Strata 2017 Kafka Tutorial

Hands-On Exercise Manual

Table of Contents

Introduction

The Hands-On Exercises in this tutorial use a pre-configured Virtual Machine (VM) running Linux, which you should already have downloaded. We have installed Kafka (from the Confluent Platform distribution) on the VM. You will automatically be logged in to the machine when it starts up.

Login

If you need the login credentials, they are:

```
Username: training
Password: training
```

The `training` user has passwordless sudo enabled.

Command Line Examples

The Exercises contain commands that must be run from the command line. These commands will look like this:

```
$ pwd
/home/training

$ cat /etc/hosts
127.0.0.1   localhost kafka-training broker1 zookeeper1 kafkarest1 schemaregistry1
```

Commands you should type are shown in bold; non-bold text is an example of the output produced as a result of the command.

Machine Aliases

The VM is a single node. To make it easier to understand what you are connecting to when writing code, we have created various aliases in the `/etc/hosts` file. The hostnames `broker1`, `zookeeper1`, etc. simply resolve back to `127.0.0.1`.

Hands-On Exercise: Verifying that Kafka is Running Correctly

It is important to verify that Kafka is functioning correctly before you start to use it. You may need to restart a daemon process during the tutorial, if it has terminated for some reason.

1. Open the Terminal Emulator on the desktop.
2. Get the listing of Java processes:

```
$ sudo jps
2183 Jps
1752 SupportedKafka
1832 SchemaRegistryMain
1775 KafkaRestMain
1487 QuorumPeerMain
```

This will display a list of all Java processes running on the VM. The four Kafka-related processes you should see, and their Linux service names, are:

Java Process Name	Service Name
`QuorumPeerMain`	`zookeeper`
`SupportedKafka`	`kafka-server`
`KafkaRestMain`	`kafka-rest`
`SchemaRegistryMain`	`schema-registry`

3. If any of the four Java processes are not present, start the relevant service(s) by typing:

```
$ sudo service servicename start
```

Hands-On Exercise: Pulling Data From a Database With Kafka Connect

In this Hands-On Exercise, you will run Kafka Connect in standalone mode with a JDBC Source Connector. The Connector will read data from a SQL database and write it to a Kafka topic. You will use a command-line Kafka Consumer to view the contents of the topic.

For simplicity, we will be using SQLite, a public domain SQL database engine. However, the same techniques can be used for any RDBMS for which you have a JDBC driver.

We will store the data in a Kafka topic in Avro format. For that, we'll use the Confluent Schema Registry, which simplifies managing Avro data in Kafka.

1. Make sure you are in the `/home/training/tutorial` directory.

```
$ cd /home/training/tutorial
```

1. View the `/etc/schema-registry/connect-avro-standalone.properties` file. This file configures the standalone worker.

```
$ cat /etc/schema-registry/connect-avro-standalone.properties
```

2. Create a file called `connector-source.properties` in the `/home/training/tutorial` directory. This file will configure the JDBC source connector. Use whatever editor you are most familiar with to create the file. `vi` and `emacs` are installed on the VM; if you would prefer to use a graphical editor, use `Geany`, which can be found from the `Applications>Programming` menu. The file should contain the following lines:

```
name=tutorial-sqlite-jdbc-autoincrement
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
tasks.max=1
connection.url=jdbc:sqlite:test.db
mode=incrementing
incrementing.column.name=id
topic.prefix=test-sqlite-jdbc-
```

3. Create a SQLite database. Note that you should be in the `/home/training/tutorial` directory, since SQLite will create a file with the same name as the database you specify within the current working directory.

```
$ cd /home/training/tutorial
$ sqlite3 test.db
SQLite version 3.6.20
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

4. Now create a table, and populate it with a few sample rows of data.

```
sqlite> CREATE TABLE accounts(id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name
VARCHAR(255));
sqlite> INSERT INTO accounts(name) VALUES('alice');
sqlite> INSERT INTO accounts(name) VALUES('bob');
```

5. Open another terminal window, and launch Kafka Connect. We'll run it in standalone mode, since we are working on a single machine.

```
$ cd /home/training/tutorial
$ connect-standalone /etc/schema-registry/connect-avro-standalone.properties \
connector-source.properties
```

Leave the standalone connector running in the foreground.

6. Open a third terminal window, and run the console consumer to ensure the lines were written to Kafka:

```
$ kafka-avro-console-consumer \
--new-consumer \
--bootstrap-server broker101:9092 \
--topic test-sqlite-jdbc-accounts \
--from-beginning
{"id":1,"name":{"string":"alice"}}
{"id":2,"name":{"string":"bob"}}
```

As you can see, Connect wrote the two rows to the `test-sqlite-jdbc-accounts topic`.

7. Go back to the terminal window in which you are running SQLite, and add another row to the table.

