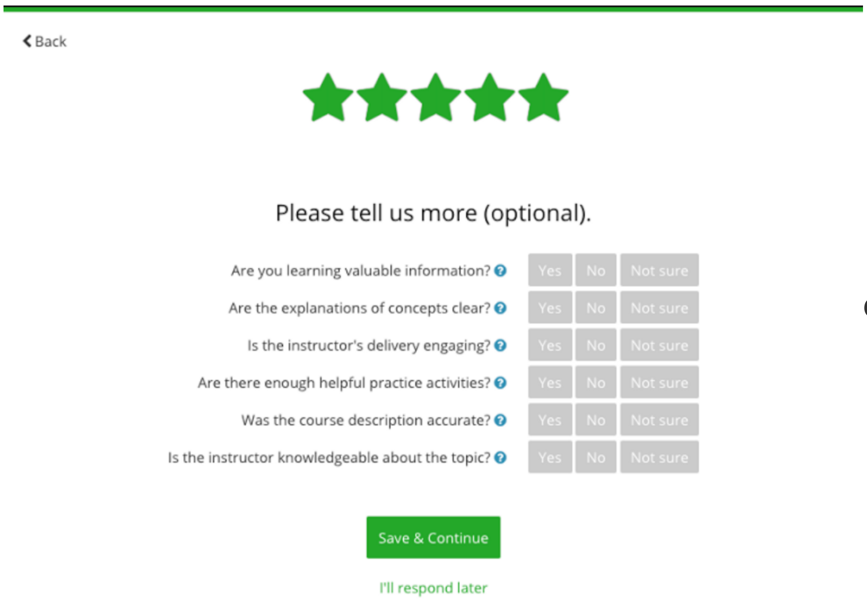
Use Apache Kafka to Transform a Batch Pipeline Into a Real-Time One, Part 1

In this series, I will thoroughly explain how to build an end-to-end real-time data pipeline by building four microservices on top of Apache Kafka.

In this blog, I will thoroughly explain how to build an end-to-end real-time data pipeline by building four microservices 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.

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 and Apache Kafka Security.



Are you learning valuable information?	Yes	No	Not sure
Are the explanations of concepts clear?	Yes	No	Not sure
Is the instructor's delivery engaging?	Yes	No	Not sure
Are there enough helpful practice activities?	Yes	No	Not sure
Was the course description accurate?	Yes	No	Not sure
Is the instructor knowledgeable about the topic?	Yes	No	Not sure

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!

These are the statistics being computed for Udemy Courses

★★★★½ 4.5 (3,061 ratings) 13,259 students enrolled

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.

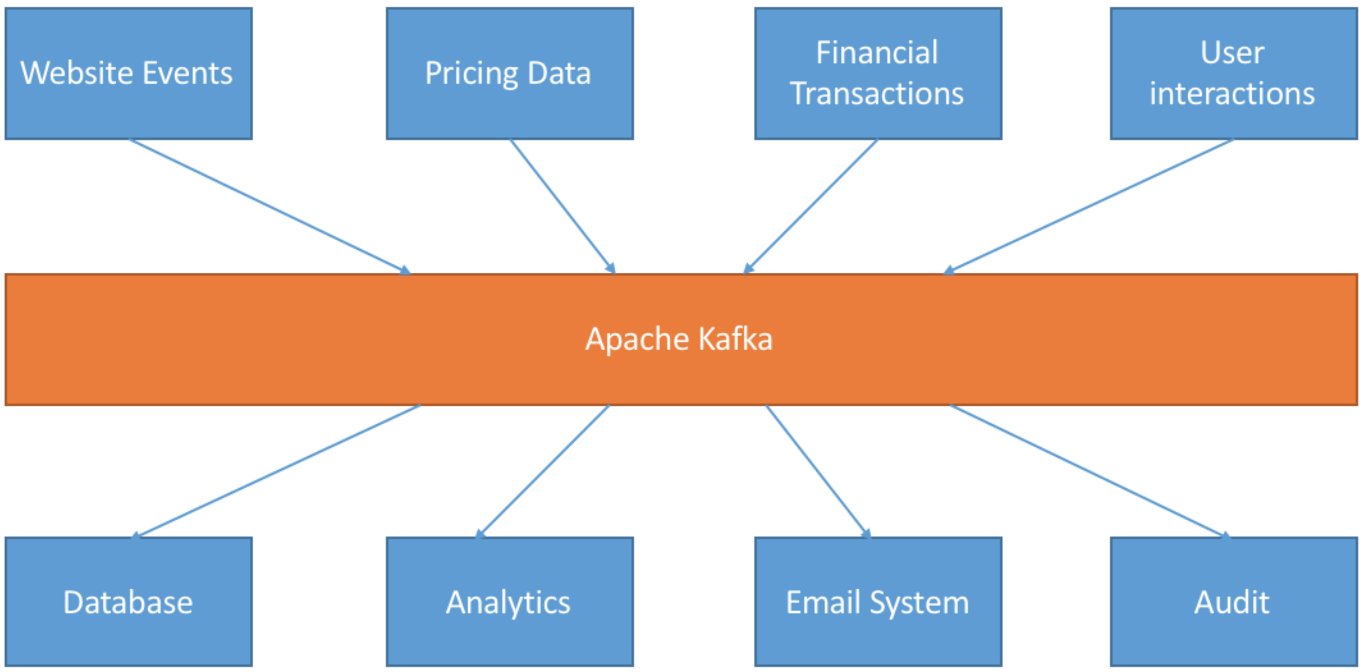
All the source code is available here: https://github.com/simplesteph/medium-blog-kafka-udemy

