Confluent Custom Training

Hands-On Exercise Manual

B5/608/A

Table of Contents

Introduction

The Hands-On Exercises in this course use a pre-configured Virtual Machine (VM) running Linux. We have installed Confluent Platform 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

Most 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 confluent-training-vm 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`.

Code Directories

For the Hands-On Exercises that involve code, you will find information at the beginning of the Exercise telling you where the project directory is. The location is relative to `/home/training/developer/exercise-code`.

Programming Exercises

For the programming exercises, we have provided three directories: `stubs`, `partial`, and `solution`. `solution`

contains a working sample solution to the problem. `partial` contains a partial solution, with some lines left for you to complete – marked with `TODO` comments. `stubs` are essentially empty. If you are feeling confident about the task, start from the `stubs` files; for more help, use `partial`; and feel free to look in `solution` for hints as you go.

Hands-On Exercise: Using Kafka's Command-Line Tools

In this Hands-On Exercise you will start to become familiar with Kafka's command-line tools. You will:

- Verify Kafka is running
- Use a console program to publish a message
- Use a console program to consume a message

Verifying that Kafka is Running Correctly

There are various ways to verify that all of the Kafka system's daemons are running. 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 class, 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
```

Console Publishing and Subscribing

Kafka has built-in command line utilities to publish messages to a topic, and read messages from a topic. These are extremely useful to verify that Kafka is working correctly, and for testing and debugging.

4. Run the command:

```
$ kafka-console-producer
```

This will bring up a list of parameters that the `kafka-console-producer` program can receive. Take a moment to look through the options. We will discuss many of their meanings later in the course.

5. Run `kafka-console-producer` again with the required arguments:

```
$ kafka-console-producer --broker-list broker1:9092 --topic testing
```

The `kafka-console-producer` program does not have a prompt character. It will simply wait for input.

6. Type:

```
some data
```

And hit 'Enter'. This will publish a message with the value `some data` to the `testing` topic. You will see the following output the first time you enter a line of data:

```
WARN Error while fetching metadata with correlation id 0 :
{testing=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
```

This is because the Topic `testing` does not initially exist. When the first line of text is published to it from `kafka-console-producer`, it will automatically be created.

7. Type:

```
more data
```

And hit 'Enter'.

8. Type:

```
final data
```

And hit 'Enter'.

9. Press `Ctrl-c` to exit the `kafka-console-producer` program.

10. Now we will use a Consumer to retrieve the data that was published. Run the command:

```
$ kafka-console-consumer
```

This will bring up a list of parameters that the `kafka-console-consumer` can receive. Take a moment to look through the options.

11. Run `kafka-console-consumer` again with the following arguments:

```
$ kafka-console-consumer \
--bootstrap-server broker1:9092 \
--new-consumer \
--from-beginning \
--topic testing
```

You should see all the messages that you published using `kafka-console-producer` earlier.

12. Press `Ctrl-c` to exit `kafka-console-consumer`.

If You Have More Time

13. The `kafka-console-producer` and `kafka-console-consumer` programs can be run at the same time. Run `kafka-console-producer` and `kafka-console-consumer` in separate terminal windows at the same time to see how `kafka-console-consumer` receives the events.

14. By default, `kafka-console-producer` and `kafka-console-consumer` assume null keys. They can also be run with appropriate arguments to write and read keys as well as values. Re-run the Producer with additional arguments to write (key,value) pairs to the topic:

```
$ kafka-console-producer \
--broker-list broker1:9092 \
--topic testing \
--property parse.key=true \
--property key.separator=,
```

(When you enter data, separate the key and the value with a comma.)

Then re-run the Consumer with additional arguments to print the key as well as the value:

```
$ kafka-console-consumer \
--bootstrap-server broker1:9092 \
--new-consumer \
--from-beginning \
--topic testing \
--property print.key=true
```

15. Kafka's data in ZooKeeper can be accessed using the `zookeeper-shell` command. Run it with:

```
$ zookeeper-shell zookeeper1
```

Use the `ls` command to view the directory structure in ZooKeeper.

```
ls /
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Consuming from Multiple Partitions

In this Hands-On Exercise, you will create a topic with multiple Partitions, Produce data to those Partitions, and then read it back to observe issues with ordering.

1. Create a topic manually with Kafka's command-line tool, specifying that it should have two Partitions:

```
$ kafka-topics \
--zookeeper zookeeper1:2181 \
--create \
--topic two-p-topic \
--partitions 2 \
--replication-factor 1
Created topic "two-p-topic"
```

2. Use the command-line Producer to write several lines of data to the topic.

```
$ seq 1 100 > numlist
$ kafka-console-producer \
--broker-list broker1:9092 \
--topic two-p-topic < numlist
```

3. Use the command-line Consumer to read the topic

```
$ kafka-console-consumer \
--bootstrap-server broker1:9092 \
--new-consumer \
--from-beginning \
--topic two-p-topic
```

4. Note the order of the numbers. Rerun the Producer command in step 2, then rerun the Consumer command in step 3 and see if you observe any difference in the output.

