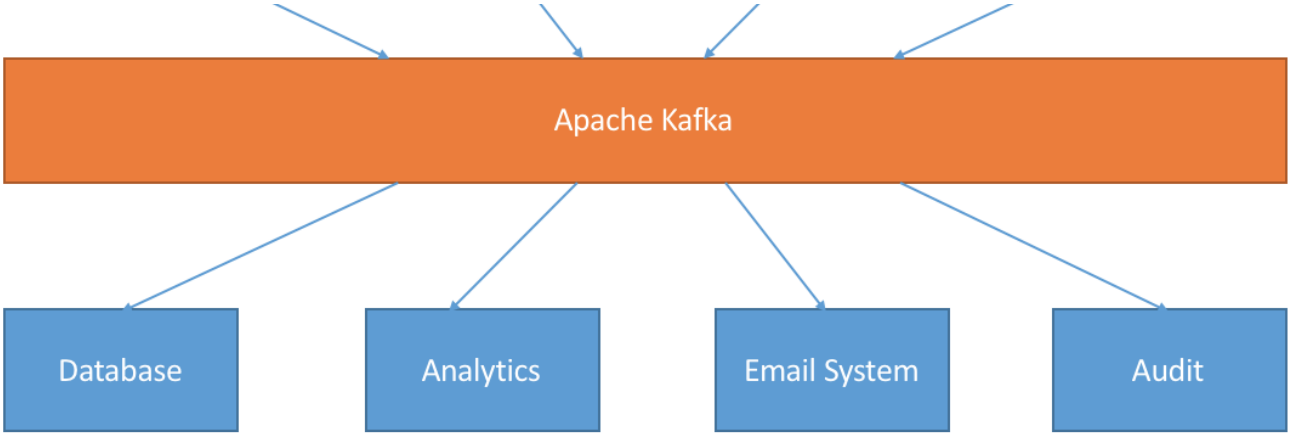
Excited? Let's get started!

. . .

What is Apache Kafka?

Apache Kafka is a distributed streaming platform. At its core, it allows systems that generate data (called Producers) to persist their data in real-time in an Apache Kafka Topic. Any topic can then be read by any number of systems who need that data in real-time (called Consumers). Therefore, at its core, Kafka is a Pub/Sub system. Behind the scenes, Kafka is distributed, scales well, replicates data across brokers (servers), can survive broker downtime, and much more.





A typical representation of a Pub/Sub system

Apache Kafka originated at LinkedIn and was open-sourced later to become an Apache top-level project. It is now being leveraged by some big companies, such as Uber, Airbnb, Netflix, Yahoo, Udemy, and more than 35% of the Fortune 500 companies.

This blog is *somewhat* advanced, and if you want to understand Kafka better before reading any further, check out Apache Kafka for Beginners.

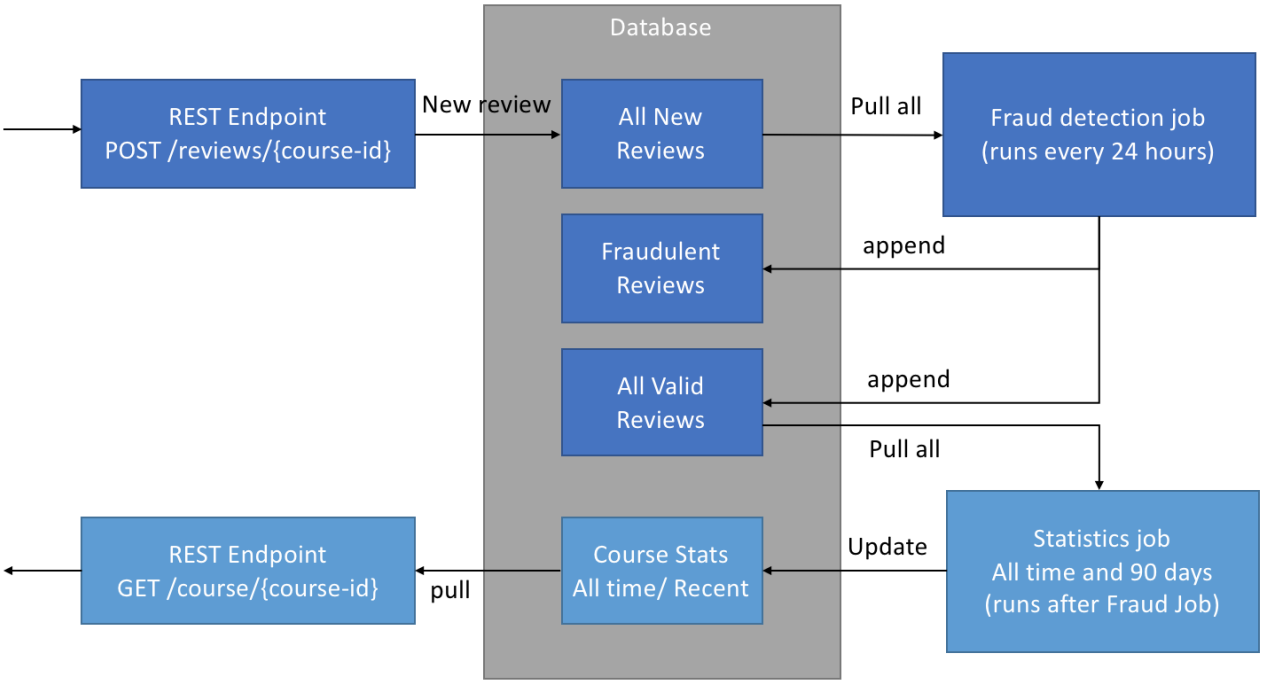
. . .

