Kafka Streams Demo Application

In this tutorial we will run Confluent's Kafka Music demo application for the Kafka Streams API. If you are looking for a similar demo application written with KSQL queries, check out the separate page on the KSQL music demo walk-thru

The Kafka Music application demonstrates how to build a simple music charts application that continuously computes, in real-time, the latest charts such as Top 5 songs per music genre. It exposes its latest processing results -- the latest charts -- via Kafka's Interactive Queries feature and a REST API. The application's input data is in Avro format and comes from two sources: a stream of play events (think: "song X was played") and a stream of song metadata ("song X was written by artist Y"); see inspecting the input data in the Appendix for how the input data looks like.

More specifically, we will run the following services:

- Confluent's Kafka Music demo application
- a single-node Kafka cluster with a single-node ZooKeeper ensemble
- Confluent Schema Registry

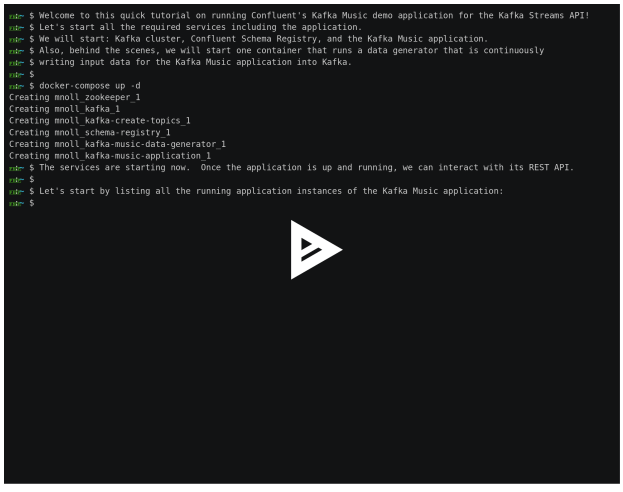
Requirements

This tutorial uses Docker Compose.

- You must install Docker and Docker Compose.
- The Confluent Docker images require Docker version 1.11 or greater.

Running the Kafka Music demo application

If you want to see an appetizer of what we will do in this section, take a look at the following screencast:



Screencast: Running Confluent's Kafka Music demo application (3 mins)

Ready now? Let's start!

Clone the Confluent Docker Images repository:

Expand Content

```
# Clone the git repository
$ git clone https://github.com/confluentinc/kafka-streams-examples.git

# Change into the directory for this tutorial
$ cd kafka-streams-examples/

# Switch to the `5.3.0-post` branch
$ git checkout 5.3.0-post
```

Now we can launch the Kafka Music demo application including the services it depends on such as Kafka.

```
$ docker-compose up -d
```

After a few seconds the application and the services are up and running. One of the started containers is continuously generating input data for the application by writing into its input topics. This allows us to look at live, real-time data when playing around with the Kafka Music application.

Now we can use our web browser or a CLI tool such as `curl` to interactively query the latest processing results of the Kafka Music application by accessing its REST API.

REST API example 1: list all running application instances of the Kafka Music application:

```
$ curl -sXGET http://localhost:7070/kafka-music/instances

# You should see output similar to following, though here
# the output is pretty-printed so that it's easier to read:
[
  {
    "host": "localhost",
    "port": 7070,
    "storeNames": [
      "all-songs",
      "song-play-count",
      "top-five-songs",
      "top-five-songs-by-genre"
    ]
  }
]
```

REST API example 2: get the latest Top 5 songs across all music genres:

```
$ curl -sXGET http://localhost:7070/kafka-music/charts/top-five

# You should see output similar to following, though here
# the output is pretty-printed so that it's easier to read:
[
  {
    "artist": "Jello Biafra And The Guantanamo School Of Medicine",
    "album": "The Audacity Of Hype",
    "name": "Three Strikes",
    "plays": 70
  },
  {
    "artist": "Hilltop Hoods",
    "album": "The Calling",
    "name": "The Calling",
    "plays": 67
  },

  ... rest omitted...
]
```

The REST API exposed by theKafka Music application supports further operations. See the top-level instructions in its source code for details.

