Confluent Stream Processing using Apache Kafka® Streams

Exercise Book

Version 7.0.0-v1.0.3

Table of Contents

Copyright & Trademarks

Lab 00 Introduction

a. Introduction

This document provides Hands-On Exercises for the course **Confluent Stream Processing using Apache Kafka® Streams**. You will use a setup that includes a virtual machine (VM) configured with Apache Kafka and Confluent tools to manage your data and clusters.

Alternative Lab Environment

As an alternative you can:

- Download the VM to your laptop and run it in VirtualBox. Make sure you have the newest version of VirtualBox installed. Download the VM from this link:

  https://s3.amazonaws.com/confluent-training-images-us-east-1/training-ubuntu-20-04-jan2022.ova

- If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine then you can run the labs there. But please note that your trainer might not be able to troubleshoot any potential problems if you are running the labs locally. If you choose to do this, follow the instructions at → Running Labs in Docker for Desktop.

Command Line Examples

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

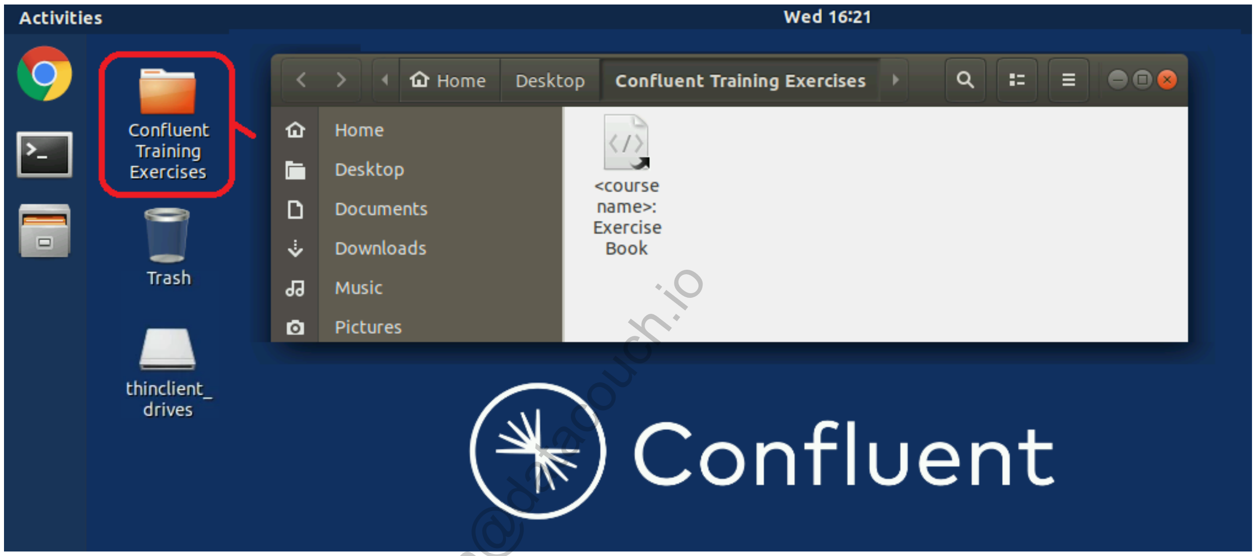
```
$ pwd
/home/training
```

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

Preparing the Labs

Welcome to your lab environment! You are connected as user **training**, password **training**.

If you haven't already done so, you should open the **Exercise Guide** that is located on the lab virtual machine. To do so, open the **Confluent Training Exercises** folder that is located on the lab virtual machine desktop. Then double-click the shortcut that is in the folder to open the **Exercise Guide**.



Copy and paste works best if you copy from the Exercise Guide on your lab virtual machine.

- Standard Ubuntu keyboard shortcuts will work: `Ctrl+C` → Copy, `Ctrl+V` → Paste
- In a Terminal window: `Ctrl+Shift+C` → Copy, `Ctrl+Shift+V` → Paste.

If you find these keyboard shortcuts are not working you can use the right-click context menu for copy and paste.

1. Open a terminal window
2. Clone the source code repository to the folder `confluent-streams` in your **home** directory:

```
$ cd ~
$ git clone --depth 1 --branch 7.0.0-v1.0.3 \
    https://github.com/confluentinc/training-kafka-streams-src.git \
    confluent-streams
```

❗ If you chose to select another folder for the labs then note that many of our samples assume that the lab folder is `~/confluent-streams`. You will have to adjust all those command to fit your specific environment.

3. Navigate to the `confluent-streams` folder:

```
$ cd ~/confluent-streams
```

4. Start the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka
```

You should see something similar to this:

```
Creating network "confluent-streams_kafka-net" with the default
driver
Creating kafka            ... done
Creating zookeeper        ... done
```

In the first steps of each exercise, you may launch the containers needed for the exercise with a `docker-compose up` command. Simply typing `docker-compose up -d` will start all of the containers defined in the docker-compose.yml file. You can start fewer containers by specifying only those you want to run, for example: `docker-compose up -d zookeeper kafka`.

The majority of the exercises use the docker-compose.yml file in the ~/confluent-streams directory. The `docker-compose up` command will search up the directory hierarchy until it finds a docker-compose.yml file, so the one in the confluent-streams directory will usually be used. The exception is the docker-compose.yml file used in the security exercise as this has additional security settings. See the comments at the beginning of Lab 11 Securing a Kafka Streams Application.

If at any time you want to get your environment back to a clean state use `docker-compose down` to end all of your containers. Then return to your last `docker-compose up` to get back to the beginning of an exercise.

Exercises do not need to be completed in order. You can start from the beginning of any exercise at any time.

If you want to completely clear out your docker environment use the script on the VM: `docker-nuke.sh`. The nuke script will forcefully end all of your running docker containers.

5. Monitor the cluster with:

```
$ docker-compose ps

   Name              Command              State              Ports
----------------------------------------------------------------------
-------
kafka        /etc/confluent/docker/run   Up      0.0.0.0:9092-
>9092/tcp
zookeeper    /etc/confluent/docker/run   Up      2181/tcp, 2888/tcp,
3888/tcp
```

All services should have `State` equal to `Up`.

6. You can also observe the stats of Docker on your VM:

```
$ docker stats

CONTAINER ID     NAME         CPU %     MEM USAGE / LIMIT    MEM %   NET
I/O        BLOCK I/O     PIDS
ab9c97077e94     zookeeper    0.14%     88.14MiB / 9.737GiB 0.88%
106kB / 130kB    0B / 0B          48
ff47bece9e4f     kafka        1.17%     421.8MiB / 9.737GiB 4.23%
646kB / 522kB    0B / 0B          77
```

Cleanup

7. Press `Ctrl+C` to exit the Docker statistics.

8. Shut down your Kafka cluster with the `docker-compose down -v` command.

b. Continued Learning After Class

Once the course ends, the VM in Content Raven will terminate and you will no longer have access to it. However, you can still download the VM onto your own machine or use Docker locally to revisit these materials. We encourage you to bring up your own test environment, explore configuration files, inspect scripts, and perform tests. Here are some activities we encourage to reinforce your learning:

- Revisit the exercises in this manual
- Summarize and discuss the student handbook with your peers
- Consult the README in this public repository for more resources and your own development playground: https://github.com/confluentinc/training-kafka-streams-src

Conclusion

In this lab you have prepared and tested the Lab Environment. Finally you have created your Apache Kafka cluster that will be used in subsequent exercises.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 01 Introduction to Kafka Streams

a. Scaling a Kafka Streams Application

In this exercise, we're going to write a Kafka producer in either Python or Java that generates a stream of temperature readings for a set of weather stations. We are also writing a simple Kafka Streams application that will consume this topic and calculate the maximum temperature per station per time window. We will then run this application in a single instance and later scale it up to several instances. We will monitor the throughput with the **Confluent Control Center**.

Prerequisites

Please make sure you have prepared your lab environment as described here: → Lab Environment

Running the Kafka Cluster and Confluent Control Center

To be able to run Kafka Streams applications we need a working Kafka cluster. We will run one consisting of a single broker and zookeeper.

1. Locate the file `docker-compose.yml` in the `~/confluent-streams` directory.

   This `docker-compose.yml` file will be used to run a simple Kafka cluster as a backend for our Kafka Streams applications. If you're curious, please open the file and analyze its contents.

2. Navigate to the folder `labs/scaling`:

```
$ cd ~/confluent-streams/labs/scaling
```

3. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

Wait a couple of minutes until the cluster is initialized.

4. Create the two topics `temperature-readings` and `max-temperatures`, each with 3 partitions using these commands:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 3 \
    --topic temperature-readings

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 3 \
    --topic max-temperatures
```

Creating the Producer

Now it is time to create a temperature readings producer. You can do it either in Python or Java. We start with the Python producer. If you prefer Java then move ahead to →here.

Create the Producer in Python

5. Navigate to the folder `labs/scaling`, and launch VS Code:

```
$ cd ~/confluent-streams/labs/scaling/temp-producer
$ code .
```

If a pop-up appears when VS Code opens, check the box and click `Yes, I trust the authors`.

6. Locate the file `main.py`.

7. Inspect the code.

- We are defining a few (temperature measurement) stations, their respective average temperatures and last measured temperatures
- we're using the Confluent Python client for Kafka to create a producer
- every ~100 ms we're generating a temperature reading for one of the randomly selected station.

8. Return to the terminal window, and install the python prerequisite. Note this may already be installed from a previous exercise:

```
$ pip3 install --upgrade pip
$ pip3 install confluent-kafka
```

9. Run the producer:

```
$ python3 main.py
```

10. In another terminal window run the `kafka-console-consumer` to display the `temperature-readings` topic:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --max-messages 25 \
    --topic temperature-readings \
    --property print.key=true \
    --property key.separator=", "
```

You should see an output similar to this:

```
S-06, {"station": "S-06", "temperature": -1}
S-03, {"station": "S-03", "temperature": 8}
S-03, {"station": "S-03", "temperature": 9}
S-06, {"station": "S-06", "temperature": 0}
S-08, {"station": "S-08", "temperature": 31}
S-09, {"station": "S-09", "temperature": -7}
...
```

The `key` is the station and the `value` is a JSON object with the station and the temperature in degree Celsius.

Create the Producer in Java

11. Navigate to the folder `labs/scaling`, and launch VS Code:

```
$ cd ~/confluent-streams/labs/scaling/temp-producer
$ code .
```

12. Locate the file `build.gradle` and analyze its content. It is the build file for a simple Kafka client.

13. Locate the file `TempProducer.java` in the subfolder `src/main/java/streams` and open it. Analyze the file and make sure you understand the code. If necessary discuss with your peers.

14. Notice the file `log4j.properties` in the folder `src/main/resources` that configures logging for the producer.

15. Use **Run** → **Start Debugging** in VS Code or `./gradlew run` in the terminal to run the Java producer.

    The first time you run the debugger it may take extra time while resources are downloaded.

    You should see an output similar to this:

```
The record is: S-06, {"station": "S-06", "temperature": -1}
The record is: S-03, {"station": "S-03", "temperature": 8}
The record is: S-03, {"station": "S-03", "temperature": 9}
The record is: S-06, {"station": "S-06", "temperature": 0}
The record is: S-08, {"station": "S-08", "temperature": 31}
The record is: S-09, {"station": "S-09", "temperature": -7}
...
```

    The `key` is the station and the `value` is a JSON object with the station and temperature in degree Celsius.

16. From another terminal window, run the `kafka-console-consumer` to display the `temperature-readings` topic:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --max-messages 25 \
    --topic temperature-readings \
    --property print.key=true \
    --property key.separator=", "
```

You should see an output similar to this:

```
S-06, {"station": "S-06", "temperature": -1}
S-03, {"station": "S-03", "temperature": 8}
S-03, {"station": "S-03", "temperature": 9}
S-06, {"station": "S-06", "temperature": 0}
S-08, {"station": "S-08", "temperature": 31}
S-09, {"station": "S-09", "temperature": -7}
...
```

17. Jump to the next section "Writing the Kafka Streams Application".

Writing the Kafka Streams Application

18. Open a terminal window and navigate to the `streams-app` folder:

```
$ cd ~/confluent-streams/labs/scaling/streams-app
```

19. In the folder `confluent-streams/labs/scaling/streams-app` locate the file `build.gradle` and analyze its content.

> In addition to the usual libraries we also load the `monitoring-interceptors` library to be able to integrate with the Confluent Control Center and the `kafka-json-serializer` library for the JSON serde.

20. We will be using the Kafka Streams application called `StreamsApp.java` in the subfolder `src/main/java/streams` for this exercise.

21. You may choose to launch VS Code with `code .` to build and run the application. Or simply use gradle with `./gradlew run`.