Also, you can see me running all the code in this video: https://youtu.be/h5i94umfzMM

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.



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.

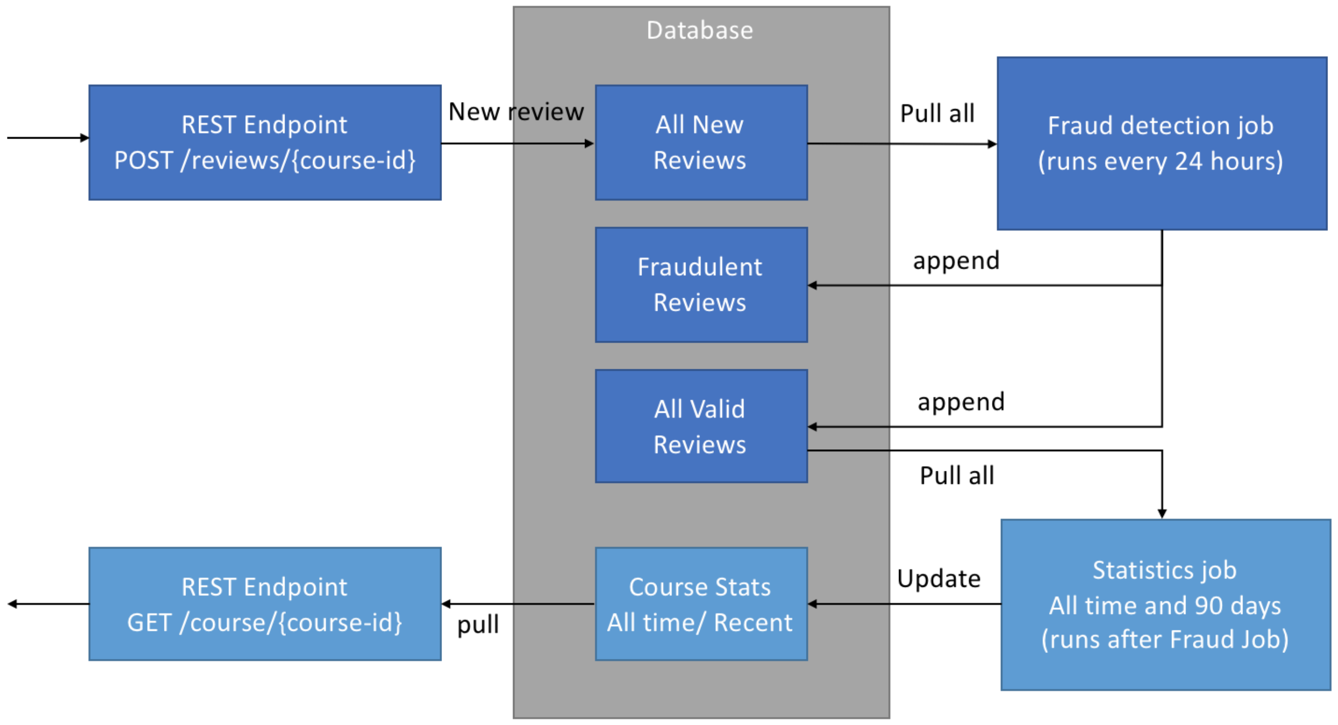
This blog is *somewhat* advanced, and if you want to understand Kafka better before reading any further, check out Apache Kafka for Beginners.