5. What do you think is happening as the data is being written?

6. Try creating a new topic with three Partitions and running the same steps again.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Writing a Basic Producer

Project directory: `helloworld`

In this Hands-On Exercise you will write a Producer.

Choosing a Language

In this Exercise, you will write programs using the Kafka API.

Using Eclipse

1. Go to the project directory (specified at the beginning of the Exercise description).
2. Create the Eclipse project with Maven:

```
$ mvn eclipse:eclipse
```

3. Open Eclipse.
4. Go to File→Import.
5. Choose General→Existing Projects into Workspace.
6. Click on Next.
7. To select the project's root directory, click on Browse.
8. Navigate to `/home/training/developer/exercise_code/helloworld`
9. Click on OK.

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

10. Click on Finish.

You will need to follow a similar procedure for each of the subsequent programming Hands-On Exercises in order to create the project and import it into Eclipse.

Creating a Producer

11. Create a KafkaProducer with the following characteristics:
    - Connects to `broker1:9092`
    - Sends five messages to a topic named `hello_world_topic`
    - Sends all messages as type `String`

Testing Your Code

13. To test your code, run the command-line Consumer and have it consume from `hello_world_topic`.

Do not attempt to write the Consumer code yet – we will do that in the next Hands-On Exercise.

14. Append `--property print.key=true` to the command-line Consumer to print the key as well as the value.

```
$ kafka-console-consumer \
--bootstrap-server broker1:9092 \
--new-consumer \
--from-beginning \
--topic hello_world_topic \
--property print.key=true
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Writing a Basic Consumer

Project directory: `helloworld`

In this Hands-On Exercise, you will write a basic Consumer.

Creating a Consumer

1. Write a Java Consumer to read the data you wrote with the Producer you created in the previous Exercise. If you did not have time to finish that Exercise, run the sample solution to write some messages to the topic first.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Accessing Previous Data

Project directory: `previous_data`

In this Hands-On Exercise, you will create a Consumer which accesses data starting from the beginning of the topic each time it launches.

Previous Data Consumer

1. Write a Consumer which reads all data from a topic each time it starts. Test it using the command-line Producer, or by reading from one of the topics you previously created.

If You Have More Time 1

2. Write a multi-threaded Consumer to read data from the `two-p-topic` topic you created in a previous exercise, and print it to the screen. As you print out the data, report which thread received the message. Investigate what happens if you use more threads than there are partitions in the topic.

If you have more time 2

3. Write a Producer (or modify an already existing one) to write a large number of messages to a topic. Make the messages sequential numbers (1, 2, 3...).
4. Write a Consumer which processes records and manually manages offsets by writing them to files on disk; it should write the offset after each message. (For simplicity, use a separate file for each Partition of the topic.) To 'process' the message, just display it to the console. When the Consumer starts, it should seek to the correct offset such that it provides exactly-once processing. Modify your Consumer so that it halts randomly during processing, so you can confirm that it performs correctly when restarted.

If you have even more time

5. Create a topic with two Partitions. Write a Producer which uses a custom Partitioner; it should write random numbers between 1 and 20 to the topic. Send numbers between 1 and 10 to one Partition, and numbers between 11 and 20 to the other. Check that it works correctly by using the multi-threaded Consumer you wrote earlier to read back the data using two threads.

STOP HERE. THIS IS THE END OF THE EXERCISE.

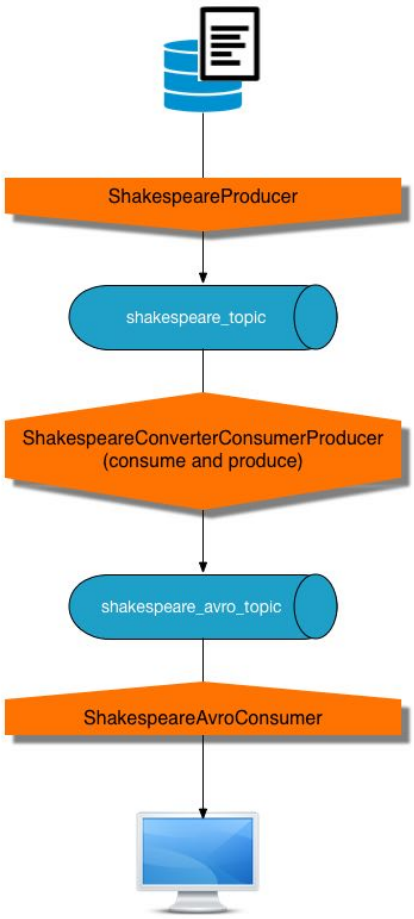
Hands-On Exercise: Using Kafka with Avro

Project directory: `kafka_avro`

In this Hands-On Exercise, you will write and read Avro data to and from a Kafka cluster, and build a processing pipeline to turn text data from one topic into Avro in another. To do this, you will:

- Write a Producer
- Write a program with a Consumer that reads data, converts it to Avro, then uses a Producer to write that Avro data to a new topic
- Write a Consumer that reads Avro data from a topic and displays it to the screen

The workflow is shown below:



The Dataset

This exercise will be using a subset of Shakespeare's plays. The files are located in `/home/training/developer/datasets/shakespeare`. Here is an example line from one of the files:

```
2360  Et tu, Brute?-- Then fall, Caesar!
```

Notice that the line of the play is preceded by the line number.

When Was That Play Written?

In the code you write, you will need to add the year that the play was written to the data. Here are the relevant years:

Hamlet: 1600
Julius Caesar: 1599
Macbeth: 1605
Merchant of Venice: 1596
Othello: 1604
Romeo and Juliet: 1594

Creating the Avro Schemas

You need to create an Avro schema for the key and value of the message.

1. Make sure you are in the right directory. The `*.asvc` schema files should be in the relative path `kafka_avro/src/main/avro`
2. Create a schema for the key with following characteristics:
   - The name should be `ShakespeareKey`
   - The schema should have a field for the name of the work, *e.g.*, Julius Caesar
   - The schema should have a field for the year the work was published, *e.g.*, 1599
3. Create a schema for the value with following characteristics:
   - The name should be `ShakespeareValue`
   - The schema should have a field for the line number
   - The schema should have a field for the line itself
4. Once you have created the schema files, generate the corresponding Java classes with:

```
$ mvn generate-sources
```

5. Verify that the corresponding java files were created for the `ShakespeareValue` and `ShakespeareKey` Avro schemas in the relative path `kafka_avro/src/main/java/partial/model`
6. If you have already imported the `kafka_avro` project into Eclipse, refresh it by right-clicking on the project in the left-hand pane and selecting 'Refresh'.

Kafka Consumers and Producers

In this Exercise, you are creating a processing pipeline. A Publisher will read in every line of Shakespeare as a `String` and write it to a topic. A Consumer will read the data from that first topic, parse it, create Avro objects, and publish the Avro objects to a new topic. The final piece of the pipeline is a Consumer that reads the Avro objects from the topic and writes them to the screen with `System.out.println`.

7. Create a Producer with the following characteristics:
    - Connects to `broker1:9092`
    - Reads in the files from the `shakespeare` directory line by line
    - Sends each line as a separate message
        - Sends messages to a topic named `shakespeare_topic`
        - Sends all messages as type `String`
        - The key should be the name of the play (which is contained in the filename)
        - The value should be the line from the play

Note: If you run your Producer from within Eclipse, specify the source directory as an argument. Choose `Run As` → `Run Configurations` → `Arguments` and specify the directory name.

8. Create a program containing a Consumer and Producer with the following characteristics:
    - Consumes messages from `shakespeare_topic`
        - Always starts from the beginning of the topic
    - Reads all keys and values as type `String`
    - Converts the key and value to Avro objects
    - Writes messages to a topic named `shakespeare_avro_topic`
        - The key should be a `ShakespeareKey` Avro object
        - The value should be a `ShakespeareValue` Avro object

9. Create a Consumer with the following characteristics:

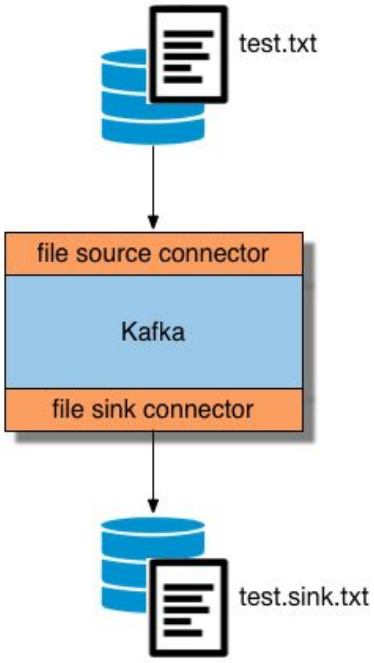
- Consumes messages from `shakespeare_avro_topic`
  - Always starts from the beginning of the topic
- Consumes all data as Avro objects
- Outputs the key and value of each message to the screen

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Running Kafka Connect in Standalone Mode

In this Hands-On Exercise, you will run Kafka Connect in standalone mode with two Connectors: a file source Connector, and a file sink Connector. The file source Connector will read lines from a file and produce them to a Kafka topic. The file sink Connector will consume from the same Kafka topic and write messages to a file.

The workflow is shown below:



1. Read through the `connect-standalone.properties` file. This file configures the standalone worker.