22. Run an instance of `kafka-console-consumer` to display the `max-temperatures` topic.

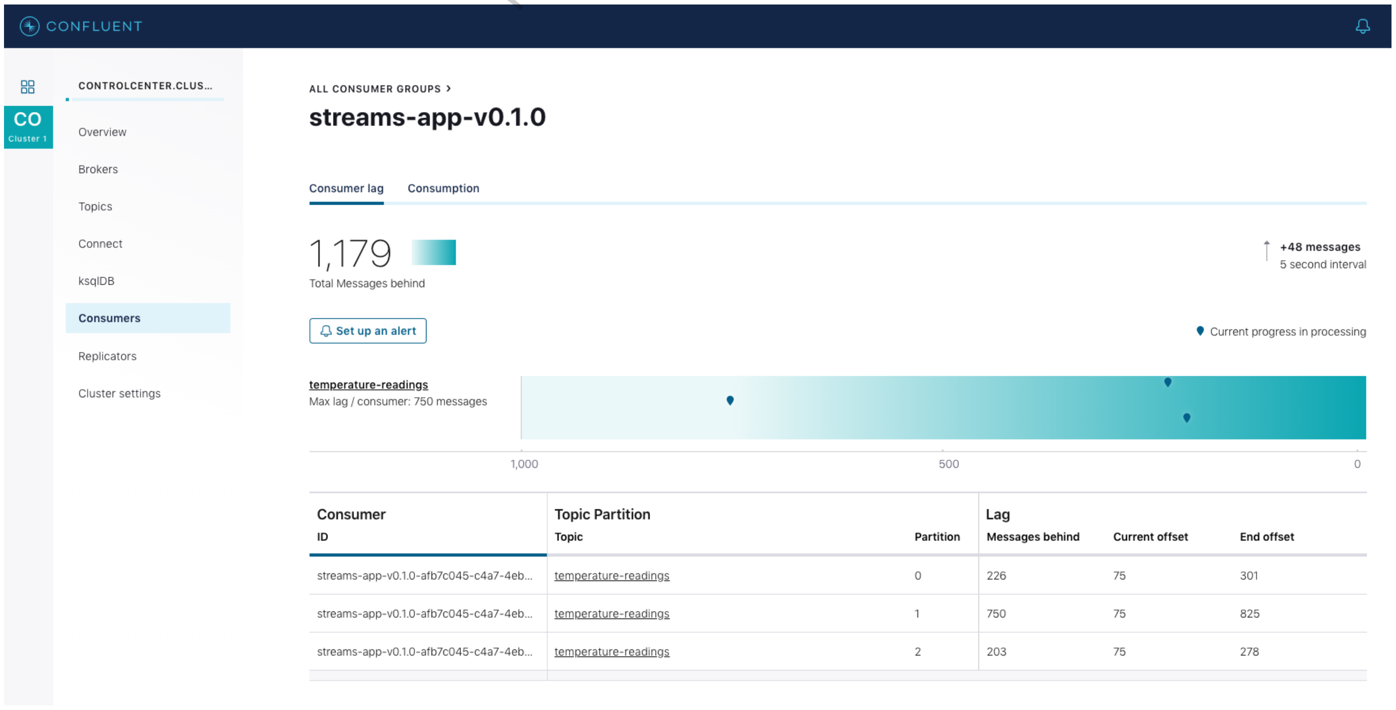
Note it may take some time for max temperatures to appear:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic max-temperatures
```

> Is the output surprising to you? Why? It is because of the nature of the `commit.interval.ms` property and how it relates to the output of KTables to Kafka topics. This will be discussed later in the course.
>
> Ignore the warnings:
> `2022-05-11 21:45:52 WARN ConsumerConfig:362 - The configuration 'admin.retry.backoff.ms' was supplied but isn't a known config. 2022-05-11 21:45:52 WARN ConsumerConfig:362 - The configuration 'admin.retries' was supplied but isn't a known config.`

23. Open Confluent Control Center at http://localhost:9021

24. In Control Center, click on `CONTROLCENTER.CLUSTER` and then under **Consumers**, monitor the consumer lag for the `streams-app-v.0.1.0`. Note that the consumer group falls more and more behind:



streams-app-v0.1.0

1,179
Total Messages behind

+48 messages
5 second interval

temperature-readings
Max lag / consumer: 750 messages

Consumer	Topic Partition		Lag		
ID	Topic	Partition	Messages behind	Current offset	End offset
streams-app-v0.1.0-afb7c045-c4a7-4eb...	temperature-readings	0	226	75	301
streams-app-v0.1.0-afb7c045-c4a7-4eb...	temperature-readings	1	750	75	825
streams-app-v0.1.0-afb7c045-c4a7-4eb...	temperature-readings	2	203	75	278

25. Alternatively, you can use the Kafka tool `kafka-consumer-groups` to check the consumer lag:

```
$ kafka-consumer-groups
    --bootstrap-server kafka:9092
    --group streams-app-v0.1.0
    --describe
GROUP              TOPIC                PARTITION  CURRENT-OFFSET
LOG-END-OFFSET  LAG    CONSUMER-ID...
streams-app-v0.1.0 temperature-readings 0          551
2417            1866   streams-app...
streams-app-v0.1.0 temperature-readings 1          363
5395            5032   streams-app...
streams-app-v0.1.0 temperature-readings 2          580
3614            3034   streams-app...
```

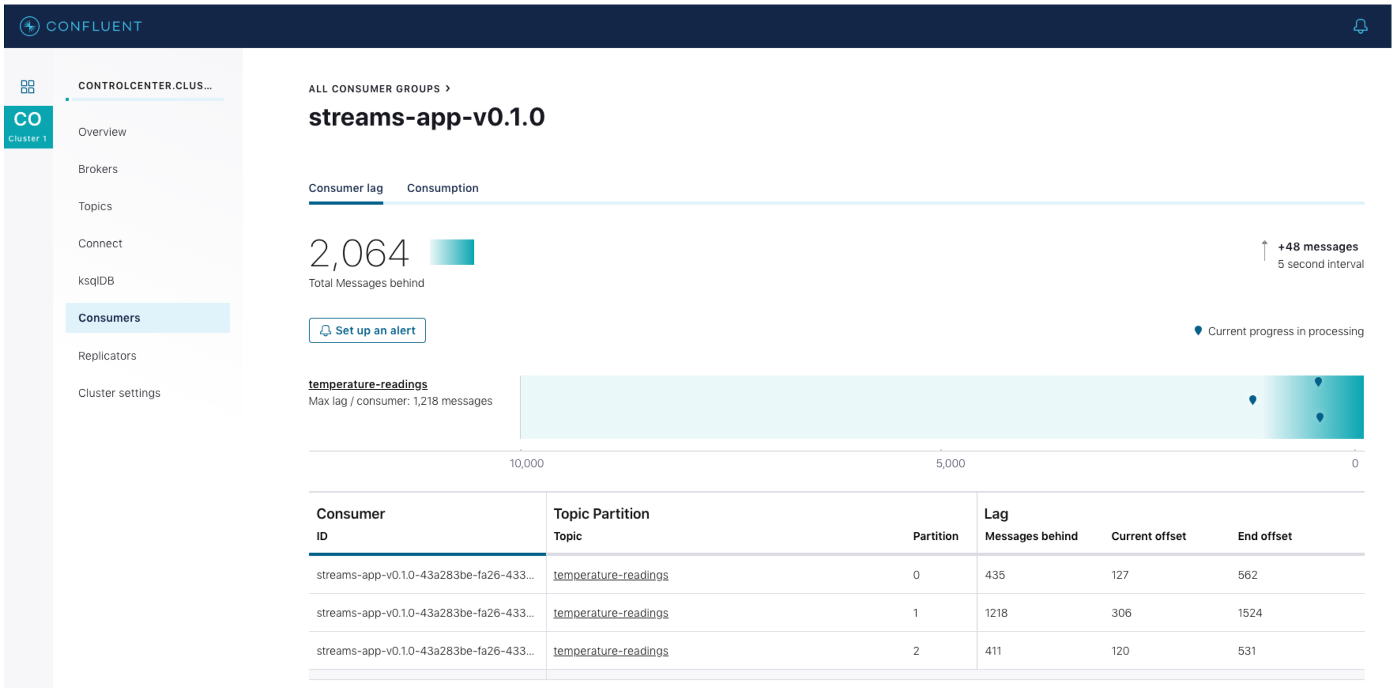
Scaling the Kafka Streams Application

26. Open another terminal window to run another instance of your Kafka Streams application:

```
$ cd ~/confluent-streams/labs/scaling/streams-app
$ ./gradlew run
```

27. In **Confluent Control Center** observe how the throughput of the streams app nearly doubles.

28. Note that we have now two consumer instances listed (recognizable by their ID):



Consumer ID	Topic Partition Topic	Partition	Lag Messages behind	Current offset	End offset
streams-app-v0.1.0-43a283be-fa26-433...	temperature-readings	0	435	127	562
streams-app-v0.1.0-43a283be-fa26-433...	temperature-readings	1	1218	306	1524
streams-app-v0.1.0-43a283be-fa26-433...	temperature-readings	2	411	120	531

29. Also observe that the consumer lag increases more slowly...

30. Now scale the streams app to 3 instances and again monitor an increase in throughput and reduction in consumer lag.

31. Finally scale the app again, this time to 4 instances. Monitor the throughput after scaling the app. What are you observing? Explain your observation. See the conclusion for an explanation of what happens here.

Cleanup

32. Stop the producer, consumers, and stream application with `Ctrl+C` in the terminal or **Run** → **Stop Debugging** in VScode.

33. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

In this exercise we have created a Kafka Streams application that processed an input topic and produced an output topic. First we ran only one application instance and then we scaled the application up to several instances. We noticed a significant boost in throughput until the number of instances was greater than the number of partitions of the input topic. At this point the additional application instances were sitting there idle.

In the solutions folder, the Java producer and Kafka Streams app include Dockerfiles so that they can be deployed as containers. A separate `docker-compose.services.yml` has also been provided to start the microservices.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 02 Working with Kafka Streams

This lab contains 2 exercises:

- Anatomy of a Kafka Streams App
- Working with JSON

a. Anatomy of a Kafka Streams App

In this exercise, you will create a Kafka Streams application and deploy it using Gradle (or, optionally, Maven). The purpose of this exercise is to illustrate the structure of Kafka Streams application code and the routine of deploying code with build tools.

The application itself reads data from a topic whose keys are integers and whose values are sentence strings. The application transforms the input so that the strings are lower-case and output to a new topic.

Prerequisites

Please make sure you have prepared your lab environment as described here: → Lab Environment

Preparing the Kafka Cluster

1. Navigate to the folder `~/confluent-streams/labs/working-streams`:

```
$ cd ~/confluent-streams/labs/working-streams
```

2. From within the `working-streams` folder run the cluster with the following command:

```
$ docker-compose up -d zookeeper kafka
Creating network "confluent-streams_kafka-net" with the default
driver
Creating kafka              ... done
Creating zookeeper          ... done
```

3. Double check that the cluster is up and running:

```
$ docker-compose ps
```

you should see something similar to this:

```
         Name                        Command            State
Ports
----------------------------------------------------------------------
--------------------
kafka                       /etc/confluent/docker/run   Up
0.0.0.0:9092->9092/tcp
zookeeper                   /etc/confluent/docker/run   Up
2181/tcp, 2888/tcp, 3888/tcp
```

Make sure all services have `State=Up`.

4. Create an input topic called `lines-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-topic
```

5. Create an output topic called `lines-lower-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-lower-topic
Created topic "lines-lower-topic".
```

6. Let's now check what topics are in Kafka using this command:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --list
```

We should see something like this:

```
__confluent.support.metrics
lines-lower-topic
lines-topic
```

If you need to delete a topic, say the one with name `<topic name>`, (e.g. to start over) you can use this command:

```
$ kafka-topics \
    --bootstrap-server kafka:9092 \
    --delete \
    --topic <topic name>
```

Now we are ready to create, build and run our first **Kafka Streams** application. We will first build and run the application using Java in conjunction with **Gradle**. Optionally, we will also see how to build and run the application with **Maven**.

Authoring the Kafka Streams Application using Java & Gradle

7. Navigate to the folder `gradle-sample` with in the `working-streams` folder, and launch VS Code:

   Be certain to include the period in the `code .` command below. That indicates starting VS Code in the current directory - otherwise some references may not correctly resolve.

```
$ cd ~/confluent-streams/labs/working-streams/gradle-sample
$ code .
```

8. In this folder locate the `build.gradle` file and open it to analyze its content.

   The external dependencies for a simple **Kafka Streams** app are the `kafka-clients` and `kafka-streams` libraries. We also add the `slf4j-log4j12` dependency for logging purposes.

9. In VS Code, open the file `MapSample.java` in the subfolder `src/main/java/streams` and inspect its content. From here, you can challenge yourself to complete the TODOs, or you can move forward to see a step-by-step walkthrough of the code. If you decide to challenge yourself, you can always peek at the corresponding subfolder in `~/confluent-streams/solutions/` if you get stuck.

10. Now we start to add actual **Kafka Streams** application logic. We will start with the configuration part. Please add this code snippet to the `main` method of the class (right after the initial `println`):

```
Properties settings = new Properties();
settings.put(StreamsConfig.APPLICATION_ID_CONFIG, "map-sample-
v0.1.0");
settings.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
```

    We're providing an ID to our application and tell it where to find the Kafka cluster. This is the minimal configuration needed!

11. Next we will define the topology of our **Kafka Streams** application. Add this snippet right after the configuration part:

```
final Serde<String> stringSerde = Serdes.String();
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> lines = builder
    .stream("lines-topic", Consumed.with(stringSerde, stringSerde));
KStream<String, String> transformed = lines
    .mapValues(value -> value.toLowerCase());
transformed.to("lines-lower-topic", Produced.with(stringSerde,
stringSerde));
Topology topology = builder.build();
```

    We're defining a builder for the topology, use it to create a `KStream` from the Kafka topic `line-topic` using `String` Serdes for both key and value of the messages. Then we're using a `mapValues` function on the `KStream` to convert the `value` into all lower case. Finally we're writing the result into the Kafka topic `line-lower-topic`. Then we build the topology.

12. With the settings and the topology at hand we can now create the **Streams** app:

```
KafkaStreams streams = new KafkaStreams(topology, settings);
```

Our application will now start consuming data from the input topic, transform it and write it to the output topic.

