Day 3 – Lab2:

Word Count with Spark Streaming & Kafka

In this lab, you'll hook Spark Streaming up to a Kafka topic to count words.

Build Java 11 Spark Docker image

The image we'll use is `bitnami/spark`. As of this writing, the existing image is built for Java 8, so you will need to locally build a new Docker image for Java 11 (see this issue for why).
Run the following script in the lab's root directory:

```
$ ./build-spark-image.sh
```

Ensure that the image got built ok by listing the image:

```
$ docker images
REPOSITORY                                  TAG              IMAGE ID       CREATED
SIZE
...
bitnami/spark                               3-java11         caf5988de8ed   2 minutes ago
1.81GB
...
```

Make sure you see the `binami/spark` image listed with the tag `3-java11`, similar to the line above.

The Java Spark streaming app

Examine the code

Using your favorite editor, open the file `wordcount-spark-streaming/src/main/java/app/SparkKafka.java`.
First, we instantiate a `JavaStreamingContext` given a `SparkConf` & a `Duration`. Then, we use `KafakUtils`, from Spark's Kafka streaming connector, to create a stream of lines coming in from the configured input topic. The Kafka properties come in via resource `streams.properties`.

Next, we establish the `Topology` of our stream by invoking methods on the stream that represent the processing logic that we want to perform.

- `flatMap` takes each line, splits it into words, then provides a stream of words.
- `filter` ensures that we didn't get anything composed of only a blank string.
- `mapToPair` is used to convert each word into an initial tuple of the word and a count (beginning with `1`).
- `reduceByKey` is invoked on all pairs with the same word and updates the word's count by summing the word's value in each pair's value with the next pair's value, which is `1`.
- `print` simply prints the reduced pairs, representing each word and its count.

NOTE: this topology isn't actually executed until the stream is started and lines are presented to the stream.

Last, we start the streaming context and wait for program termination.

Build the streaming app

Now that we have coded our app, we need to build it. Fortunately, we can use `docker` for this so that we don't have to have `maven` and its prerequisites installed locally. Open a terminal in the lab's `wordcount-spark` directory and issue the following command:

```
$ docker run -it --rm -v "$(cd "$PWD/../.."; pwd)":/course-root -w "/course-
root/$(basename $(cd "$PWD/.."; pwd))/$(basename "$PWD")" -v
"$HOME/.m2/repository":/root/.m2/repository maven:3-jdk-11 ./mvnw clean package
```

The command above will build and package our uber jar with the application and all of its dependencies.

Run Kafka & Spark

In order to run our app, we first need to run Kafka & Spark. First, ensure that you've shut down any prior docker containers.

Next, open a new terminal in the lab's root directory & run the Docker Compose stack using the `spark-streaming.yaml` configuration file:

```
$ docker-compose -f spark-streaming.yaml up
```

You will see logs from all the containers that are launched as part of the solution. Once the terminal stops reflecting new output, the infrastructure is initialized.

Submit our Spark application

Now, let's submit our Spark application to the Spark cluster running in our docker environment.

In a new terminal in the lab's root directory, open a bash prompt with the following command:

```
$ docker-compose -f spark-streaming.yaml exec spark-master bash
```

Then, now inside the container, submit your Spark application to the cluster with `spark-submit`:

```
I have no name!@0c68c1412a1d:/opt/bitnami/spark$ spark-submit --master spark://spark-master:7077 --class app.SparkKafka /lab-root/wordcount-spark/target/wordcount-spark-1.0.0-SNAPSHOT.jar 2>/dev/null
```

NOTE: we have mapped the lab's root directory as `/lab-root` in the `spark-master` & `kafka` containers so that you have access to the built Spark streaming jar and the input files that you'll feed into Kafka.

After a short time, you will see output representing one-second windows of batch operations in Spark:

```
-------------------------------------------
Time: 1644874421000 ms
-------------------------------------------

-------------------------------------------
Time: 1644874422000 ms
-------------------------------------------

-------------------------------------------
Time: 1644874423000 ms
-------------------------------------------
...
```

Now that our Spark application is running, it's time to feed it some input lines via the Kafka topic.

In yet another terminal, change into the lab's root directory again and this time, start a bash session in the *kafka* container:

```
$ docker-compose -f spark-streaming.yaml exec kafka bash
```

At the subsequent prompt, pump some lines into the kafka console producer:

```
$ cat /lab-root/ickle-pickle-tickle.txt | kafka-console-producer.sh --bootstrap-server kafka:9092 --topic stream-input
```

This should produce the output count in the terminal where we submitted the spark app:

```
-------------------------------------------
Time: 1644874789000 ms
-------------------------------------------
(went,1)
(captain,1)
(tickle,8)
(coffee,1)
(hope,1)
(ride,1)
(never,1)
(pickle,8)
(flew,2)
(as,1)
...
```

For fun, you can submit the full text of Leo Tolstoy's "War & Peace"!

```
$ cat /lab-root/war-and-peace.txt | kafka-console-producer.sh --bootstrap-server
kafka:9092 --topic stream-input
```

In the lab's root directory, you can now bring down the cluster with the command

```
$ docker-compose -f spark-streaming.yaml down
```

Congratulations, you've completed this lab!