The reviews processing batch pipeline

Before jumping straight in, it's very important to map out the current process and see how we can improve each component. Below are my personal assumptions:

- When a user writes a review, it gets POSTed to a Web Service (REST Endpoint), which will store that review into some kind of database table
- Every 24 hours, a batch job (could be Spark) would take all the new reviews and apply a spam filter to filter fraudulent reviews from legitimate ones.
- New valid reviews are published to another database table (which contains all the historic valid reviews).
- Another batch job or a SQL query computes new stats for courses. Stats include all-time average rating, all-time count of reviews, 90 days average rating, and 90 days count of reviews.

- The website displays these metrics through a REST API when the user navigates a website.

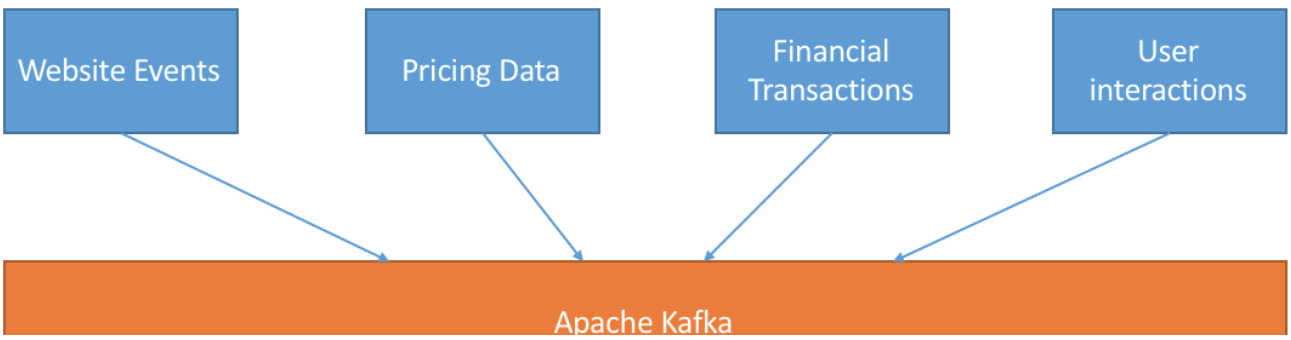


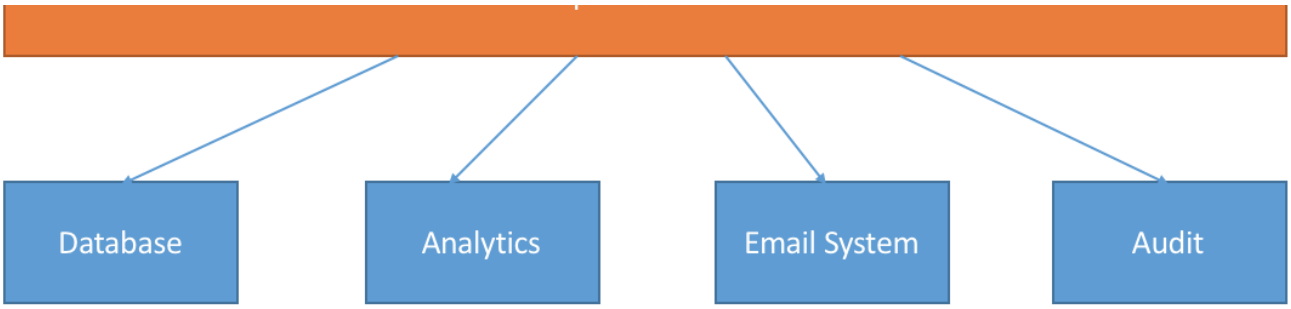
Personal assumptions of current pipeline. Looks familiar?