13. To have our application terminate in an orderly way when requested without leaving any resource leaks behind we add a **shutdown hook** at the end of the main method:

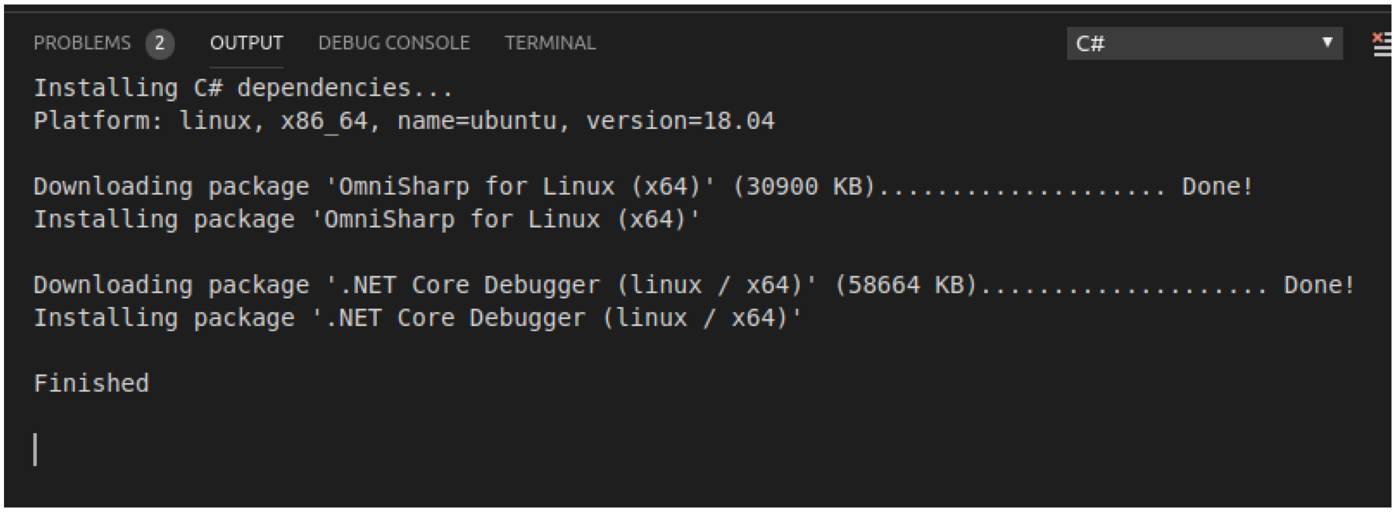
```
final CountDownLatch latch = new CountDownLatch(1);
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    System.out.println("### Stopping Map Sample Application ###");
    streams.close();
    latch.countDown();
 }));

try{
    streams.start();
    latch.await();
} catch (final Throwable e) {
    System.exit(1);
}
System.exit(0);
```

This **Shutdown Hook** will be executed when the application receives a `SIG_TERM` signal. The `CountDownLatch` is used as a best practice to avoid rare cases of deadlock. Notice the use of the `await()` method after the application starts.

14. Within the project folder `gradle-sample` locate the folder for the Java resources `src/main/resources`. In this folder we have a file `log4j.properties`. This file is used to configure the logger for our Kafka Streams app.

Use `INFO` instead of `WARN` for the `rootLogger` if you want to be more verbose in the logs.

And that's all we need. Our first **Kafka Streams** app is ready to go! Use **Run → Start Debugging** to run your code.



```
Installing C# dependencies...
Platform: linux, x86_64, name=ubuntu, version=18.04

Downloading package 'OmniSharp for Linux (x64)' (30900 KB).................... Done!
Installing package 'OmniSharp for Linux (x64)'

Downloading package '.NET Core Debugger (linux / x64)' (58664 KB).................... Done!
Installing package '.NET Core Debugger (linux / x64)'

Finished
```

In the VS Code DEBUG CONSOLE tab you should see something like this:

```
*** Starting Map Sample Application ***
2018-07-04 13:47:36 WARN  ConsumerConfig:287 - The configuration
'admin.retries' was supplied but isn't a known config.
```

At this time you can safely ignore the WARN log items.

At this time nothing will happen since we did not yet produce any data in the `lines-topic` in Kafka. Let your Kafka Streams app remain running in the VS Code debugger. This will be our next task **after** we have shown how to build the same app using Maven instead of Gradle.
If you want to skip the **Maven** part then go to Producing some Input Data.

Optional: Build and Run the application with Java & Maven

Here we're basically showing the same steps as in the previous section, except we will now use **Maven** instead of **Gradle**.

15. Navigate to the `maven-sample` folder, and launch VS Code:

```
$ cd ~/confluent-streams/labs/working-streams/maven-sample
$ code .
```

16. Locate the `pom.xml` file, open it and analyze its content.

> this is the minimum Maven file required to compile, package and run a Java application that has external dependencies on the 2 required libraries `kafka-clients` and `kafka-streams`. It also uses the `slf4j-log4j12` library for logging.

17. Locate the file `MapSample.java` in the subfolder `src/main/java/streams` and open it. Double check that it looks similar to the one you created in the Gradle example.

18. Note the same file `log4j.properties` in the folder `src/main/resources` as in the Gradle example.

> Use `INFO` instead of `WARN` for the `rootLogger` if you want to be more verbose in the logs.

19. Solution code has been provided in this exercise, so our first **Kafka Streams** app is ready to go! Use **Run** → **Start Debugging** to run your code.

    You should see something like this:

```
*** Starting Map Sample Application ***
2018-07-04 13:53:12 WARN  ConsumerConfig:287 - The configuration
'admin.retries' was supplied but isn't a known config.
```

    At this time you can safely ignore the WARN log items.

> At this time nothing will happen since we did not yet produce any data in the `lines-topic` in Kafka. Let's do this next. Let your Kafka Streams app remain running in the VS Code debugger.

Producing some Input Data

We're going to use the `kafka-console-producer` tool to create some input data in the topic `lines-topic`.

20. Open a new terminal window and navigate to the `working-streams` folder:

```
$ cd ~/confluent-streams/labs/working-streams
```

21. From a terminal window execute this command:

```
$ cat << EOF | kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --property "parse.key=true" \
    --property "key.separator=:" \
    --topic lines-topic
1:"Kafka powers the Confluent Streaming Platform"
2:"Events are stored in Kafka"
3:"Confluent contributes to Kafka"
EOF
```

This writes 3 entries into the topic called `lines-topic` using `String` serializers for both key and value.

Reading the Transformed Messages

Here we're using the `kafka-console-consumer` tool to read from the output topic. We're again using String deserializers for key and value.

22. In the same terminal window, run:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic lines-lower-topic
```

You should see this:

```
"kafka powers the confluent streaming platform"
"events are stored in kafka"
"confluent contributes to kafka"
```

You might need to be a bit patient until the messages appear. It can take a few seconds to a minute or so depending on the performance of your computer…

Cleanup

23. Terminate the Kafka console consumer by pressing `Ctrl+C`.
24. Terminate your **Kafka Streams** application with **Run** → **Stop Debugging**. If you used the `./gradlew run` command instead, you can terminate with `Ctrl+C`.
25. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

We have created our first complete **Kafka Streams** application. We built the application using Gradle, and then again with Maven. We have used the command line tools provided by Kafka to produce input data and display the transformed output data.

STOP HERE. THIS IS THE END OF THE EXERCISE.

b. Working with JSON

The purpose of this exercise is to learn how to create serializers and deserializers for custom Java objects.

In this case, we will create a Serde for an object that records temperature data using the `KafkaJsonSerializer` and `KafkaJsonDeserializer` helper classes. The application itself reads temperature data from an input topic, filters for temperatures higher than 25 degrees, and outputs that data to a new output topic.

Prerequisites

1. Navigate to this lab's folder `~/confluent-streams/labs/working-streams`:

   ```
   $ cd ~/confluent-streams/labs/working-streams
   ```

2. If it is not already running, start the Kafka cluster:

   ```
   $ docker-compose up -d zookeeper kafka
   ```

   Do not proceed until all services are up and running; test with:

   ```
   $ docker-compose ps
   ```

   and assert that all services are in state `Up`.

3. Create an input topic called `temperatures-topic` and an output topic called `high-temperatures-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic temperatures-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic high-temperatures-topic
```

Writing the Kafka Streams App

4. Open a new terminal window and navigate to the `working-streams/json-sample` folder, and launch VS Code:

```
$ cd ~/confluent-streams/labs/working-streams/json-sample
$ code .
```

5. Locate the file `build.gradle` in this folder and open it to analyze its content.

> Compared to the `build.gradle` file in the previous exercise we have added the `kafka-json-serializer` library for the JSON serializer/deserializer

6. In the subfolder `src/main/java/streams` locate the file `JsonSample.java` and familiarize yourself with the code. It basically does the configuration of the **Kafka Streams** app but the interesting code is missing. As before, you can challenge yourself to implement the missing code or follow the step-by-step instructions in this book.

7. To create a **Serde** (Serializer/Deserializer) for JSON formatted data, add this code to the function `getJsonSerde()` after the **TODO** comment:

```
Map<String, Object> serdeProps = new HashMap<>();
serdeProps.put("json.value.type", TempReading.class);

final Serializer<TempReading> temperatureSerializer = new
KafkaJsonSerializer<>();
temperatureSerializer.configure(serdeProps, false);

final Deserializer<TempReading> temperatureDeserializer = new
KafkaJsonDeserializer<>();
temperatureDeserializer.configure(serdeProps, false);

return Serdes.serdeFrom(temperatureSerializer,
temperatureDeserializer);
```

We're basically using the two helper classes `KafkaJsonSerializer` and `KafkaJsonDeserializer` to create a serializer and a deserializer which in turn we then use to create a **Serde**.
We will use this Serde to serialize and deserialize our `TempReading` POJO.

8. The final thing to do is to define the **Topology** for our application. We want to keep it simple and just filter the input topic `temperatures-topic` for high temperatures (>25 degrees) and output the result to the output topic `high-temperatures-topic`. Add this code to the `getTopology()` function after the **TODO** comment:

```
builder.stream("temperatures-topic", Consumed.with(stringSerde,
temperatureSerde))
    .filter((key,value) -> value.temperature > 25)
    .to("high-temperatures-topic", Produced.with(stringSerde,
temperatureSerde));
return builder.build();
```

Note how the filter function uses the `value` which is an object of type `TempReading`.

9. Note the file `log4j.properties` in the folder `src/main/resources` which is used to configure logging for our application.

10. Use **Run** → **Start Debugging** to run your code. Let your Kafka Streams app remain running in the VS Code debugger.

You should get this output:

```
*** Starting JSON Sample Application ***
...
```

Creating input Data

11. Switch back to the terminal.

12. Use the following command to generate some temperature readings in JSON format:

```
$ cat << EOF | kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --property "parse.key=true" \
    --property "key.separator=:" \
    --topic temperatures-topic
"S1":{"station":"S1", "temperature": 10.2, "timestamp": 1}
"S1":{"station":"S1", "temperature": 11.2, "timestamp": 2}
"S1":{"station":"S1", "temperature": 11.1, "timestamp": 3}
"S1":{"station":"S1", "temperature": 12.5, "timestamp": 4}
"S2":{"station":"S2", "temperature": 15.2, "timestamp": 1}
"S2":{"station":"S2", "temperature": 21.7, "timestamp": 2}
"S2":{"station":"S2", "temperature": 25.1, "timestamp": 3}
"S2":{"station":"S2", "temperature": 27.8, "timestamp": 4}
EOF
```

Run this command repeatedly to generate more messages...

Reading the Output

13. Use the following command to read the output generated:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic high-temperatures-topic
```

You should get this output showing only readings with temperature higher than 25 degrees:

```
{"station":"S2","temperature":25.1,"timestamp":3}
{"station":"S2","temperature":27.8,"timestamp":4}
```

Cleanup

14. Terminate the Kafka console consumer by pressing `Ctrl+C`.

15. Terminate your **Kafka Streams** application with **Run** → **Stop Debugging**.

16. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

In this sample we have built a **Kafka Streams** application that uses custom serializer and deserializer to work with data that is JSON formatted.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04 Windowing & Aggregations

In this exercise you will create a Kafka Streams application to sessionize click data from a website. You will be organizing the user's click behaviour data collected from a website using sessions window.

a. Windowing & Aggregations

Prerequisites

Please make sure you have prepared your lab environment as described here: → Lab Environment

Writing the data to Kafka

1. Use the command in the table below to navigate to the project folder for your language:

```
cd ~/confluent-streams/labs/windowing
```

2. If your Kafka cluster is not already running, start it with:

```
$ docker-compose up -d zookeeper kafka schema-registry control-center
```

3. Create one input topic called `clicks-topic` and one output topic called `window-streams` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic clicks-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic window-streams
```

4. Start producing data to Kafka using the Java producer `clicks-producer`:

   a. Navigate to the producer folder:

   ```
   $ cd ./protobuf-java-producer
   ```

   b. Use `gradle` to generate the PROTOBUF class:

   ```
   $ ./gradlew build
   ```

   c. Run this command to start the Java producer:

   ```
   $ ./gradlew run
   ```

5. We have the clicks data in `clicks-topic`. Now, we are going to use **Kafka Streams** to do the same operation (use session windows to count the number of clicks for each IP address).

Using Session Windows in Kafka Streams

6. Open a new Terminal window and run the following command to open **count-streams-app** in Visual Studio Code:

```
$ code ~/confluent-streams/labs/windowing/count-streams-app
```

7. In VS Code, navigate to `src/main/java/io/confluent/training/app/` and open the Java file `StreamsApp.java`

8. Locate in the code the `TO-DO` lines and `{{ WRITE-MISSING-CODE }}` markers. Try to write yourself the missing code by checking the documentation:

   ◦ TO-DO 1 - documentation: create a KStream from the "clicks-topic" topic and configure the Key-Serde and Value-Serde that can read the String key, and Clicks value.

Solution

```
final KStream<String, ClicksProtos.Clicks> clicks =
builder.stream("clicks-topic", Consumed.with(Serdes.String(),
clicksSerde));
```

- TO-DO 2 - documentation: group by key the KStream "clicks".

Solution

```
final KGroupedStream<String, ClicksProtos.Clicks> clicksGrouped
= clicks.groupByKey();
```

- TO-DO 3 - documentation: apply a Session Window of 5 minutes with a Grace period of 30 seconds to the KGroupedStream "clicksGrouped" and apply a count to get the number of clicks per IP per Session Window.