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.


Database
REST Endpoint POST /reviews/{course-id}
New review
All New Reviews
Pull all
Fraud detection job (runs every 24 hours)
Fraudulent Reviews
append
All Valid Reviews
append
Pull all
Statistics job All time and 90 days (runs after Fraud Job)
Update
Course Stats All time/ Recent
pull
REST Endpoint GET /course/{course-id}

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

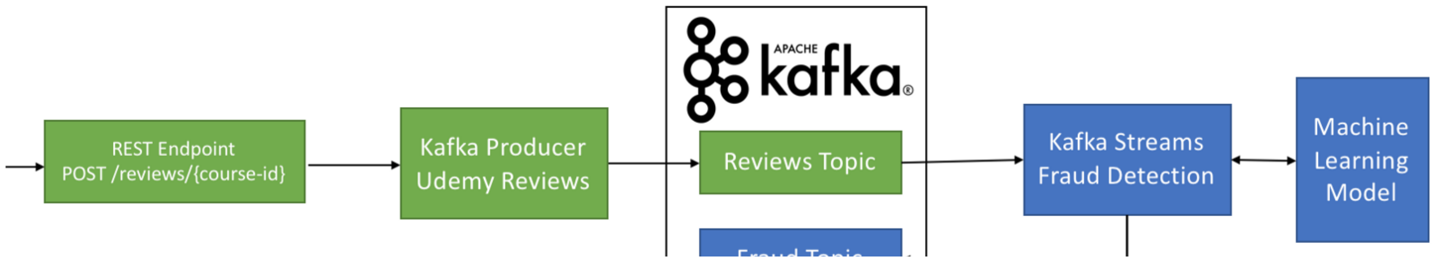
The Target Architecture

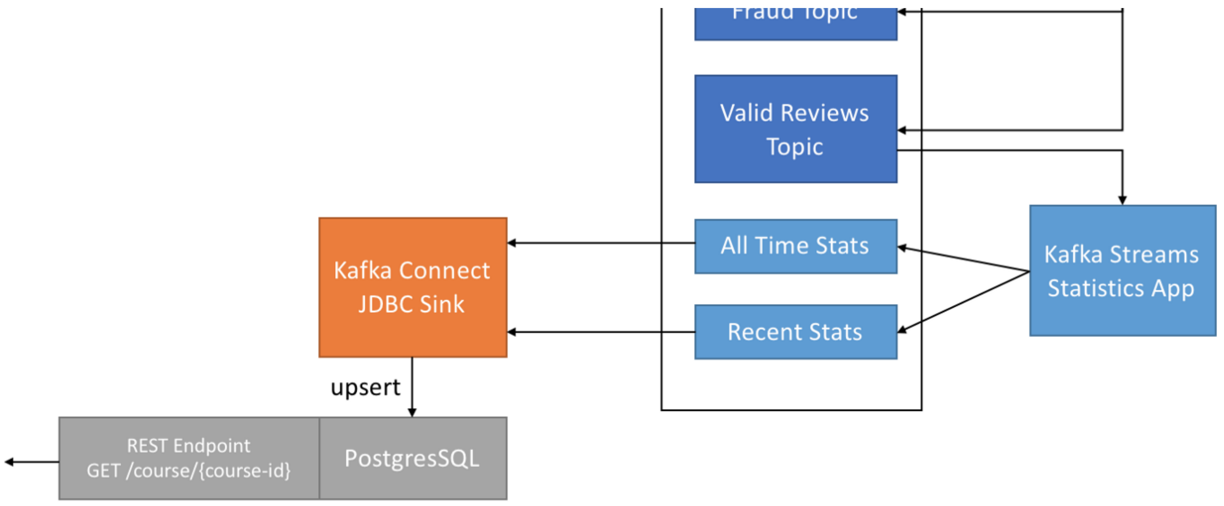
When building a real-time pipeline, you need to think **microservices**. Microservices are small components designed to do one task very well. They interact with one another, **but not directly**. Instead, they interact indirectly by using an intermediary, in our case a Kafka topic. Therefore, **the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

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

Note all of the microservices in this post 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 microservices 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.