Let's see how we can transform that batch pipeline into a scalable, real-time and distributed pipeline with Apache Kafka.

The target architecture

When building a real-time pipeline, you need to think **micro-services**. Micro-services are small components designed to do one task very well. They interact with one another, **but not directly**. Instead, they interact indirectly by using an intermediary, in our case a Kafka topic. Therefore, **the contract between two micro-services is the data itself.** That contract is enforced by leveraging schemas (more on that later)





Gentle reminder

The contract between two micro-services is the data itself

To summarise, our only job is to model the data, because **data is king**.

Note all of the micro-services in this blog are just **normal Java applications**, lightweight, portable, and you can easily put them in Docker containers (that's a stark contrast from say… Spark). Here are the micro-services we are going to need:

1. **Review Kafka Producer:** when a user posts a review to a REST Endpoint, it should end up in Kafka right away.
2. **Fraud Detector Kafka Streams:** we're going to get a stream of reviews. We need to be able to score these reviews for fraud using some real-time machine learning, and either validate them or flag them as a fraud.
3. **Reviews Aggregator Kafka Streams:** now that we have a stream of valid reviews, we should aggregate them either since a course launch, or only taking into account the last 90 days of reviews.
4. **Review Kafka Connect Sink:** We now have a stream of updates for our course statistics. We need to sink them in a PostgreSQL database so that other web services can pick them up and show them to the users and instructors.

Target Architecture for our Real-Time Pipeline. Every color is a micro service

Now we get a clear view of our end-to-end real-time pipeline, and it looks like we have a lot of work ahead. Let's get started!

. . .

1) Reviews Kafka Producer

To get the reviews data, I will use the external REST API Udemy provides to fetch a list of existing and published reviews for a course.

The Producer API helps you produce data to Apache Kafka. It will take an object combined with a `Serializer` (a class that allows you to transform your objects in raw bytes) and send it across.

So here, we have two steps to implement:

1. Create a way to fetch reviews for any course using the Udemy REST API
2. Model these reviews into a nice Avro Object and send that across to Kafka.

A typical Kafka Producer

You can find the source code for the producer here.

Fetching Udemy Reviews

Getting reviews is actually easy, you can learn about the REST API here. We're just going to figure out how many reviews a course has in total, and then repeatedly call the REST API from the last page to the first. We add the reviews to a java queue.

```
while (keepOnRunning){
    List<Review> reviews = udemyRESTClient.getNextReviews();
    log.info("Fetched " + reviews.size() + " reviews");
    if (reviews.size() == 0){
       keepOnRunning = false;
    } else {
       for (Review review : reviews){
           reviewsQueue.put(review);
       }
    }
    Thread.sleep(50);
}
```

FetchReviews.java hosted with ❤️ by GitHub view raw

The queue has a fixed size of 100 so it's blocking until not full

Sending the Reviews to Kafka

Sending the reviews to Kafka is just as easy as creating and configuring a Kafka Producer:

```
public KafkaProducer<Long, Review> createKafkaProducer(AppConfig appConfig) {
    Properties properties = new Properties();
    properties.put("bootstrap.servers", "localhost:9092");
    properties.put("acks", "all");
    properties.put("retries", Integer.MAX_VALUE);
    properties.put("max.in.flight.requests.per.connection", 1);
    properties.put("key.serializer", LongSerializer.class.getName());
    properties.put("value.serializer", KafkaAvroSerializer.class.getName());
    properties.put("schema.registry.url", "http://localhost:8081");
    return new KafkaProducer<>(properties);
```

```
11  }
```

CreateProducer.java hosted with ❤️ by GitHub view raw

Typical Kafka Producer properties

And then producing data with it:

```
1  while (udemyRESTClientRunning()){
2      Review review = reviewsQueue.poll();
3      if (review == null) {
4          Thread.sleep(200);
5      } else {
6          reviewCount += 1;
7          log.info("Sending review " + reviewCount + ": " + review);
8          kafkaProducer.send(new ProducerRecord<>("udemy-reviews", review));
9      }
10 }
```

ProducerReviews.java hosted with ❤️ by GitHub view raw

Easy, right? Tie that with a couple of threads, some configuration, parsing JSON documents to create an Avro object, shutdown hooks and you got yourself a rock-solid producer!