Solution

```
final KTable<Windowed<String>, Long> clicksCount =
clicksGrouped
        .windowedBy(SessionWindows.with(Duration.ofMinutes(5)).g
race(Duration.ofSeconds(30)))
        .count();
```

- TO-DO 4 - documentation: convert the KTable "clicksCount" into a KStream.

Solution

```
final KStream<Windowed<String>, Long> clicksCountStream =
clicksCount.toStream();
```

- TO-DO 5 - documentation: produce the data of the KStream "clicksCountStreamModified" to the topic "window-streams" selecting the appropriate Serdes for key and value.

Solution

```
clicksCountStreamModified.to("window-streams",
Produced.with(Serdes.String(), clicksCountSerde));
```

9. After completing all the TO-DO's, start the Kafka Streams application by clicking in the top menu: **Run** → **Start Debugging**.
10. Let the application run for a few seconds and check the results in Confluent Control Center http://localhost:9021:
    - Go to **Topics** and select **window-streams** topic
    - Click on the tab **Messages**
    - In the box at the top where says `offset` with a magnifier, type `0`. A dropdown list will appear, then select `0/Partition: 0`
    - Wait for a few seconds and you will see the output from the Kafka Streams app
11. Check the results and try to answer the following questions:
    - How many different IP addresses are there?
    - How many different sessions are there?
    - Which IP has the highest number of clicks?

It might be easier to answer those questions when "Table View" is selected for the messages in top right corner in Control Center, as compared to the "Cards View".

Cleanup

12. Terminate your **Kafka Streams** application with **Run** → **Stop Debugging**.
13. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

In this exercise you created a **Kafka Streams** application to sessionize click data from a

website. A given user might visit a website multiple times a day, but in distinct visits; so using Session window you could automatically organize the data in sessions based on a period of inactivity.
You also learned how to implement time extractors in Kafka Streams.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 05 Joins

a. Joining Two Streams

In event-driven architecture, it is important to think about what event will trigger an output. Different kinds of joins will trigger outputs under different conditions. The purpose of this exercise is to experiment with the behavior of various stream-stream joins.

The streaming application itself performs a stream-stream join. It takes a string value from a "left stream" and a string value from a "right stream" and concatenates them together inside of brackets. The output is produced to a new stream. Remember that all stream-stream joins must be windowed since streams are unbounded. This application will use a tumbling window of 5 minutes.

Prerequisites

1. Please make sure you have prepared your lab environment as described here: →Lab Environment
2. If your Kafka cluster is not already running, start it with:

```
$ cd ~/confluent-streams/labs/joining-streams
$ docker-compose up -d zookeeper kafka control-center schema-registry
```

3. Create two input topics called `left-topic` and `right-topic` and an output topic called `joined-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic left-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic right-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic joined-topic
```

Remember that joins require the input topics to have the same number of partitions so that all input records with the same key, from both sides of the join, are delivered to the same stream task during processing. (called co-partitioning).

Create the Streaming App

4. Open another terminal and navigate to the `joining-streams` folder, and launch VS Code:

```
$ cd ~/confluent-streams/labs/joining-streams
$ code .
```

5. Open the file `build.gradle` and analyze its content.

6. Locate the file `JoinSample.java` in the folder `src/main/java/streams` and open it. Familiarize yourself with the code. It basically does the configuration of the **Kafka Streams** app but the interesting code is missing. If you would like to challenge yourself, take this time to create the streaming application logic yourself.

7. Now we define the **Topology** for our application. Add the following code snippet to the

`getTopology()` function after the **TODO** comment:

```
KStream<String, String> leftStream = builder.stream("left-topic",
    Consumed.with(stringSerde, stringSerde));
KStream<String, String> rightStream = builder.stream("right-topic",
    Consumed.with(stringSerde, stringSerde));
leftStream
    .join(rightStream,
        (leftValue, rightValue) -> "[" + leftValue + ", " +
rightValue + "]",
        JoinWindows.of(Duration.ofMinutes(5)),
        StreamJoined.with(stringSerde, stringSerde, stringSerde)
    )
    .to("joined-topic", Produced.with(stringSerde, stringSerde));

Topology topology = builder.build();
return topology;
```

What does the above code do? Discuss with your peers.

8. Note the file `log4j.properties` in the folder `src/main/resources`, which is used to configure logging for our streams application.

9. Use **Run** → **Start Debugging** to run your code. Let your Kafka Streams app remain running in the VS Code debugger.

   You should get this output:

```
*** Starting Join Sample Application ***
...
```

Ignore the WARNINGS.

Creating input Data

10. Open 3 terminal windows and arrange them side by side so you can see all three of them at the same time.

    We will be using the tool `kafkacat` to generate data and monitor the output:

11. In the first terminal window start the tool `kafkacat` as a producer for the `left-topic`

topic:

```
kafkacat \
-b kafka:9092 \
-t left-topic \
-P -K: -Z
```

12. In the second terminal window start `kafkacat` as a producer for the `right-topic` topic:

```
kafkacat \
-b kafka:9092 \
-t right-topic \
-P -K: -Z
```

13. In the third terminal window run an instance of `kafkacat` as a consumer of the `joined-topic` topic:

```
kafkacat \
-b kafka:9092 \
-t joined-topic \
-C -K\\t
```

14. In window 1 enter `FL:` (a record key of `FL` for Florida a `<NULL>` value for the record value) and observe the output in window 3. Hint: Nothing should happen. ...why?
15. In window 2 also enter the value `FL:` and observe the output in window 3. Hint: Nothing should happen. ...why?
16. Now in window 1 enter the value `FL:Orlando` and observe the output in window 3. Hint: Nothing should happen. ...why?
17. In window 2 enter the value `FL:Tampa` and observe the output in window 3. You should see:

```
FL  [Orlando, Tampa]
```

18. Back in window 1 enter `FL:` and observe the output in window 3. What do you see? Why?
19. Still in window 1 enter `FL:Miami` and observe the output in window 3. You should see:

```
FL  [Miami, Tampa]
```

20. Continue with window 2 and value `FL:Naples`. What do you see this time?

21. Continue to enter more values with the same key `FL`. Here are other cities in Florida to experiment with:

    - Jacksonville
    - Alachua
    - Pensacola
    - Destin
    - Fort Meyers

22. What happens if you use a different key, say `NY`? Why? Discuss the results with your peers.

23. What happens when an event falls outside of the tumbling window?

Left Join

24. Modify the `getTopology()` function and replace the `join` function with a `leftJoin` function instead.

25. Recompile and run the application.

26. This time, use a car brand for the key and different car models for values to play with input data as you have done for the (inner) join example.

    What is different? Discuss with your peers if needed.

Outer Join

27. Modify the `getTopology()` function and replace the current **join** function with an `outerJoin` function instead.

28. Recompile and run the application.

29. This time, use your home country for key and different cities for values to play with input data as you have done for the (inner) join example and observe the output.

    What is different? Discuss with your peers if needed.

Optional: Stream - Table Joins

30. Perform a similar experiment with a stream - table **left join** (the most common join in most streaming applications). Make sure to experiment with sending null keys and values. How are the results different from the stream - stream left join?

Cleanup

31. Terminate the first 2 instances of `kafkacat` (the producer instances) by pressing `Ctrl+D`.
32. Terminate the third instance of `kafkacat` (the consumer instance) by pressing `Ctrl+C`.
33. Terminate your **Kafka Streams** application with **Run** → **Stop Debugging**.
34. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

In this exercise you created a **Kafka Streams** application that joins two streams with inner, left, and outer joins. You then created data for the left and the right input stream and observed the generated output. You used the command line tool `kafkacat` to generate input data and observe output data. Consider summarizing your observations and comparing them to the information found here: https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#kstream-kstream-join. Especially focus on the subsection called "Semantics of stream-stream joins" with an illustrative table of the output records that are produced from a join as events flow into the left and right streams.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 06 Custom Processing

a. Using the Processor API

The purpose of this exercise is to create an application with the lower-level Processor API. This may be required in applications that require a greater level of control over state store management and more sophisticated application logic than the Kafka Streams DSL can provide.

This exercise will give you experience creating a Kafka Streams application using the DSL and creating a new node in the topology using the Processor API through the transform() method.

This application uses a simple source → word count processor → sink processing topology. The source node takes in records from an input topic whose values are sentence strings. The word count processor uses each record's value to update its internal state store for word counts (key=word, value=count) and sends that state to the sink processor every second. The sink processor produces the resulting records to an output topic.

Prerequisites

1. Please make sure you have prepared your lab environment as described here: →Lab Environment
2. If your Kafka cluster is not running already, start it with:

```
$ cd ~/confluent-streams/labs/processor-api
$ docker-compose up -d zookeeper kafka
```

3. Create two topics called `lines-topic`, and `word-count-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic word-count-topic
```

Create the Streaming App

4. Open another terminal and navigate to the `processor-api` folder, and launch VS Code:

```
$ cd ~/confluent-streams/labs/processor-api
$ code .
```

5. Open the file `build.gradle` in the `processor-api` folder and analyze its content. It should be quite familiar by now.

6. Open the file `WordCountTransformer.java` in subfolder `src/main/java/streams`.

   Familiarize yourself with the code. We are creating an instance of type `Transformer` and overriding its methods. Thoroughly document what the `init` method does. Pay particular note to the call to the context.schedule() method, with its punctuation and the call to context.forward(). As always, you can check the corresponding code in the `solutions` folder for feedback.

7. You can challenge yourself to write the `transform` method, or continue with the next step.

8. Make it so the `transform` method of the class has these contents:

```
Long oldValue = this.kvStore.get(word);
if (oldValue == null) {
   this.kvStore.put(word, 1L);
} else {
   this.kvStore.put(word, oldValue + 1L);
}
```

This code gets the correct entry from the keystore and updates it or it creates a new entry in the keystore with a key of the incoming word and a count of 1. It leaves it to the context scheduler to forward the entries in the key value store, producing them to Kafka for durability.

9. Open the file `CustomTransformerApp.java` in the same folder, and familiarize yourself with the code. This code is entirely DSL - It defines the topology, creates the configuration, sets up the shutdown hook and starts the **Kafka Streams** app.

In the topology, it creates a stream from the input topic, uses the flatMapValues() method to break up the lines of input text into individual words, writes a rekeyed stream out to a repartition topic and reads it back in. Then it calls the transform() method we created in the WordCountTransformer code. And finally, it directs those results to the output topic.

10. Notice the file `log4j.properties` in the folder `src/main/resources` used to configure logging for the application.
11. Use **Run** → **Start Debugging** to run your code. Let your Kafka Streams app remain running in the VS Code debugger.

    You should get this output:

    ```
    *** Starting Custom Transformer App Application ***
    ```

Creating input Data

12. Open 2 terminal windows and arrange them side by side so you can see the two at the same time.
13. In the first terminal window start the tool `kafkacat` as a producer for the `lines-topic` topic:

    ```
    kafkacat \
    -b kafka:9092 \
    -t lines-topic \
    -P -K :
    ```

14. In the second terminal window run an instance of `kafka-console-consumer` as a consumer of the `word-count-topic` topic, printing the String key and the Long value:

```
kafka-console-consumer --bootstrap-server kafka:9092 \
 --topic word-count-topic --from-beginning \
 --property print.key=true \
 --value-deserializer
org.apache.kafka.common.serialization.LongDeserializer
```

15. In window 1 enter each of these strings one at a time:

```
kafka:Kafka powers the Confluent streaming platform
kafka:A streaming application uses Kafka
kafka:Everyone loves Kafka
kafka:Many contributors to Kafka work for Confluent
```

    After each line observe the output in window 2.

16. Discuss the result with your peers.

Cleanup

17. Terminate the `kafkacat` instance by pressing `Ctrl+D`.
18. Terminate the `kafka-console-consumer` instance by pressing `Ctrl+C`.
19. Terminate your **Kafka Streams** application with **Run** → **Stop Debugging**.
20. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

In this sample we have demonstrated the use of a custom transform written using the Processor API to count words. This is included into a Stream processing application written entirely in the Streams DSL that takes in sentences from an input topic, process them and writes the resulting word and count pairs to an output topic.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07 Testing, Monitoring and Troubleshooting

This lab contains 2 exercises:

- Testing:
  - Integration Tests using Embedded Kafka
- Monitoring:
  - Using JConsole to monitor a Streams App

a. Integration Tests using Embedded Kafka

In this exercise, we are going a step further to use a full blown **embedded** Kafka single node cluster to test our **Kafka Streams** application. The application is a simple **word count** streaming app. As in the previous test, we do not need to run our Docker container Kafka cluster.

Prerequistes

Please make sure you have prepared your lab environment as described here: → Lab Environment

Creating the Artifacts to Test

1. Navigate to folder `~/confluent-streams/labs/testing/wordcount-test`, and launch VS Code:

```
$ cd ~/confluent-streams/labs/testing/wordcount-test
$ code .
```

2. Open the file `build.gradle` and analyze its content. Notice the many dependencies that we have to add to enable this kind of integration testing.

we have a few dependencies that use the **test** version of the respective `JAR` files. This can be achieved by adding `classifier: 'test'` or `classifier: 'tests'` to the dependency.

3. Open the file `TopologyProvider.java` class, located in subfolder `src/main/java/streams`, that defines the **Kafka Streams** topology that we will test. Analyze the code. Make sure you understand it. If not discuss it with your peers.

Writing the Integration Test

Now it is time to write the actual test that is leveraging the testbed that we have just prepared.

4. Locate the file `TopologyProviderTest.java` in subfolder `src/test/java/streams` and open it. It will host our test code.

   Let's analyze the code a bit:

   - In the `@BeforeClass` we're initializing and running an embedded single node Kafka cluster. We are preparing the cluster by creating the **input** and the **output** topic.
   - The method `shouldCountWords` contains our first test. We have clearly documented the steps:
     1. define the configuration of the streaming app to use during the test
     2. get the topology that we want to test
     3. initialize and start the streaming application
     4. produce some data for the **input** topic
     5. finally verify that the produced data in the **output** topic corresponds to the expected values

5. Inspect the method `getStreamsConfiguration`. Notice that we take the information about the bootstrap server(s) from our `CLUSTER` variable which represents our single node embedded Kafka cluster. We also provide a path to where the state store should store its information (`STATE_DIR_CONFIG`).