REST Endpoint POST /reviews/{course-id}
Kafka Producer Udemy Reviews
APACHE kafka
Reviews Topic
Kafka Streams Fraud Detection
Machine Learning Model



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!

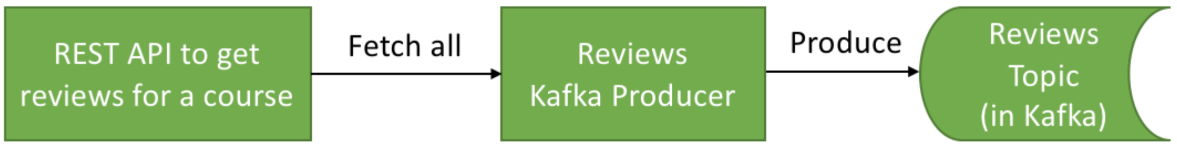
1) Reviews Kafka Producer

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.



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);
}
```

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);
}
```

And then producing data with it:

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

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.

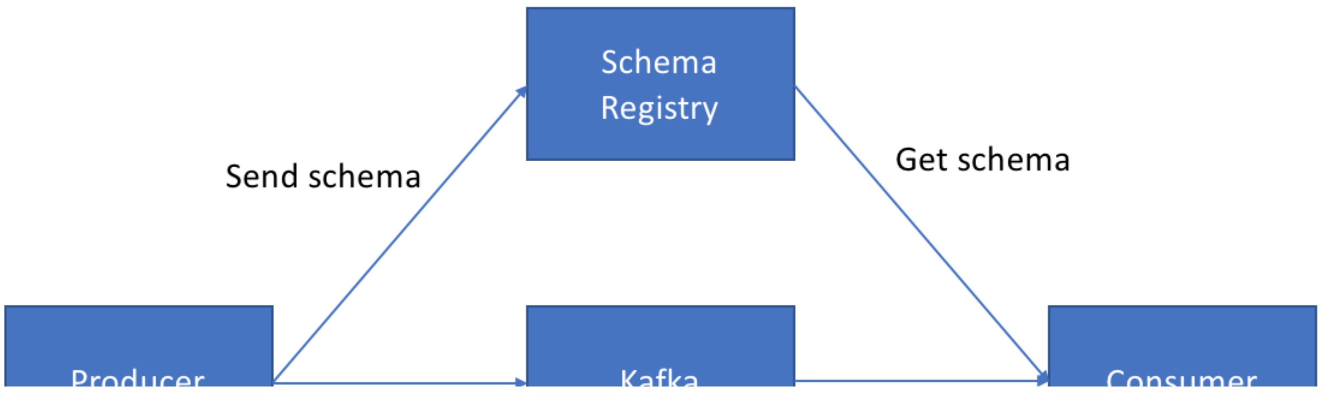
With Avro, you define Schemas. These Schemas define the fields of your data, alongside their types, and their optionality. To picture an Avro object, think of a JSON document, although your schema strictly dictates how the Avro object can be formed. As a bonus, once your Avro is formed (like a POJO), 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 if pr
    {"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"}
  ]
}
```

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.



Send Avro content
Read Avro content

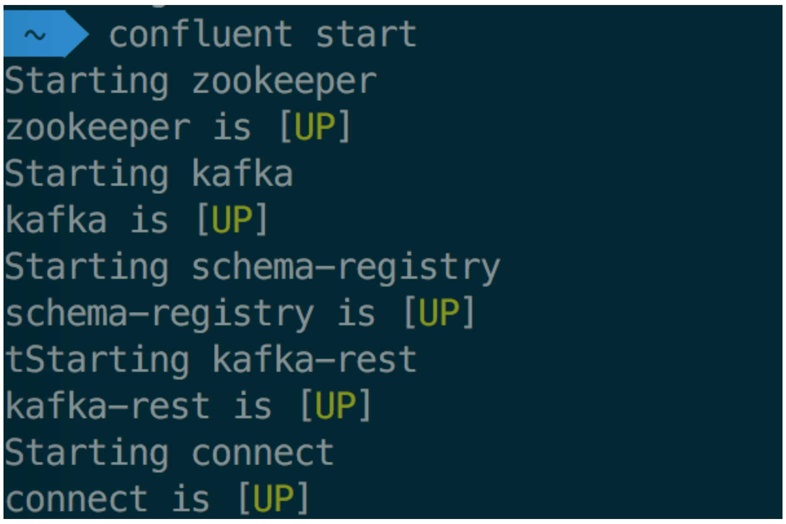
There's also another use for the Schema Registry, in order to enforce backward and forward compatible schema evolution, but that's out of scope for that already super long blog post.

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 on 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!