Avro and the Schema Registry

> *Hey! (you may say). What's your `Review` object?*

Good question. If you've been paying close attention to the configuration of the Kafka Producer, you can see that the `"value.serializer"` is of type `KafkaAvroSerializer` . There's a lot to learn about Avro, but I'll try to make it short for you.

With Avro, you define Schemas. These Schemas define the fields of your data, alongside their types, and their optionality. To picture an Avro object, think of a JSON document, although your schema strictly dictates how the Avro object can be formed. As a bonus, once your Avro is formed (like a POJO), it can be easily serialized as an array of bytes, which is exactly what Kafka likes. Any other programming language can read the Avro bytes, and deserialize them to an object-specific to that programming language.

This Avro schema is defined for our `Review` :

```
{"namespace": "com.github.simplesteph.avro.udemy",
  "type": "record",
  "name": "Review",
  "fields": [
    {"name": "id", "type": "long", "doc": "Review ID as per Udemy's db" },
    {"name": "title", "type": ["null", "string"], "default": null },
    {"name": "content", "type": ["null", "string"], "default": null, "doc": "Review text
    {"name": "rating", "type": "string", "doc": "review value"},
    {"name": "created",  "type": { "type" : "long", "logicalType" : "timestamp-millis" }
    {"name": "modified",  "type": { "type" : "long", "logicalType" : "timestamp-millis"
    {"name": "user", "type": "com.github.simplesteph.avro.udemy.User"},
    {"name": "course", "type": "com.github.simplesteph.avro.udemy.Course"}
  ]
}
```

udemy-review.avsc hosted with ❤ by GitHub view raw

An extract of the Review Avro Schema

Any other programming language can read the Avro bytes, and deserialize them.

> *Hey! (may you say). What's the role of the schema registry then?*

The Confluent Schema Registry has an awesome role in your data pipeline. Upon sending some data to Kafka, your `KafkaAvroSerializer` will separate the schema from the data in your Avro object. It will send the Avro schema to the schema registry, and the actual content bytes (including a reference to the schema) to Kafka. Why? Because the result is that the payload sent to Kafka is much lighter, as the schema wasn't sent. That optimization is a great way to speed up your pipeline to achieve extreme volumes.

How the Schema Registry works.

There's also another use for the Schema Registry, in order to enforce backward and forward compatible schema evolution, but that's out of scope for that already super long blog post.

In summary, You Really Need One Schema Registry.

If you want to learn about Avro and the Schema Registry, see my course here!

Running the producer

All the instructions to run the project are in GitHub, but here is the output you will see. After downloading and installing the Confluent Platform 3.3.0, and running `confluent start`, you should have a fully-featured Kafka cluster!

Starting Kafka using the Confluent Distribution

First, we create a topic:

```
$ kafka-topics --create --topic udemy-reviews --zookeeper
localhost:2181 --partitions 3 --replication-factor 1
```

Then we run the producer from the command line:

```
$ git clone https://github.com/simplesteph/medium-blog-kafka-udemy
$ mvn clean package
$ export COURSE_ID=1075642  # Kafka for Beginners Course
$ java -jar udemy-reviews-producer/target/uber-udemy-reviews-
producer-1.0-SNAPSHOT.jar
```

and observe the log:

```
[2017-10-19 22:59:59,535] INFO Sending review 7: {"id": 5952458,
"title": "Fabulous on content and concepts", "content": "Fabulous on
content and concepts", "rating": "5.0", "created": 1489516276000,
"modified": 1489516276000, "user": {"id": 2548770, "title": "Punit
G", "name": "Punit", "display_name": "Punit G"}, "course": {"id":
1075642, "title": "Apache Kafka Series - Learn Apache Kafka for
Beginners", "url": "/apache-kafka-series-kafka-from-beginner-to-
intermediate/"}} (ReviewsAvroProducerThread)
```

If we fire up a Kafka Avro Console Consumer:

```
$ kafka-avro-console-consumer --topic udemy-reviews --bootstrap-
server localhost:9092 --from-beginning
{"id":5952458,"title":{"string":"Fabulous on content and
concepts"},"content":{"string":"Fabulous on content and
concepts"},"rating":"5.0","created":1489516276000,"modified":1489516
276000,"user":{"id":2548770,"title":"Punit
G","name":"Punit","display_name":"Punit G"},"course":
{"id":1075642,"title":"Apache Kafka Series - Learn Apache Kafka for
Beginners","url":"/apache-kafka-series-kafka-from-beginner-to-
intermediate/"}}
```

Excellent, we now have a real-time stream of reviews landing in a Kafka topic! **Step 1: done.**

If you're interested in learning all of the Kafka fundamentals, check out my Kafka for Beginners Udemy Course. That's 4 hours of content to get you up to speed before you read further down!