6. Inspect the method `produceInputData`. Notice that we're configuring a producer and

are using it to feed the data into the **input** topic.

7. The final step is to write code that validates the results. Inspect the `verifyOutputData` method. You can challenge yourself to complete the method, or you can proceed to the next step.

8. Make it so the `verifyOutputData` method has these contents:

```java
List<KeyValue<String, Long>> expectedWordCounts = Arrays.asList(
    new KeyValue<>("hello", 1L),
    new KeyValue<>("all", 1L),
    new KeyValue<>("streams", 2L),
    new KeyValue<>("lead", 1L),
    new KeyValue<>("to", 1L),
    new KeyValue<>("join", 1L),
    new KeyValue<>("kafka", 3L),
    new KeyValue<>("summit", 1L)
);

Properties consumerConfig = new Properties();
consumerConfig.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, CLUSTER
.bootstrapServers());
consumerConfig.put(ConsumerConfig.GROUP_ID_CONFIG,
    "wordcount-lambda-integration-test-standard-consumer");
consumerConfig.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest");
consumerConfig.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class);
consumerConfig.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    LongDeserializer.class);

List<KeyValue<String, Long>> actualWordCounts =
    IntegrationTestUtils.waitUntilMinKeyValueRecordsReceived(
        consumerConfig, outputTopic, expectedWordCounts.size());
assertThat(actualWordCounts, containsInAnyOrder(expectedWordCounts
.toArray()));
```

We define an array with the expected data, define a consumer configuration, and then compare the produced data to the expected data.

Running the Test(s)

9. In the terminal window:

```
$ ./gradlew test
```

You should see something like this (shortened):

```
...
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test

BUILD SUCCESSFUL in 46s
3 actionable tasks: 3 executed
...
```

10. Navigate to the `wordcount-test/build/reports/tests/test/` folder and open the `index.html` test report in a browser.

Cleanup

There is no special cleanup needed.

Conclusion

In this exercise we have written an integration test for a Kafka Streams app topology. The test bed uses an embedded version of Kafka and Zookeeper.

STOP HERE. THIS IS THE END OF THE EXERCISE.

b. Using JConsole to monitor a Streams App

In this exercise, we will use `JConsole` to monitor the various metrics a simple **Kafka Streams** application exposes.

We will be using an application we created in an earlier lab that reads data from a topic whose keys are integers and whose values are sentence strings. The input values are transformed to lower-case and output to a new topic.

Prerequisites

Please make sure you have prepared your lab environment as described here: → Lab Environment

Preparing the Lab

1. Navigate to the module folder:

```
$ cd ~/confluent-streams/labs/monitoring
```

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka
```

3. Create an input topic called `lines-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-topic
```

4. Create the output topic called `lines-lower-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-lower-topic
```

5. In a terminal window navigate to the project folder `jmx-sample` and build the artifact:

```
$ cd ~/confluent-streams/labs/monitoring/jmx-sample
$ ./gradlew build
```

6. Examine `build.gradle` and observe it has been configured to expose JMX metrics on port `4444` that we can then use to attach `JConsole`:

```
$ cat build.gradle
...
applicationDefaultJvmArgs = [
"-Dcom.sun.management.jmxremote",
"-Dcom.sun.management.jmxremote.authenticate=false",
"-Dcom.sun.management.jmxremote.ssl=false",
"-Djava.rmi.server.hostname=127.0.0.1",
"-Dcom.sun.management.jmxremote.rmi.port=4444",
"-Dcom.sun.management.jmxremote.port=4444"]
...
```

7. Still in the project folder `jmx-sample` run your **Kafka Streams** application:

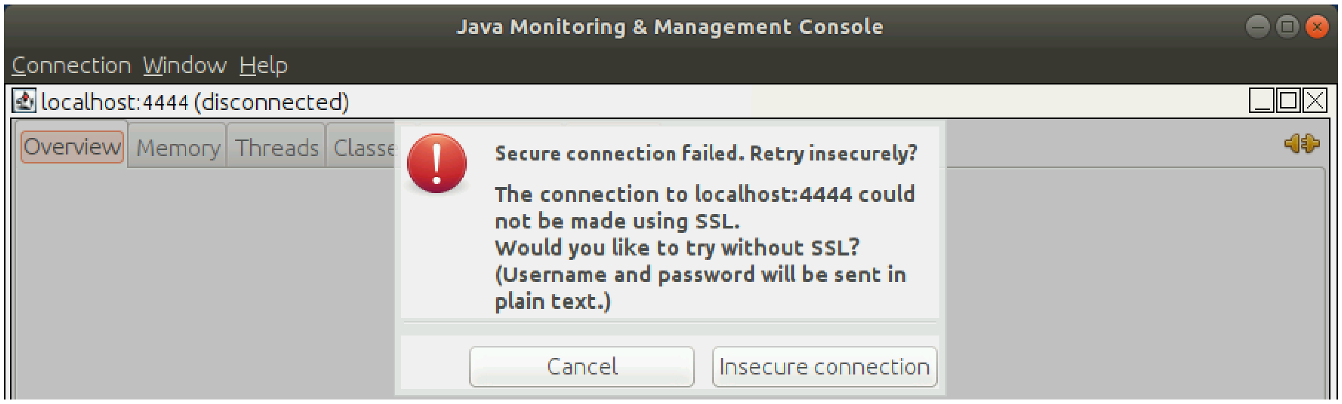
```
$ ./gradlew run
```

8. Observe the JMX metrics of the **Kafka Streams** application:

   a. Open a new Terminal window and open a `jconsole` connection to port `4444` which is the JMX port for the **Kafka Streams** application.

   ```
   $ jconsole localhost:4444 &
   ```

   b. Select **Insecure connection** when asked

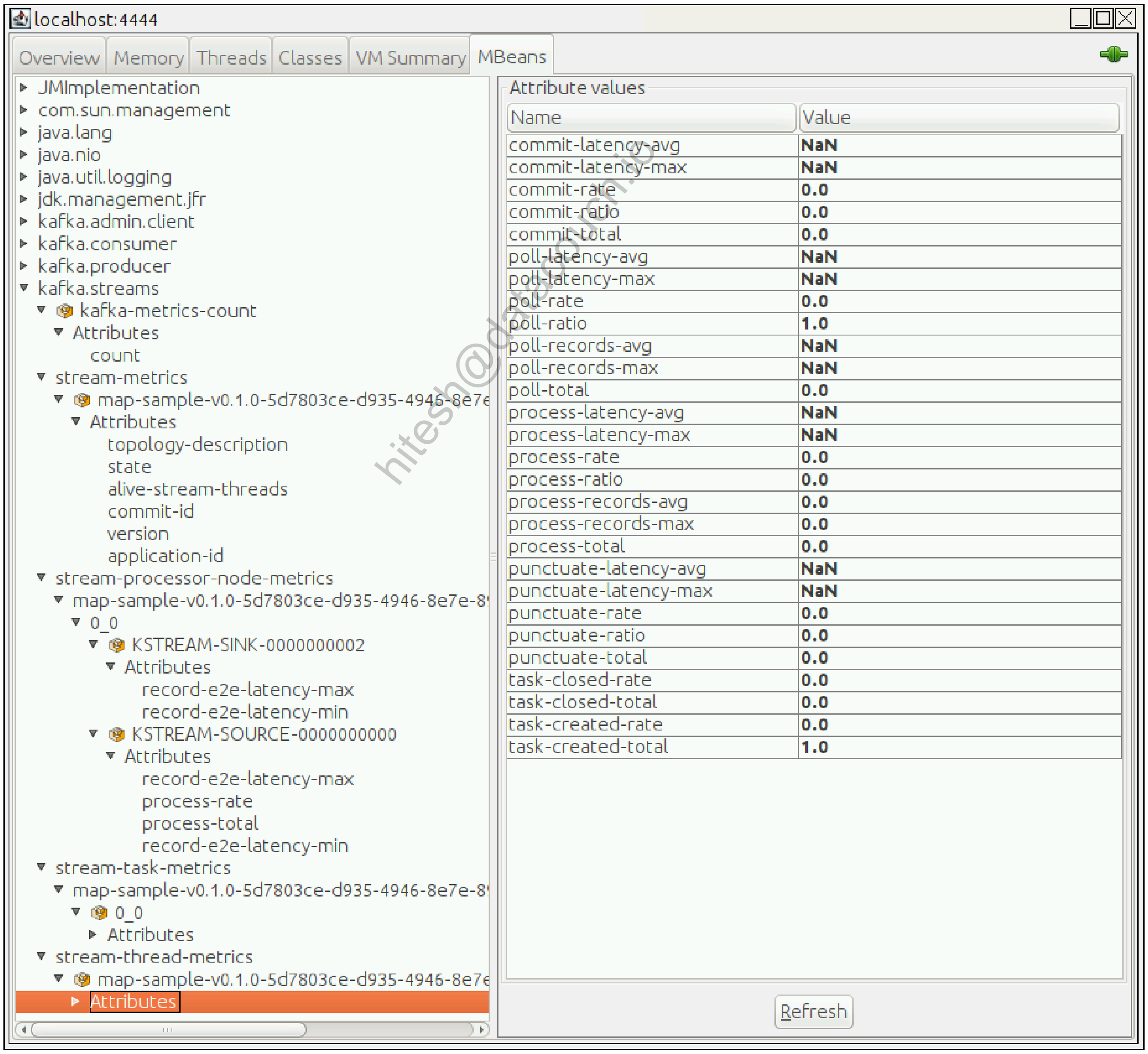

Java Monitoring & Management Console
Connection Window Help
localhost:4444 (disconnected)
Overview Memory Threads Classe
Cancel Insecure connection

Secure connection failed. Retry insecurely?

The connection to localhost:4444 could not be made using SSL.
Would you like to try without SSL?
(Username and password will be sent in plain text.)

c. Navigate to the **MBeans** tab

d. Explore the node under **kafka-streams** as indicated in the below image:



```
▸ JMImplementation
▸ com.sun.management
▸ java.lang
▸ java.nio
▸ java.util.logging
▸ jdk.management.jfr
▸ kafka.admin.client
▸ kafka.consumer
▸ kafka.producer
▾ kafka.streams
  ▾ kafka-metrics-count
    ▾ Attributes
        count
  ▾ stream-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e
      ▾ Attributes
          topology-description
          state
          alive-stream-threads
          commit-id
          version
          application-id
  ▾ stream-processor-node-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e-8
      ▾ 0_0
        ▾ KSTREAM-SINK-0000000002
          ▾ Attributes
              record-e2e-latency-max
              record-e2e-latency-min
        ▾ KSTREAM-SOURCE-0000000000
          ▾ Attributes
              record-e2e-latency-max
              process-rate
              process-total
              record-e2e-latency-min
  ▾ stream-task-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e-8
      ▾ 0_0
        ▸ Attributes
  ▾ stream-thread-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e
      ▸ Attributes
```

Name	Value
commit-latency-avg	NaN
commit-latency-max	NaN
commit-rate	0.0
commit-ratio	0.0
commit-total	0.0
poll-latency-avg	NaN
poll-latency-max	NaN
poll-rate	0.0
poll-ratio	1.0
poll-records-avg	NaN
poll-records-max	NaN
poll-total	0.0
process-latency-avg	NaN
process-latency-max	NaN
process-rate	0.0
process-ratio	0.0
process-records-avg	0.0
process-records-max	0.0
process-total	0.0
punctuate-latency-avg	NaN
punctuate-latency-max	NaN
punctuate-rate	0.0
punctuate-ratio	0.0
punctuate-total	0.0
task-closed-rate	0.0
task-closed-total	0.0
task-created-rate	0.0
task-created-total	1.0

Initially some of the numbers, such as `process total` will be zero.

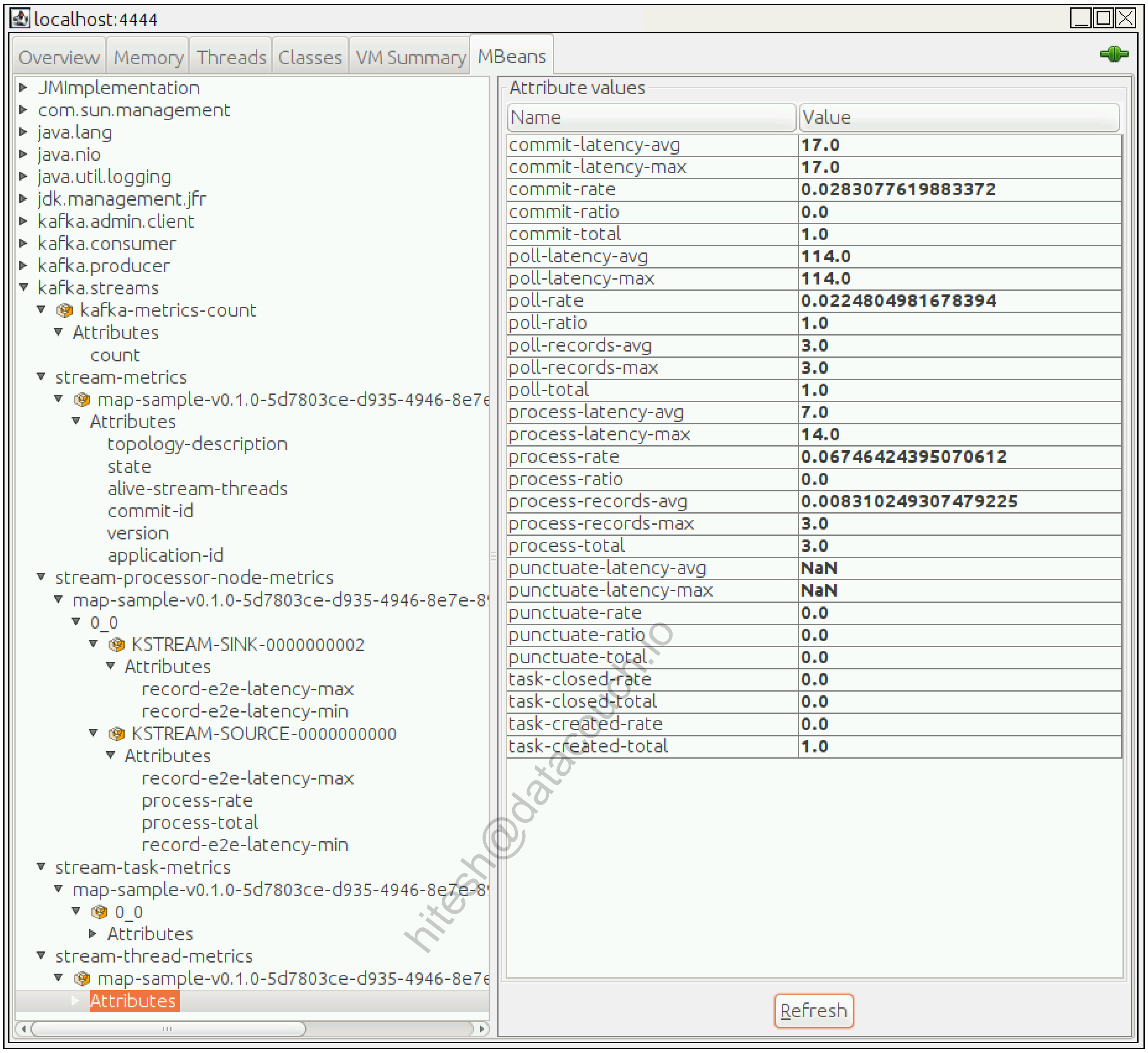
9. Open a new terminal window and navigate to the `monitoring` folder:

```
$ cd ~/confluent-streams/labs/monitoring
```

10. Now create some data that will be consumed and processed by the **Kafka Streams** application:

```
$ cat << EOF | kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --property "parse.key=true" \
    --property "key.separator=:" \
    --topic lines-topic
1:"Kafka powers the Confluent Streaming Platform"
2:"Events are stored in Kafka"
3:"Confluent contributes to Kafka"
EOF
```

11. Observe how the values of the metrics in **JConsole** change (you will need to click the **Refresh** button):



```
▸ JMImplementation
▸ com.sun.management
▸ java.lang
▸ java.nio
▸ java.util.logging
▸ jdk.management.jfr
▸ kafka.admin.client
▸ kafka.consumer
▸ kafka.producer
▾ kafka.streams
  ▾ kafka-metrics-count
    ▾ Attributes
        count
  ▾ stream-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e
      ▾ Attributes
          topology-description
          state
          alive-stream-threads
          commit-id
          version
          application-id
  ▾ stream-processor-node-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e-8
      ▾ 0_0
        ▾ KSTREAM-SINK-0000000002
          ▾ Attributes
              record-e2e-latency-max
              record-e2e-latency-min
        ▾ KSTREAM-SOURCE-0000000000
          ▾ Attributes
              record-e2e-latency-max
              process-rate
              process-total
              record-e2e-latency-min
  ▾ stream-task-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e-8
      ▾ 0_0
        ▸ Attributes
  ▾ stream-thread-metrics
    ▾ map-sample-v0.1.0-5d7803ce-d935-4946-8e7e
      ▸ Attributes
```

Attribute values

Name	Value
commit-latency-avg	17.0
commit-latency-max	17.0
commit-rate	0.028307761988 3372
commit-ratio	0.0
commit-total	1.0
poll-latency-avg	114.0
poll-latency-max	114.0
poll-rate	0.0224804981678394
poll-ratio	1.0
poll-records-avg	3.0
poll-records-max	3.0
poll-total	1.0
process-latency-avg	7.0
process-latency-max	14.0
process-rate	0.06746424395070612
process-ratio	0.0
process-records-avg	0.008310249307479225
process-records-max	3.0
process-total	3.0
punctuate-latency-avg	NaN
punctuate-latency-max	NaN
punctuate-rate	0.0
punctuate-ratio	0.0
punctuate-total	0.0
task-closed-rate	0.0
task-closed-total	0.0
task-created-rate	0.0
task-created-total	1.0

12. **Optional:** add more data to the topic and monitor the attributes and how the values change.

Cleanup

13. Quit **JConsole**.

14. Stop the sample Kafka Streams application by pressing `Ctrl+C`.

15. Shut down your Kafka cluster with the `docker-compose down -v` command.

Conclusion

We have used Java code to retrieve the metrics directly from the `KafkaStreams` object and also configured the **Kafka Streams** sample application to expose JMX data that we then explored using `JConsole`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 09 Security

a. Securing a Kafka Streams Application

In this exercise, we will interact with a secure Kafka cluster in several ways. The cluster is configured to use SASL-PLAIN for authentication and SSL for transport encryption. A certificate creation script is used to create keystores and truststores for clients and brokers so that they can authenticate with each other. There is a client `.properties` that will be used to create topics, to produce to an input topic, and to consume from an output topic. There is another `.properties` file to configure the security for a Kafka Streams application. In practice, SSL certificates must be carefully managed, credentials should be created separately for different clients, and permissions should be applied to secure credentials. These areas are beyond the scope of the course. Rather, the purpose of the exercise is to provide hands-on experience with the configurations necessary to connect a Kafka Streams application to a secured cluster.

Preparing the Project

Please make sure you have prepared your lab environment as described here: → Lab Environment

⚠️ If you have a cluster running from an earlier lab, shut it down now. This lab depends on the settings in the local docker-compose.yml file. You can check if you have a cluster running with `docker-compose ps` and you can shutdown a running cluster with `docker-compose down -v`.

Preparing the Certificates

First we need to generate all the necessary certificates and credentials that are used to create a secure (single node) Kafka cluster.

1. In a terminal window, navigate to the folder `security/secure-app/scripts/security`:

```
$ cd ~/confluent-streams/labs/security/secure-app/scripts/security
```

2. Take a few minutes to inspect the contents of the files in this directory and discuss them with your peers.

3. Run the `certs-create.sh` script to generate the necessary certs and credential files:

```
$ ./certs-create.sh
```

Running the Cluster

4. In a terminal window navigate to folder `security/secure-app`, and launch VS Code:

```
$ cd ~/confluent-streams/labs/security/secure-app
$ code .
```

5. From this folder open the file `docker-compose.yml` and analyze it. Specifically focus on the settings of the security relevant environment variables. Discuss the content with your peers if you don't fully understand it.

6. Start the cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

and wait a couple of minutes until the cluster is initialized. You may observe the progress by using:

```
$ docker-compose ps
```

or following the logs, e.g.:

```
$ docker-compose logs -f kafka
```

Press `Ctrl+C` to stop following the log.

Creating the Application

7. In the `secure-app` folder, locate the file `build.gradle` and analyze its content. There should be no surprises at this point.

8. Open the file `SecureAppSample.java` located in the subfolder `src/main/java/streams` and inspect its contents. As you can see, we're using a class `ConfigProvider` to get us the configuration for the **Kafka Streams** app and a class `TopologyProvider` to get us the topology for the application. Other than that all is well known and familiar code by now.

9. Now let's have a look at the file `ConfigProvider.java`. Notice that this code gets the configuration settings from a `.properties` file, which is preferable to hard-coding properties into our application.

10. In the project sub-folder `scripts/security`, open the file called `secureapp-sample.properties` analyze its content.

    - Note that we're using **SASL** to authenticate our application with the secured Kafka cluster.

    - The selection of the **protocol** (SASL_SSL) also has an implication on which port we're using for the communication with the brokers; `9091` in this case. The brokers are also listening to `SSL` on port `11091`, if we wanted to do mutual SSL instead of SASL + SSL. In that case, we would also have to configure `ssl.keystore` location and password on the clients so that Brokers could authenticate them.

    - We need to define the **trust store** and its password as well as the login module and the credentials.

    - We're also securing the monitoring interceptors so that the app can be monitored from **Confluent Control Center**.

11. Copy the security files to the location in `/etc/kafka/secrets/` - remember our user ID is training and the password which will be requested is also training:

```
$ sudo mkdir -p /etc/kafka/secrets/
$ sudo cp scripts/security/* /etc/kafka/secrets/
```

12. Finally have a look at the file `TopologyProvider.java` which should have this content:

```
package streams;

import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.StreamsBuilder;

public class TopologyProvider {
    public Topology getTopology() {
        final StreamsBuilder builder = new StreamsBuilder();
        builder.stream("secure-input").to("secure-output");
        return builder.build();
    }
}
```

Since this sample is all about securing a **Kafka Streams** application we have chosen a very simple **Topology**, it's basically a `NO-OP` topology. We read input from topic `secure-input` and directly and unmodified output all the messages to the topic `secure-output`.

13. Before we can run our **Kafka Streams** sample application we need to first manually create the topics `secure-input` and `secure-output` since we have configured the Kafka cluster to **disable** auto-create of topics (see setting in the `docker-compose.yml` file). The `AdminClient` that creates topics must also authenticate with the Kafka cluster, so we must use the `--command-config` option to point to the `.properties` file that contains the security configurations.

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9091 \
    --replication-factor 1 \
    --partitions 3 \
    --topic secure-input \
    --command-config /etc/kafka/secrets/client_security.properties

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9091 \
    --replication-factor 1 \
    --partitions 3 \
    --topic secure-output \
    --command-config /etc/kafka/secrets/client_security.properties
```

You will see some warnings such as

```
2018-10-15 18:12:39 WARN  ProducerConfig:287 - The
configuration
'confluent.monitoring.interceptor.security.protocol' was
supplied but isn't a known config.
```

You can safely ignore these warnings. The properties are necessary monitoring producers and consumers in Confluent Control Center, but are not recognized by the core open source Apache Kafka `ProducerConfig` and the `ConsumerConfig` classes.

14. Now we're ready to run the app. Use **Run** → **Start Debugging** to run your **Kafka Streams** application.

Ignore the WARNINGS.

Creating Input Data

To see the sample streams application working we need to generate some data.

15. In the `~/confluent-streams/labs/security/secure-app/scripts/security` folder, there is a file called `client_security.properties` with the following content:

```
bootstrap.servers=kafka:9091
security.protocol=SASL_SSL
ssl.truststore.location=/etc/kafka/secrets/kafka.client.truststore.jk
s
ssl.truststore.password=confluent
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginMod
ule required \
    username=\"client\" \
    password=\"client-secret\";

# authenticate the monitor interceptor with Kafka.
confluent.monitoring.interceptor.bootstrap.servers=kafka:9091
confluent.monitoring.interceptor.security.protocol=SASL_SSL
confluent.monitoring.interceptor.ssl.truststore.location=/etc/kafka/s
ecrets/kafka.client.truststore.jks
confluent.monitoring.interceptor.ssl.truststore.password=confluent
confluent.monitoring.interceptor.sasl.mechanism=PLAIN
confluent.monitoring.interceptor.sasl.jaas.config=\
    org.apache.kafka.common.security.plain.PlainLoginModule required
\
    username=\"client\" \
    password=\"client-secret\";
```

In an earlier step, we copied this file, along with others, to `/etc/kafka/secrets/`. This file will be used by the client tools that we're going to use in a moment to authenticate themselves, as it was earlier when we created the secure-input and secure-output topics.

16. Open a new terminal window, navigate to the `secure-app` folder:

```
$ cd ~/confluent-streams/labs/security/secure-app
```

17. To run the `kafka-console-producer` that we will use to feed some data to the topic `secure-input` use this command:

```
$ NS=io.confluent.monitoring.clients.interceptor && \
kafka-console-producer \
    --bootstrap-server kafka:9091 \
    --topic secure-input \
    --producer.config /etc/kafka/secrets/client_security.properties \
    --producer-property \
    interceptor.classes="${NS}.MonitoringProducerInterceptor"
```

- We're configuring the producer for monitoring via Control Center through the parameter `--producer-property`
- We're passing the **properties** file with the security settings through the parameter `--producer.config` to the producer.