```
$ cat /etc/kafka/connect-standalone.properties
```

2. Read through the `connect-file-source.properties` file. This file configures the file source connector.

```
$ cat /etc/kafka/connect-file-source.properties
```

3. Seed a test file with some test data:

```
$ echo -e "log line 1\nlog line 2" > test.txt
```

4. Run the standalone connector with the file source connector. Note that it may take a few seconds to start.

```
$ connect-standalone \
/etc/kafka/connect-standalone.properties \
/etc/kafka/connect-file-source.properties
```

Leave the standalone connector running in the foreground in one terminal.

5. In a separate terminal, run the console consumer to ensure the lines were written to Kafka:

```
$ kafka-console-consumer \
--bootstrap-server broker1:9092 \
--from-beginning \
--topic connect-test \
--new-consumer
```

You should see two JSON messages, each of whose payload has one line of the source file.

You should leave the console consumer running for the duration of the Exercise.

6. Read through the `connect-file-sink.properties` file. This file configures the file sink connector:

```
$ cat /etc/kafka/connect-file-sink.properties
```

7. In the original terminal with Connect running, press `Ctrl-c` to stop the Connector. (Two standalone Connectors cannot run at the same time because Connect binds to a port, even in standalone mode. More on this in a later exercise.)

8. Now, restart Connect in standalone mode, passing in both the source and sink Connector configuration files:

```
$ connect-standalone \
/etc/kafka/connect-standalone.properties \
/etc/kafka/connect-file-source.properties \
/etc/kafka/connect-file-sink.properties
```

Leave the standalone connector running in the foreground in one terminal.

9. Notice that a new file has been created in the current working directory, called `test.sink.txt`. View this file to

see that the sink connector did what it was expected to do:

```
$ cat test.sink.txt
```

10. In a separate terminal, append a new line to the source file, `test.txt`:

```
$ echo "log line 3" >> test.txt
```

11. View the sink file, `test.sink.txt`:

```
$ cat test.sink.txt
```

Notice that the new log line has been written to the sink file as well. Kafka Connect read the new line from the source file and wrote it to the topic; it then read from the topic and appended the new message to the sink file.

12. Switch to the terminal running the console consumer that you started in Step 5. Notice that the new log line has appeared.

13. In each terminal window, press `Ctrl-c` to terminate the process that is running.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Hands-On Exercise: Using the Kafka Connect REST API

In this Hands-On Exercise, you will interact with the REST API to get a Connector's configuration and update it.
The REST API is the only way to create and update Connectors in distributed mode. It can be used in standalone mode, but as shown in the previous exercise, standalone mode is easier to use with the command line. However, to make this exercise simple, the REST calls in this exercise will be made against Connect running in standalone mode.

1. Start the file source and sink connectors in the same way they were started in the previous example.

```
$ connect-standalone \
/etc/kafka/connect-standalone.properties \
/etc/kafka/connect-file-source.properties \
/etc/kafka/connect-file-sink.properties
```

2. In a separate terminal, run a `GET` query on the `/connectors` endpoint to get a list of active connectors:

```
$ curl http://localhost:8083/connectors
["local-file-source","local-file-sink"]
```

3. Get the configuration of the `local-file-sink` connector:

```
$ curl http://localhost:8083/connectors/local-file-sink/config
```

4. Run the previous command again, but this time redirect the output to a file:

```
$ curl http://localhost:8083/connectors/local-file-sink/config > local-
file-sink.config.json
```

5. Modify `local-file-sink.config.json`, by changing the file the sink connector writes to. Open the file in your favorite editor and change `test.sink.txt` to `test-new.sink.txt`. Then, perform a REST `PUT` query to update the configuration:

```
$ curl -X PUT http://localhost:8083/connectors/local-file-sink/config \
-d @local-file-sink.config.json \
--header "Content-Type: application/json"
```

6. Append a new line to the source file:

```
$ echo "foobarbaz" >> test.txt
```

7. Notice that the original sink file `test.sink.txt` does not contain the new line. However, the new sink file `test-new.sink.txt` does:

```
$ cat test.sink.txt
$ cat test-new.sink.txt
```

8. In the terminal window where you started the connectors, hit `Ctrl-c` to terminate the process.

STOP HERE. THIS IS THE END OF THE EXERCISE.