```
~ confluent start
Starting zookeeper
zookeeper is [UP]
Starting kafka
kafka is [UP]
Starting schema-registry
schema-registry is [UP]
tStarting kafka-rest
kafka-rest is [UP]
Starting connect
connect is [UP]
```

First, we create a topic:

```
$ kafka-topics --create --topic udemy-reviews --zookeeper localhost:2181 --partitions 3 --repl
```

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
```

If we fire up a Kafka Avro Console Consumer:

```
$ kafka-avro-console-consumer --topic udemy-reviews --bootstrap-server localhost:9092 --from-b
{"id":5952458,"title":{"string":"Fabulous on content and concepts"},"content":{"string":"Fabul
```

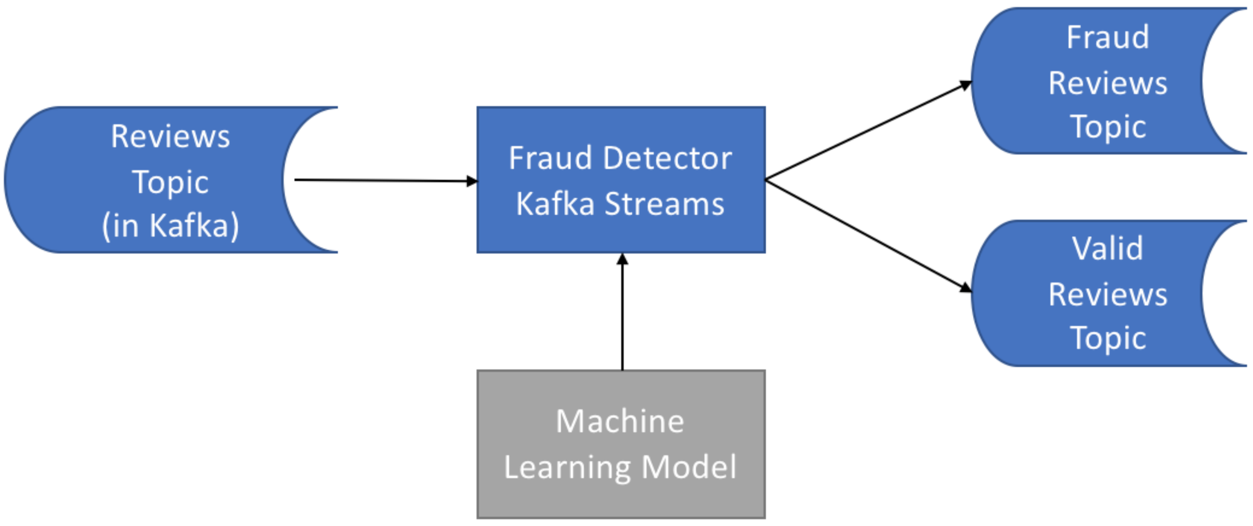
Excellent, we now have a real-time stream of reviews landing in a Kafka topic! **Step 1: done.**

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.



For this, we will use Kafka Streams. The Kafka Streams API is made for real-time applications and microservices 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 applications. 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));
}
```

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");

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

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:

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

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 --zo
$ kafka-topics --create --topic udemy-reviews-fraud --partitions 3 --replication-factor 1 --zo
```

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 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, over all the reviews or just the most recent ones in a window of 90 days. Thanks for reading down to here!

That's it for today! Tune in next time when we'll cover aggregators in Kafka streams and exposing data back to users.

Hortonworks Community Connection (HCC) is an online collaboration destination for developers, DevOps, customers and partners to get answers to questions, collaborate on technical articles and share code examples from GitHub. Join the discussion.