. . .

Still here? Perfect. It's about to get really fun!

2) Fraud Detector Kafka Streams

At this stage, we have simulated a stream of reviews in Kafka. Now we can plug in another service that will read that stream of reviews and apply a filter against a dummy machine learning model to figure out if a review is or isn't spam.

Our Fraud Detection Micro-Service

For this, we will use Kafka Streams. The Kafka Streams API is made for real-time applications and micro-services that get data from Kafka and end up in Kafka. It has recently gained exactly-once capability when running against a cluster that is version ≥ 0.11.

Kafka Streams applications are fantastic because, in the end, they're "just" Java application. No need to run them on a separate cluster (like Spark does on YARN), it just runs standalone the way you know and like, and can be scaled by just running some more instances of the same application. To learn more about Kafka Streams you can check out my Kafka Streams Udemy course.

Kafka Streams application topology

A Kafka Streams application is defined through a topology (a sequence of actions) and to define one we will use the simple High-Level DSL. People familiar with Spark or Scala can relate to some of the syntaxes, as it leverages a more functional paradigm.

The app itself is dead simple. We get our config, create our topology, start it, and add a shutdown hook:

```
private void start() {
    Properties config = getKafkaStreamsConfig();
    KafkaStreams streams = createTopology(config);
    streams.cleanUp();
    streams.start();
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
```

FraudAppOverall.java hosted with ❤️ by GitHub view raw

Pretty much every Kafka Stream app

The topology can be written as:

```
KStreamBuilder builder = new KStreamBuilder();

KStream<Bytes, Review> udemyReviews = builder.stream("udemy-reviews");
KStream<Bytes, Review>[] branches = udemyReviews.branch(
        (k, review) -> isValidReview(review),
        (k, review) -> true
);

KStream<Bytes, Review> validReviews = branches[0];
KStream<Bytes, Review> fraudReviews = branches[1];

validReviews.to("udemy-reviews-valid");
fraudReviews.to("udemy-reviews-fraud");
```

```
14
15  return new KafkaStreams(builder, config);
```

FraudAppTopology.java hosted with ❤️ by GitHub view raw

A very simple Kafka Streams topology

Fraud detection algorithm

Currently, my algorithm deterministically classifies a review as a fraud based on a hash value and assigns 5% of the reviews as Spam. Behind this oversimplified process, one can definitely apply any machine learning library to test the review against a pre-computed model. That model can come from Spark, Flink, H2O, anything.

The simplistic example:

```
1  private boolean isValidReview(Review review) {
2      try {
3          int hash = Utils.toPositive(Utils.murmur2(review.toByteBuffer().array()));
4          return  (hash % 100) >= 5; // 95 % of the reviews will be valid reviews
5      } catch (IOException e) {
6          return false;
7      }
8  }
```

FraudDetectionAlgorithm.java hosted with ❤️ by GitHub view raw

If you're interested in running more complex machine learning models with Kafka Streams, it's 100% possible: check out these articles.

Running the fraud streams application

Running the application is easy, you just start it like any other java application. We just first ensure the target topics are properly created:

```
$ kafka-topics --create --topic udemy-reviews-valid --partitions 3 --replication-factor 1 --zookeeper localhost:2181
$ kafka-topics --create --topic udemy-reviews-fraud --partitions 3 --replication-factor 1 --zookeeper localhost:2181
```

And then to run:

```
(from the root directory)
$ mvn clean package
$ java -jar udemy-reviews-fraud/target/uber-udemy-reviews-fraud-1.0-SNAPSHOT.jar
```

At this stage, we have a valid reviews topic that contains 95% of the reviews, and 5% in another fraud topic. Think about all of the possible applications! One could improve the model with all the fraud reviews, run manual checks, create reports, etc. **Step 2: done.**

Learning Kafka Streams

To learn about Kafka Streams, you can check out my Kafka Streams Udemy course.

. . .

It's about to get more difficult. We now want to compute statistics such as average rating or number of reviews, overall the reviews or just the most recent ones in a window of 90 days. Thanks for reading down here!