Once you're done playing around you can stop all the services and containers with:

```
$ docker-compose down
```

We hope you enjoyed this tutorial!

Running further Confluent demo applications for the Kafka Streams API

The container named `kafka-music-application`, which runs the Kafka Music demo application, actually contains all of Confluent's Kafka Streams demo applications. The demo applications are packaged in the fat jar at `/usr/share/java/kafka-streams-examples/kafka-streams-examples-|release|-standalone.jar` inside this container. This means you can easily run any of these applications from inside the container via a command similar to:

```
# Example: Launch the WordCount demo application (inside the `kafka-music-application` container)
$ docker-compose exec kafka-music-application \
        java -cp /usr/share/java/kafka-streams-examples/kafka-streams-examples-5.3.0-standalone.jar \
        io.confluent.examples.streams.WordCountLambdaExample \
        kafka:29092
```

Of course you can also modify the tutorial's `docker-compose.yml` for repeatable deployments.

Note that you must follow the full instructions of each demo application (see its respective source code at https://github.com/confluentinc/examples). These instructions include, for example, the creation of the application's input and output topics. Also, each demo application supports CLI arguments. Typically, the first CLI argument is the `bootstrap.servers` parameter and the second argument, if any, is the `schema.registry.url` setting.

Available endpoints **from within the containers** as well as **on your host machine**:

Endpoint	Parameter	Value (from within containers)	Value (from your host machine)
Kafka Cluster	`bootstrap.servers`	`kafka:29092`	`localhost:9092`
Confluent Schema Registry	`schema.registry.url`	`http://schema-registry:8081`	`http://localhost:8081`
ZooKeeper ensemble	`zookeeper.connect`	`zookeeper:32181`	`localhost:32181`

The ZooKeeper endpoint is not required by Kafka Streams applications, but you need it to e.g. manually create new Kafka topics or to list available Kafka topics.

Appendix

Inspecting the input topics of the Kafka Music application

Inspect the "play-events" input topic, which contains messages in Avro format:

```
# Use the kafka-avro-console-consumer to read the "play-events" topic
$ docker-compose exec schema-registry \
    kafka-avro-console-consumer \
        --bootstrap-server kafka:29092 \
        --topic play-events --from-beginning

# You should see output similar to:
{"song_id":11,"duration":60000}
{"song_id":10,"duration":60000}
{"song_id":12,"duration":60000}
{"song_id":2,"duration":60000}
{"song_id":1,"duration":60000}
```

Inspect the "song-feed" input topic, which contains messages in Avro format:

```
# Use the kafka-avro-console-consumer to read the "song-feed" topic
$ docker-compose exec schema-registry \
    kafka-avro-console-consumer \
        --bootstrap-server kafka:29092 \
        --topic song-feed --from-beginning

# You should see output similar to:
{"id":1,"album":"Fresh Fruit For Rotting Vegetables","artist":"Dead Kennedys","name":"Chemical Warfare","genre":"Punk"
}
{"id":2,"album":"We Are the League","artist":"Anti-Nowhere League","name":"Animal","genre":"Punk"}
{"id":3,"album":"Live In A Dive","artist":"Subhumans","name":"All Gone Dead","genre":"Punk"}
{"id":4,"album":"PSI","artist":"Wheres The Pope?","name":"Fear Of God","genre":"Punk"}
```

Creating new topics

You can create topics manually with the `kafka-topics` CLI tool, which is available on the `kafka` container.

```
 # Create a new topic named "my-new-topic", using the `kafka` container
 $ docker-compose exec kafka kafka-topics \
     --zookeeper zookeeper:32181 \
     --create --topic my-new-topic --partitions 2 --replication-factor 1

# You should see a line similar to:
Created topic "my-new-topic".
```

Listing available topics

You can list all available topics with the `kafka-topics` CLI tool, which is available on the `kafka` container.

```
# List available topics, using the `kafka` container
$ docker-compose exec kafka kafka-topics \
    --zookeeper zookeeper:32181 \
    --list
```

Additional topic information is displayed by running `--describe` instead of `-list`.

Please report any inaccuracies on this page or suggest an edit.

6 Votes