```
sqlite> INSERT INTO accounts(name) VALUES('charlie');
```

8. Switch back to the console consumer window. You will see that the new row has been added to the topic, and displayed by the console consumer.

9. Switch back to the Kafka Connect window, and terminate the process by hitting `Ctrl-c`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Writing Data to a Database with Kafka Connect

In this Hands-On Exercise, you will write data from a Kafka topic to a table in a SQL database. As before, we will use SQLlite; the data in the topic will be in Avro format.

1. Make sure you are in the `/home/training/tutorial` directory.

```
$ cd /home/training/tutorial
```

1. Create a file called `connector-sink.properties` in the `/home/training/tutorial` directory. This file will configure the JDBC sink connector. The file should contain the following lines:

```
name=tutorial-sink
connector.class=io.confluent.connect.jdbc.JdbcSinkConnector
tasks.max=1
topics=orders
connection.url=jdbc:sqlite:test.db
auto.create=true
```

2. Open another terminal window, and launch Kafka Connect. We'll run it in standalone mode, since we are working on a single machine.

```
$ cd /home/training/tutorial
$ connect-standalone /etc/schema-registry/connect-avro-standalone.properties \
connector-sink.properties
```

Leave the standalone connector running in the foreground.

3. Open a new terminal window, and run the Avro console producer.

```
$ sudo kafka-avro-console-producer \
 --broker-list broker101:9092 --topic orders \
 --property
value.schema='{"type":"record","name":"myrecord","fields":[{"name":"id","type":"int"},{"name
":"product","type":"string"},{"name":"quantity","type":"int"},{"name":"price","type":
"float"}]}'
```

You will see some 'SLF4J' warning lines. Ignore these. When the producer has started, it waits for you to enter lines of data. Enter the following:

```
{"id": 123, "product": "widget", "quantity": 100, "price": 12.99}
```

Note: Do *not* enter a blank line. This will cause the Producer to terminate, since it cannot parse the blank line as valid JSON. If this happens, just re-run the Producer.

4. Now, query SQLite to see that the new table `orders` has been created, and the record has been written to it.

```
$ sqlite3 test.db
sqlite> select * from orders;
widget|12.99|100|123
```

5. Write another record to the topic, and verify that it, too, is written to the `orders` table.

6. When you have finished, switch back to the Kafka Connect window, and terminate the process by hitting `Ctrl-c`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Running a Kafka Streams Application

In this Hands-On Exercise, you will run a Kafka Streams application which will process a topic containing lines of text.

1. First, create and populate a topic with lines of text. We will use the contents of the `/var/log/dmesg` file for this exercise.

```
$ cat /var/log/dmesg | kafka-console-producer \
--broker-list broker101:9092 \
--topic dmesg
```

Note: You will see warnings about `Error while fetching metadata`. Dont worry about this; the Producer is looking for the topic and, when it sees that it is not present, Kafka automatically creates it.

2. Verify that the data has been written by running the console consumer.

```
$ kafka-console-consumer \
--new-consumer \
--bootstrap-server broker101:9092 \
--topic dmesg \
--from-beginning
```

The Consumer will continue to run, waiting to display any more data that is added to the topic, so hit `Ctrl-c` to terminate it.

3. Create the Eclipse project using Maven, so that you can import it into Eclipse.

```
$ cd /home/training/kafka2017/streams
$ mvn eclipse:eclipse
```

4. Launch Eclipse by double-clicking the Eclipse icon on the desktop.
5. Go to File→Import.
6. Choose General→Existing Projects into Workspace.
7. Click on Next.
8. To select the project's root directory, click on Browse.
9. Navigate to `/home/training/tutorial/streams`
10. Click on OK.

The Projects section of the Eclipse screen will update and add the StreamsExercise project.

11. Examine the code by clicking on the disclosure arrows in the left-hand pane to reveal `StreamsExercise.java` under `streams/src/main/java/streamdemo`. Then run it, by clicking on the green 'run' arrow in the toolbar.
12. Use the `kafka-console-consumer` to view the contents of the new topic, `UppercasedDmesg1. It should contain the same text as the` `dmesg` topic, but with all the text turned to upper case.

```
$ kafka-console-consumer \
--new-consumer \
--bootstrap-server broker101:9092 \
--topic UppercasedDmesg \
--from-beginning
```

STOP HERE. THIS IS THE END OF THE EXERCISE.