It's about to get real. I'm making sure I have your attention for the rest of the blog

3) Reviews Aggregator Kafka Streams

Target Architecture

Our third application is also a Kafka Streams application. It's a stateful one so the state will be transparently stored in Kafka. From an external eye, it looks like the following:

Architecture for our stateful Kafka Streams Application

KStream and KTables

In the previous section, we learned about the early concepts of Kafka Streams, to take a stream and split it in two based on a spam evaluation function. Now, we need to perform some stateful computations such as aggregations, windowing in order to compute statistics on our stream of reviews.

Thankfully we can use some pre-defined operators in the High-Level DSL that will transform a `KStream` into a `KTable`. A KTable is basically a table, that gets new events every time a new element arrives in the upstream KStream. The KTable then has some level of logic to update itself. Any KTable updates can then be forwarded downstream. For a quick overview of KStream and KTable, I recommend the quickstart on the Kafka website.

Aggregation Key

In Kafka Streams, aggregations are always-key based, and our current stream messages have a `null` key. We want to aggregate over each course therefore we first have to re-key our stream (by course-id). Re-keying our stream in Kafka Stream is very easy if you look at the code:

```
KStream<String, Review> validReviews = builder.stream("udemy-reviews-valid")
        .selectKey(((key, review) -> review.getCourse().getId()));
```

StreamRekey.java hosted with ❤ by GitHub view raw

Looks easy, but there's a catch!

But you need to be aware of something. When you re-key a KStream, and chain that with some stateful aggregations (and we will), the Kafka Streams library will write the re-keyed stream back to Kafka and then read it again. That network round trip has to do with data distribution, parallelism, state storage, recovery, and it could be an expensive operation. So be efficient when you change the key of your stream!

Statistics since course inception

Now, we profit! We have the guarantee that all the reviews belonging to one course will always go to the same Kafka Streams application instance. As our topic holds all the reviews since inception, we just need to create a `KTable` out of our stream and sink that somewhere.

```
// we build a long term topology (since inception)
KTable<String, CourseStatistic> longTermCourseStats =
        validReviews.groupByKey().aggregate(
                this::emptyStats,
                this::reviewAggregator,
                courseStatisticSpecificAvroSerde
        );
longTermCourseStats.toStream().to("long-term-stats");
```

LongTermStatsAggreation.java hosted with ❤ by GitHub view raw

A simple Kafka Streams aggregation. Note the .groupByKey() call

Good things to note:

- You need to define what the `emptyStats()` look like (course statistics with 0 reviews) — see the source code for such implementation
- You need to define how your stats change after a new review comes in (that's your aggregator)
- Each new review is seen as new data, not an update. The KTable has no recollection of past reviews. If you wanted to compute the statistics on updates as well, one could change the event format to capture "old" and "new" review state within one message.
- You should make sure your source topic does not expire data! It's a topic config. For this, you could either enable log compaction or set `retention.ms` to something like 100 years. As Jay Kreps (creator of Kafka, CEO of Confluent) wrote, it's okay to store data in Kafka.

Statistics for the past 90 days

Here come the fun and funky part. When we are dealing with data streaming, most of the time a business application will only require to analyze events over a time window. Some use cases include:

- Am I under DDOS? (sudden peak of data)
- Is a user spamming my forums? (high number of messages over short period for a specific user-id)
- How many users were active in the past hour?
- How much financial risk does my company have at right now?

For us, this will be:

> What is each course statistics over the past 90 days?

Let's note the aggregation computation is exactly the same. The only thing that changes over time is the data set we apply that aggregation onto. We want it to be recent (from the past 90 days), over a time window, and making sure that window advances every

day. In Kafka Streams, it's called a Hopping Window. You define how big the window is, and the size of the hop. Finally, to handle late arriving data, you define how long you're willing to keep a window for:

```
// A hopping time window with a size of 91 days and an advance interval of 1 day.
// the windows are aligned with epoch
long windowSizeMs = TimeUnit.DAYS.toMillis(91);
long advanceMs = TimeUnit.DAYS.toMillis(1);
TimeWindows timeWindows = TimeWindows.of(windowSizeMs).advanceBy(advanceMs);
```

HoppingWindow.java hosted with ❤ by GitHub view raw

Defining a hopping window of 90 days

Please note that this will generate about 90 different windows at any time. We will only be interested in the first one.

We filter only for recent reviews (really helps speed up catching up with the stream), and we compute the course statistics over *each time window*:

```
KTable<Windowed<String>, CourseStatistic> windowedCourseStatisticKTable = validReviews
        .filter((k, review) -> !isReviewExpired(review))
        .groupByKey().aggregate(
                this::emptyStats,
                this::reviewAggregator,
                timeWindows,
                courseStatisticSpecificAvroSerde
        );
```

TimeWindowAggregation.java hosted with ❤ by GitHub view raw

The aggregation is similar as before, but we now have an additional parameter.