18. Open another terminal window, navigate to the `secure-app` folder:

```
$ cd ~/confluent-streams/labs/security/secure-app
```

19. To list the data produced by our sample application to the topic `secure-output` use this command:

```
$ NS=io.confluent.monitoring.clients.interceptor && \
kafka-console-consumer \
    --group secure-console-consumer \
    --bootstrap-server kafka:9091 \
    --topic secure-output \
    --from-beginning \
    --consumer.config /etc/kafka/secrets/client_security.properties \
    --consumer-property \
    interceptor.classes="${NS}.MonitoringConsumerInterceptor"
```

- We're configuring the consumer for monitoring via Control Center through the parameter `--consumer-property`
- We're passing the **properties** file with the security settings through the parameter `--consumer.config` to the consumer

20. In the terminal where the producer is running enter a few lines such as:

```
Kafka is powering the Confluent streaming platform
Real time streaming is exceedingly important
Confluent offers support for Kafka Streams
In my company we build Kafka Streams applications
```

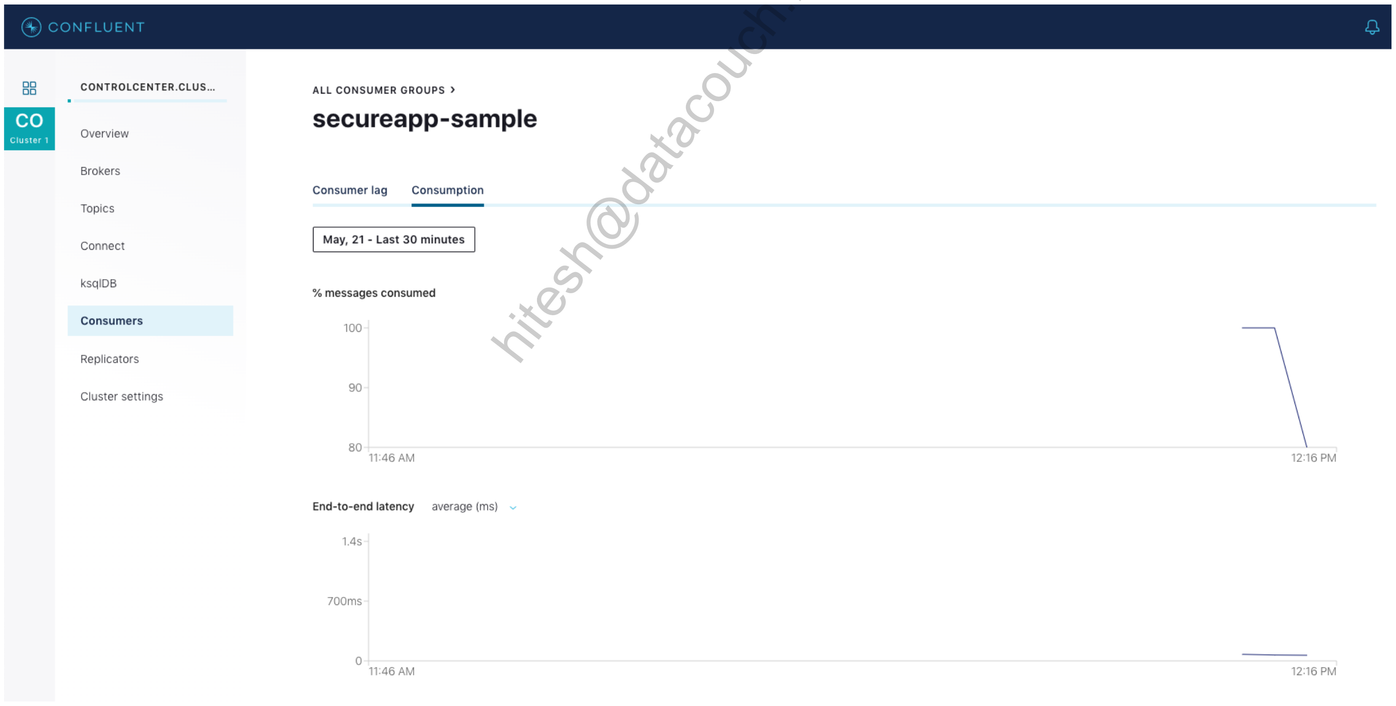
21. Observe how output is generated in the terminal window where the `kafka-console-consumer` is running. Specifically notice that the order of the output relative to the input might be changed (if you enter values fast enough) due to the fact that we have 3

partitions per topic and ordering is only guaranteed within a partition but not globally per topic!

22. **Optional:** Add a lot more data using the `kafka-console-producer`. For example:

```
$ NS=io.confluent.monitoring.clients.interceptor
$ for i in {1 .. 100}; do
        sleep 1
        seq 1000 | kafka-console-producer \
    --bootstrap-server kafka:9091 \
    --topic secure-input \
    --producer.config /etc/kafka/secrets/client_security.properties \
    --producer-property \
    interceptor.classes="${NS}.MonitoringProducerInterceptor"
done
```

23. Open **Confluent Control Center** at http://localhost:9021 and go to **Consumers → secureapp-sample** and you should see something like this:



Cleanup

24. Quit both the producer and consumer by pressing `Ctrl+C`.

25. Quit the sample **Kafka Streams** application with **Run** → **Stop Debugging**.

26. Shutdown the Kafka cluster:

```
$ docker-compose down -v
```

Conclusion

In this example we have shown how to run a secure Kafka cluster and then build a **Kafka Streams** application that integrates with this cluster using **SASL** for authentication and SSL for encryption. The application logic was trivial yet that is not the point of this example. The important fact is the integration with the Kafka cluster security settings.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix A: Additional Kafka Streams Testing & Monitoring Labs

This lab contains 3 extra exercises about Testing and Monitoring:

- Testing:
    - Building Unit Tests
- Monitoring:
    - Getting Metrics from a Kafka Streams Application
    - Monitoring a Kafka Streams App in Confluent Control Center

a. Building Unit Tests

It is essential for every application to have full test coverage of each of its components. The purpose of this exercise is to build unit tests for an existing Kafka Streams application and test the application with Gradle.

Prerequisites

1. Please make sure you have prepared your lab environment as described here: → Lab Environment
2. Notice that we do not need to run our Kafka cluster - the unit and integration testing do not require it.
3. Navigate to the folder `~/confluent-streams/labs/testing/simple-test`, and launch VS Code:

```
$ cd ~/confluent-streams/labs/testing/simple-test
$ code .
```

4. Open the file `build.gradle` inside this folder and analyze its content. Notice the `junit`, as well as the `kafka-streams-test-utils` dependencies that we use to enable testing.

Authoring the Processor

5. Have a look at the content of subfolder `src/main/java/streams`. You should find 3 Java files in it:
   - `CustomMaxAggregatorSupplier.java`
   - `ConfigProvider.java`
   - `TopologyProvider.java`
6. Open the class `CustomMaxAggregatorSupplier` and have a look into the code and try to understand what's happening. This is the code we're going to test ultimately. Maybe discuss the code with your peers.
7. Now a **Kafka Streams** application also needs some configuration. For this purpose we have the `ConfigProvider` class. Once again make sure you understand the code before you proceed.
8. Finally we have a `TopologyProvider` class which defines the topology of the **Kafka Streams** app that we want to test. And again we invite you to analyze the code and discuss it with your peers if needed.
9. Due to the fact that we use this topology in a test scenario it has some settings that would not be recommended in production. Can you spot them? If yes, discuss how we could improve this class to work well for both scenarios, testing and production run.
10. Now we're ready for the actual test class. Open the class `ProcessorTest.java` located in subfolder `src/test/java/streams`. This file contains the skeleton of the test class.

    Let's discuss the code:

    - We are using the `TopologyTestDriver` class to test the topology. This is a helper class from the `kafka-streams-test-utils` library.
    - To send records to the test driver we are using the `TestInputTopic` class.
    - To verify the results of our application we are using the `TestOutputTopic` class.
    - In the `setup()` method, we're using our two classes `ConfigProvider` and `TopologyProvider` to get the configuration and topology of our **Kafka Streams** application.
    - With the latter two we create a test driver instance that we will use in our tests

- The processor is stateful and we pre-populate the state store with a value of `21` for the key `a`.
- In the tear down method we simply clean up by closing the test driver instance.

Writing a Test

Now we are ready to write our first test.

> You may want to examine the Javadocs for the following classes we will be using:
>
> **TopologyTestDriver:**
> https://kafka.apache.org/25/javadoc/org/apache/kafka/streams/TopologyTestDriver.html
>
> **TestInputTopic:**
> https://kafka.apache.org/25/javadoc/org/apache/kafka/streams/TestInputTopic.html
>
> **TestOutputTopic:**
> https://kafka.apache.org/25/javadoc/org/apache/kafka/streams/TestOutputTopic.html

11. The first test method to the `ProcessorTest` class will test whether reading from the `result-topic` will give the current max value of `21` for the key `a` after inputting a smaller value for the same key. If so, this assures us that the first input flushed to the local state store and the result was produced to the output topic. You can choose to implement this test yourself by researching the `TopologyTestDriver`, `TestInputTopic`, and `TestOutputTopic` classes as well as the org.hamcrest.MatcherAssert and org.hamcrest.CoreMatchers classes before moving on.

```
@Test
public void shouldFlushStoreForFirstInput() {
    // TODO: add test code here
}
```

So far this is just standard `jUnit` test method.

12. Add code inside the above method to create an input record using the `pipeInput` method of the `TestInputTopic` class:

```
    inputTopic.pipeInput("a", 1L);
```

13. We can use the `readKeyValue` method of the `TestOutputTopic` class to read the output record generated by the processor and the `assertThat` method to compare the key value with the expected result

```
assertThat(outputTopic.readKeyValue(), equalTo(new KeyValue<>("a",
21L)));
```

14. Finally we make sure that this was the only result that we got as output for the given input:

```
assertThat(outputTopic.isEmpty(), is(true));
```

15. The final test method should look like this:

```
@Test
public void shouldFlushStoreForFirstInput() {
    // TODO: add test code here
    inputTopic.pipeInput("a", 1L);
    assertThat(outputTopic.readKeyValue(), equalTo(new KeyValue<>(
"a", 21L)));
    assertThat(outputTopic.isEmpty(), is(true));
}
```

And that's it! We have just authored a complete test.

Running the Test(s) & Displaying the Test Report

16. In the terminal window:

```
$ ./gradlew test
```

The output of this command should look similar to this:

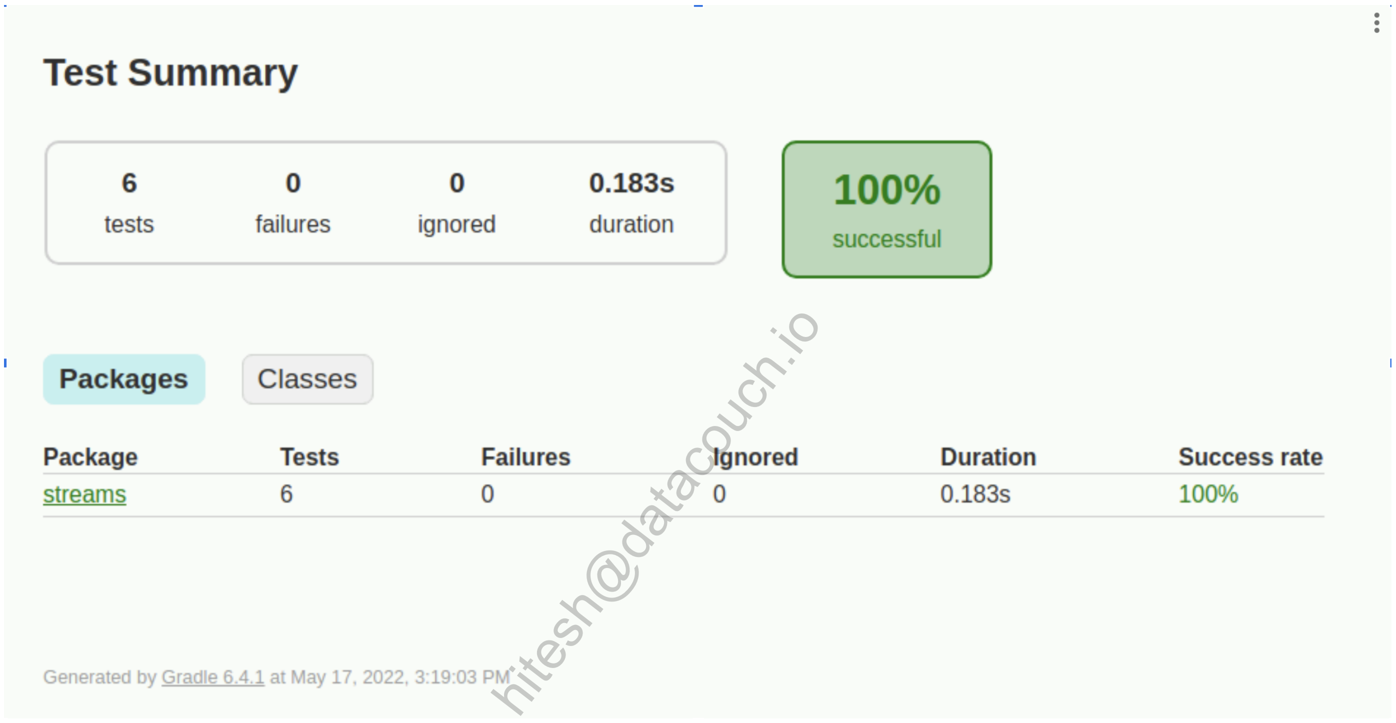
```
BUILD SUCCESSFUL in 4s
3 actionable tasks: 2 executed, 1 up-to-date
```

As we can see, the code compiled and the tests ran without a problem.

17. Navigate to the `simple-test/build/reports/tests/test/` folder and open the `index.html` test report in a browser.

> Open from file explorer providing the full path `confluent-streams/labs/testing/simple-test/build/reports/tests/test/`

it should look similar to this:



Test Summary

6	0	0	0.183s	100%
tests	failures	ignored	duration	successful

Packages | Classes

Package	Tests	Failures	Ignored	Duration	Success rate
streams	6	0	0	0.183s	100%

Generated by Gradle 6.4.1 at May 17, 2022, 3:19:03 PM

At this point, we should see 1 test that ran successfully.

Adding more Tests

18. Provided is a second test that makes sure that the store value is not updated for smaller input values

```
@Test
public void shouldNotUpdateStoreForSmallerValue() {
    inputTopic.pipeInput("a", 1L);
    assertThat(store.get("a"), equalTo(21L));
    assertThat(outputTopic.readKeyValue(), equalTo(new KeyValue<>(
"a", 21L)));
    assertThat(outputTopic.isEmpty(), is(true));
}
```

Discuss with your peers what this test does.

19. Add more tests and repeat the `./gradlew test` command to see new results.

    a. Add a test that asserts that the store value is updated for a larger input value.

    > If you get stuck, please have a look in the solutions folder where you will find the complete sample solution.

    b. Add a test that asserts a new store value is generated if adding a value with a new key "b". Also make sure the existing store value for "a" is unchanged.

    c. Write a test that verifies that the processor **punctuates** if the **event time** advances.

    d. Write a test that verifies that the processor **punctuates** if the **wall clock time** advances.

    > Use the function `testDriver.advanceWallClockTime(…)` for this.

Conclusion

In this sample we have shown how to test a simple **Kafka Streams** application that uses the Processor API.

b. Getting Metrics from a Kafka Streams Application

The purpose of this exercise is to learn how to expose metrics in a Kafka Streams application. This particular application sends metrics to standard output every 10 seconds. In practice, these metrics can be exposed to external sources for aggregation and analysis, as we will see in the exercises that follow.