That operation can become a bit costly as we keep 90 time windows for each course, and only care about one specific window (the last one). Unfortunately we cannot perform aggregations on sliding windows (yet), but hopefully, that feature will appear soon! It is still good enough for our needs.

In the meantime, we need to filter to only get the window we're interested in: it's the window which ends after today and ends before tomorrow:

```
KStream<String, CourseStatistic> recentStats = windowedCourseStatisticKTable
        .toStream()
        // we keep the current window only
        .filter((window, courseStat) -> keepCurrentWindow(window))
        .selectKey((k, v) -> k.key()); // the course id

recentStats.to("recent-stats");
```

FilterWindows.java hosted with ❤️ by GitHub view raw

see the source code for keepCurrentWindow(window)

And that's it, we get a topic fully updated in real-time with the most recent stats for our course.

Running the app

Running the application is easy, you just start it like any other java application. We just first ensure the target topics are properly created:

```
$ kafka-topics --create --topic long-term-stats --partitions 3 --
replication-factor 1 --zookeeper localhost:2181
$ kafka-topics --create --topic recent-stats --partitions 3 --
replication-factor 1 --zookeeper localhost:2181
```

And then to run:

```
(from the root directory)
$ mvn clean package
$ java -jar udemy-reviews-aggregator/target/uber-udemy-reviews-
aggregator-1.0-SNAPSHOT.jar
```

Feel free to fire up a few Avro consumers to see the results:

```
$ kafka-avro-console-consumer --topic recent-stats --bootstrap-
server localhost:9092 --from-beginning
$ kafka-avro-console-consumer --topic long-term-stats --bootstrap-
server localhost:9092 --from-beginning
```

Results may include a stream of:

```
{“course_id”:1294188,”count_reviews”:51,”average_rating”:4.539}
{“course_id”:1294188,”count_reviews”:52,”average_rating”:4.528}
{“course_id”:1294188,”count_reviews”:53,”average_rating”:4.5}
```

We have now two topics that get a stream of updates for long-term and recent stats, which is pretty cool. By the way, this topic is a very good candidate for long compaction. We only really care about the last value for each course. **Step 3: done.**

Notes

Although the Kafka Streams syntax looks quite simple and natural to understand, a lot happened behind the scenes. Here are a few things to note:

- **Exactly Once:** as we want that aggregation to be perfectly accurate, we need to enable exactly-once processing semantics (EOS). This feature appeared in 0.11, and the name stirred up a lot of debate. So, to make it short and clear, it means “effectively once”, and is exactly what we need (pun intended). That means no reviews will somehow be counted twice in case of broker, network, or application failure. Neat!
- **Incoming data format:** as mentioned before, it’ll be awesome if the data had a “new” and an “old” field. This would allow to handle updates in reviews and compute the correct average in case of updates to a review
- **Windowed aggregations:** there is a massive performance hit to computing 90 windows only to discard them all and keep the last one. I have evaluated it and found it to be 25 times less efficient than using the (way more advanced) lower level API
- **Lower Level API:** using this API, you can create your own transformers and compute exactly what you need. In the source code, you can find how to do the recent statistics computations using that API, although I won’t discuss it in this post as it goes way beyond the already immense quantity of information I just threw at you.

- **Performance:** these apps can be parallelized to the number of partitions in the incoming topic. It has horizontal scaling natively which is quite awesome. Kafka Streams in that regards makes it really easy to scale without maintaining some sort of back-end cluster.

. . .

…One last component!

4) Kafka Connect Sink — Exposing that data back to the users

Eventually, all we care about is people browsing the Udemy website and visualizing the course statistics. As with most web services, serving information is often backed by some sort of database. For my example, I have chosen a relational database (PostgreSQL), but one could choose a NoSQL one like MongoDB, or a search index such as ElasticSearch. Possibilities are endless, and there exists Kafka Connect Sinks for pretty much any technology out there.

The Kafka Connect Pipeline

Kafka Connect

Kafka Connect is a framework upon which developers can create connectors. These connectors can be of two kinds: Source and Sink. Source are producers, Sink are consumers. The beautiful thing behind Kafka Connect is that it provides you infrastructure to run any connector. For an end user, running a connector is as easy as pushing configuration. Re-using other people's work sounds like a dream, right? Well, that's what Kafka Connect is about.

To learn about Kafka Connect in details, check out my Kafka Connect course

The JDBC Sink Connector

Here's the good news: I'm not going to show you any more Java code. We're not going to re-invent the wheel to put our topic data into a PostgreSQL table. Instead, we're going to leverage a well written and battled tested Kafka connector by just pushing a bit of configuration.

We are using the excellent Kafka Connect JDBC Sink by Confluent. The configuration itself is dead simple:

```
name=SinkTopics
connector.class=io.confluent.connect.jdbc.JdbcSinkConnector
tasks.max=3
connection.url=jdbc:postgresql://localhost:5432/postgres
connection.user=postgres
connection.password=postgres
insert.mode=upsert
pk.mode=record_value
pk.fields=course_id
auto.create=true
topics=recent-stats,long-term-stats
key.converter=org.apache.kafka.connect.storage.StringConverter
```

SinkTopicsInPostgres.properties hosted with ❤️ by GitHub view raw

Our configuration. Not hard to come up with when you read the docs

Things to note:

- `tasks.max=3` : that's the level of parallelism of your connector. That means we will spin at most three tasks to read the input topics. You can increase that number to scale up, up to the number of partitions you're reading from. That's because any Kafka Connect Sink is behind the scene just a Kafka Consumer
- `key.converter` : I chose to have my topics keyed by course-id exposed as a String. The default converter provided to the connect workers being Avro, it would throw an error if I didn't override the key.converter. Hence, we use the simplistic `StringConverter` here.

- You could deploy many connectors (more than one configuration) to a Kafka Connect Cluster. Benefits? Well, we could sink our topics in 10 different databases, 10 different technologies, to serve different purposes and applications in your organization, all from the same connect cluster. We could also extend the list of topics to sink so that some data scientists can perform some cool analysis on your fraud algorithm effectiveness for example.

Some of the results we get in our PostgreSQL database

. . .

Last but not least — Final notes

As you have seen, the possibilities in Kafka are endless. The ecosystem is extremely large and there are tons of patterns and cool concepts to learn. The takeaways I want you to have today are the following:

- **Event sourcing in Kafka is awesome**. Getting a stream of every event that has happened in your company ever could be a dream come true.
- **Kafka is an excellent candidate as a backbone for your micro-services**. Break down some complex flows into easy ones, and make each micro-service perform its core capability at its best. If the fraud application improves, there would be no disruption to your other systems!
- **Use the Confluent Schema registry.** Data is your first class citizen in Apache Kafka, and schemas make everyone's life so much simpler. Documentation is embedded, parsing errors are virtually inexistent. You can even make your schema evolve over time, as long as you ensure they're forward and backward compatible.
- **Leverage the right tools for each job.** As you've seen, there was a mix of Producer, Streams, and Connect. I made the maximum effort in order not to re-invent the

wheel. Take your time to evaluate solutions before diving right into your favorite technology.

- **Never stop learning.** I have been using Kafka for over a year now, and I keep on learning every day. I also want to share my experience, so check out Apache Kafka for Beginners, Kafka Connect, Kafka Streams, Kafka Setup & Administration, Confluent Schema Registry & REST Proxy, Apache Kafka Security, and Kafka Monitoring & Operations, Confluent KSQL
- **What this blog did not cover** (and the range of stuff there's yet to learn or write about): Kafka Consumers API, Kafka Admin Client, Kafka Streams Lower Level API, Kafka Streams joins to enrich data, Kafka Connect Source, Kafka Security, Kafka Monitoring, Kafka Setup and Administration, Kafka REST Proxy, KSQL, Zookeeper (and I might have forgotten other things). *The ecosystem is huge*
- **KSQL is the future**: Most if not all of the written Kafka Streams applications in this blog can be replaced by only a few KSQL statements as soon as it has official Avro support. It will open up stream processing to a much wider audience and enable the rapid migration of many batch SQL applications to Kafka. *I plan on publishing a subsequent blog when I migrate the code to KSQL. Stay tuned!*

Kafka is a fantastic piece of technology. I am convinced it will make all organizations thrive in flexibility and reactiveness. There is a ton to learn about Kafka, and I sincerely hope that through this blog, I have clearly exposed how to chain micro-services in order to transform a batch pipeline into a real-time one.

Clap, share, comment, give me feedback. I'd love to hear your thoughts! Thanks to Michael, Gwen, Cam, Octav and Eric for proofreading and providing improvements :)

Happy learning!

Big Data Kafka Udemy Data Tutorial