Preparing the application

In this exercise we're going to use the **word count** exercise from a previous lab.

1. Please make sure you have prepared your lab environment as described here: → Lab Environment

2. Navigate to the folder for this lab:

```
$ cd ~/confluent-streams/labs/monitoring
```

3. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

4. Create the two topics called `lines-topic` and `word-count-topic` in Kafka:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic word-count-topic
```

Building & Running the Application

5. In a terminal window navigate to the `word-count` folder, and launch VS Code:

```
$ cd ~/confluent-streams/labs/monitoring/word-count
$ code .
```

6. Open the file `build.gradle` in folder `word-count` and analyze its content. It should be quite familiar by now.

7. Notice the four Java files in subfolder `src/main/java/streams`:

`ConfigProvider.java`	Defines the configuration for the streams application
`MetricsReporter.java`	Defines how the list of metrics is output every 10 seconds by the app
`TopologyProvider.java`	Defines the topology of the stream application
`WordCountSample.java`	Main class of the application. This class makes use of the 3 following classes

Analyze their code. Specifically note the use of the `MetricsReporter` class. Make sure you understand what's going on in the code.

8. Use **Run → Start Debugging** in VS Code or `./gradlew run` in the terminal to run your code.

The application will print out the list of metrics to the terminal every 10 seconds. It should look similar to this (shortened for readability):

```
--- Application Metrics ---
MetricName [name=count, group=kafka-metrics-count, description=total
number of registered metrics, tags={client-id=wordCount-3e02d8d9-
490b-492a-9bf0-cf85629e7fcf-StreamThread-1-producer}], 81.0
MetricName [name=io-time-ns-avg, group=producer-metrics,
description=The average length of time for I/O per select call in
nanoseconds., tags={client-id=wordCount-3e02d8d9-490b-492a-9bf0-
cf85629e7fcf-StreamThread-1-producer}], 612250.0
```

Producing Input Data

To see how the metrics change, we can produce some input data that the application will process. We use the `kafka-console-producer` tool for this job.

9. Return to the terminal window and create a list of input sentences that will be randomly produced to the input topic:

```
$ INPUTS=('Kafka powers the Confluent streaming platform' \
    'All Streams come from Kafka'\
    'Streams will all flow to Kafka'\
    'Follow the streams to Kafka Summit' \
    'Check out Confluent Cloud')
```

10. Run Kafkacat to produce a steady flow of sentences to the input topic:

```
$ while true; do
     MESSAGE=${INPUTS[${RANDOM} % ${#INPUTS[@]}]}
     echo ${MESSAGE} | kafkacat -P \
         -b kafka:9092 \
         -t lines-topic
     sleep 0.1
  done
```

11. Observe the metric values printed by the Word Count application.

Use `grep` to track a metric. For example, run: `./gradlew run | grep name=poll-records-avg` to track the metric **poll-records-avg** and then start/stop the kafkacat producer of Step 10.

12. In another terminal window run `kafka-console-consumer` to report the output of our

sample app:

```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
    --topic word-count-topic \
    --property print.key=true \
    --value-deserializer
org.apache.kafka.common.serialization.LongDeserializer
```

Cleanup

13. Quit the Producer with `Ctrl+C`.

14. Quit the Consumer with `Ctrl+C`.

15. Quit our sample app with **Run** → **Stop Debugging**.

16. Shut down your Kafka cluster with the `docker-compose down -v` command.

c. Monitoring a Kafka Streams App in Confluent Control Center

In this exercise we're going to reuse the **word count** processor API example application from a previous exercise and extend it so that it can be monitored in **Confluent Control Center**.

Prerequisites

1. Please make sure you have prepared your lab environment as described here: → Lab Environment
2. In a terminal window navigate to the `processor-sample` folder:

```
$ cd ~/confluent-streams/labs/monitoring/processor-sample
```

3. Open this application's root directory in VS Code.

```
$ code .
```

4. Open the `build.gradle` file and observe `monitoring-interceptors` in the list of dependencies:

```
compile group: "io.confluent", name: "monitoring-interceptors",
version: "6.0.0"
```

   This dependency contains the interceptors classes that we will use to configure our application for monitoring via **Confluent Control Center**.

5. Open the file `ProcessorSample.java` (in folder `src/main/java/streams`) which contains the `main` function of the sample application. Notice these lines that have been added to the `getConfig` function:

```
settings.put(StreamsConfig.producerPrefix(ProducerConfig.INTERCEPTOR_
CLASSES_CONFIG),

"io.confluent.monitoring.clients.interceptor.MonitoringProducerInterc
eptor");
settings.put(StreamsConfig.consumerPrefix(ConsumerConfig.INTERCEPTOR_
CLASSES_CONFIG),

"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterc
eptor");
```

Please note the **prefix** used for both consumer and producer interceptors.

With interceptors configured, we can monitor our app through **Confluent Control Center**.

6. Start the Kafka cluster and the **Confluent Control Center** server with:

```
$ docker-compose up -d zookeeper kafka control-center
```

As usual, you are encouraged to inspect the content of the `docker-compose` file in the ~confluent-streams directory to make sure you understand all the settings and discuss it with your peers. Wait a couple of minutes until the cluster is initialized. Open **Confluent Control Center** at http://localhost:9021 and wait until it displays the system health.

7. Create the input and output topics called `lines-topic` and `word-count-topic` respectively. For the moment let's give them each 1 partition and replication factor 1:

```
$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic lines-topic

$ kafka-topics \
    --create \
    --bootstrap-server kafka:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic word-count-topic
```

8. Use **Run** → **Start Debugging** in VS Code or `./gradlew run` in the terminal to run your

streams app.

Ignore the WARNINGS.

Producing Data

9. Run the `kafka-console-producer` that we will use to feed some data to the topic `lines-topic`:

```
$ NS=io.confluent.monitoring.clients.interceptor && \
kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --topic lines-topic \
    --producer-property \
    "interceptor.classes=${NS}.MonitoringProducerInterceptor"
```

10. Open another terminal tab and run the `kafka-console-consumer` for the `word-count-topic` topic:

```
$ NS=io.confluent.monitoring.clients.interceptor && \
kafka-console-consumer \
    --group word-count-consumer \
    --bootstrap-server kafka:9092 \
    --topic word-count-topic \
    --property print.key=true \
    --consumer-property \
    interceptor.classes="${NS}.MonitoringConsumerInterceptor"
```

Note the use of the group name `word-count-consumer` and the `MonitoringConsumerInterceptor` class to enable monitoring of the consumer in Control Center.

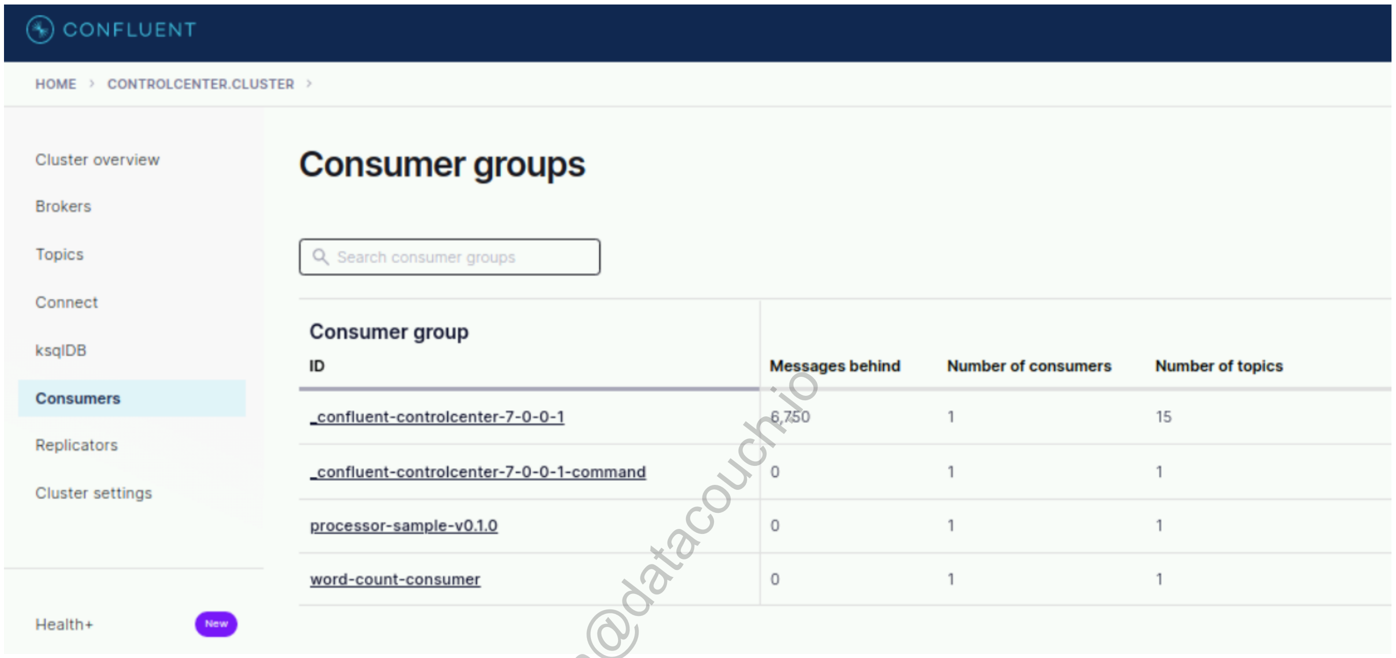
11. In the terminal window where the producer runs, enter a few lines of text such as:

```
Kafka is powering the Confluent streaming platform
Streaming in real-time is more and more important
Our company will invest in real-time streaming
For this we need to know loads about Kafka and Kafka Streams
ksqlDB is a simpler alternative to Kafka Streams
Everybody loves ksqlDB since no programming is required
```

and observe the output in the terminal window where the consumer runs.

> Add more data at will so that there is some activity ongoing.

12. In **Confluent Control Center** (http://localhost:9021) navigate to **Consumers**. You should see something like this:



Consumer groups

Consumer group ID	Messages behind	Number of consumers	Number of topics
_confluent-controlcenter-7-0-0-1	6,750	1	15
_confluent-controlcenter-7-0-0-1-command	0	1	1
processor-sample-v0.1.0	0	1	1
word-count-consumer	0	1	1

Click on our Kafka Streams (processor-sample-v0.1.0) application to investigate its metrics.

Cleanup

13. To stop the producer hit `Ctrl+C`.
14. To stop the consumer hit `Ctrl+C`.
15. To stop the **Kafka Streams** application with **Run** → **Stop Debugging**.
16. To stop the Kafka Cluster and delete the volumes execute this command:

```
$ cd ~/confluent-streams/labs/monitoring/processor-sample
$ docker-compose down -v
```

STOP HERE. THIS IS THE END OF THE EXERCISE.

Appendix B: Running All Labs with Docker

Running Labs in Docker for Desktop

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine you are able to complete the course by building and running your applications from the command line.

- Increase the memory available to Docker Desktop to a minimum of 6 GiB. See the advanced settings for Docker Desktop for Mac, and Docker Desktop for Windows.
- Follow the instructions at → The Lab Environment & Sample Solutions to `git clone` the source code, in each exercise follow the instructions to launch the cluster containers with `docker-compose` on your host machine. The exercise source code will now be on your host machine where you can edit the source code with any editor.
- Begin the exercises by first opening a bash shell on the tools container. All the command line instructions will work from the tools container. This container has been preconfigured with all of the tools you use in the exercises, e.g. `kafka-topics` and `python.`

  ```
  $ docker-compose exec tools bash
  bash-4.4#
  ```

- At the time of writing, the Python prerequisite `confluent_kafka` won't install with the version of Python 3 available in the tools container. You should use `pip` and `python` rather than `pip3` and `python3`.
- At the time of writing, Maven is not installed in the tools container, so the optional maven exercise would require a hefty 300 MB download with `apt-get install maven`. It might be best to skip this optional exercise and simply use it as reference if you use Maven in your day-to-day work.
- Anywhere you are instructed to open additional terminal windows you can `exec` additional bash shells on the tools container with the same command as above on your host machine.

- Any subsequent `docker` or `docker-compose` instructions should be run on your host machine.

Running the Exercise Applications

From the `tools` container you can use command line alternatives to the VS Code steps used in the instructions. Complete the exercise code with an editor on your host machine, then use the following command line instructions to build and run the applications from the exercise directory.

- For Java applications: `./gradlew run`
- For Python applications: `python main.py`

Where you are instructed to use **Run** → **Stop Debugging** in VS code, use `Ctrl+C` to end the running exercise.

Our `docker-compose.yml` file sets the working directory inside the tools container to the `~/confluent-streams directory` Be sure to change the working directory to each exercise directory as stated in each exercise instructions.

```
$ docker-compose exec tools bash
bash-4.4# pwd
/root/confluent-streams
```

To build and run the **Anatomy of a Kafka Streams App** exercise. First make the source code updates to `~/confluent-streams/labs/streams-writing/gradle-sample/src/main/java/streams/MapSample.java` on your host machine. Then enter into the bash shell on your tools container:

```
bash-4.4# cd labs/streams-writing/gradle-sample
bash-4.4# ./gradlew run
```

In the **Monitoring Kafka Streams Applications** exercise, you must use `jconsole` on port 4444 on the host to view JMX metrics. If `jconsole` is not already installed on your host system, install using the password `training`:

```
$ sudo apt install -y openjdk-11-jdk
```

Run a new tools container with a port mapping to expose JMX metrics to the host:

```
$ docker-compose run -p 4444:4444 tools
bash-4.4# cd jmx-sample && ./gradlew run
```

The application will now expose metrics to port 4444 in the container, which is mapped to port 4444 on the host. Running `jconsole` on the host on port 4444 will now pick up the metrics exposed by the application.