Day 1 – Lab2:

1. Publish and Subscribe

Introduction

In this section we'll use Java and Maven to create a typical Java based producer and consumer of Kafka messages. You have two options for running Maven and Java. Either, you install them natively or you can run the tools through a docker container.

Native install

If you already have Java and Maven installed, make sure you have:

- Java 8 (you can get away with Java 7 until we get to streaming)
- Maven 3.X

Simple Google searches should help you install both tools.

With a successful installation, you should be able to run `mvn -v` from command line. E.g., here is what that may look like (it may look slightly different on your machine based on the maven version and the operating system you use):

```
$ mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T10:41:47-
06:00)
Maven home: /home/pgraff/.jenv/candidates/maven/current
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-oracle/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-78-generic", arch: "amd64", family: "unix"
```

Docker install of Maven and Java

If you decide to use docker, you would want to open a docker instance with all the tools in the root of your lab and keep it open during the lab.

For OSX and Linux, you can simply run:

```
$ cd DIR_WHERE_MY_LAB_IS
```

```
$ docker run -it --rm --name lesson -v "$PWD":/usr/src/lesson -w /usr/src/lesson
maven:3-jdk-8 bash
root@58b8ca1d738c:/usr/src/lesson#
```

If you are on Windows, you may have to replace the `$PWD` with the full path of your lab directory.

You are now running a `bash` shell inside your docker instance. Your directory is mapped to `/usr/src/lesson` inside the docker instance.

You should be able to run the maven command to check that everything is working `mvn --version`. The output should be:

```
root@58b8ca1d738c:/usr/src/lesson30# mvn --version
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-03T19:39:06Z)
Maven home: /usr/share/maven
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-8-openjdk-amd64/jre
Default locale: en, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-79-generic", arch: "amd64", family: "unix"
root@58b8ca1d738c:/usr/src/lesson#
```

For simplicity of the lab instruction, we'll simply refer to the maven builds using the native install. However, you should be able to use the docker image above to build any of the labs. Hence, if you select to use the docker approach, when we specify

```
$ mvn package
```

We assume that you are inside your docker container when you do.

This remainder of the lab consists of two parts. In the first part, we'll create a producer that can create messages in Kafka. In the second part, we'll consume these messages.

Both the `consumer` and `producer` are written in Java. If you're not sure-footed in Java, you may want to simply study the solution or if you can pair program with someone that knows Java.

2. Developing Kafka Applications - Producer API

In this section, you will create a Kafka Producer using the Java API. The next lab will be the creation of the Kafka Consumer so that you can see an end to end example using the API.

Objectives

1. Create topics on the Kafka server for the producer to send messages
2. Understand what the producer Java code is doing
3. Compile and run the example producer

Prerequisites

Like the previous lab, Docker will be used to start a Kafka and Zookeeper server. We will also use a Maven or the Maven Docker image to compile the Java code. You should have a text editor available with Java syntax highlighting for clarity. You will need a basic understanding of Java programming to follow the lab although coding will not be required. The Kafka Producer example will be explained and then you will compile and execute it against the Kafka server.

Instructions

All the directory references in this lab is relative to where you expended the lab files and `labs/02-Publish-And-Subscribe`

1. Open a terminal in this lesson's directory: `docker/`.
2. Open the `docker-compose.yml` file which contains the following:

```
version: '2'
services:
  zookeeper:
    image: wurstmeister/zookeeper:3.4.6
    ports:
      - 2181:2181
  kafka:
    image: wurstmeister/kafka:1.1.0
```

```
    ports:
      - 9092:9092
      - 7203:7203
    environment:
      KAFKA_ADVERTISED_HOST_NAME: [INSERT IP ADDRESS HERE]
#      KAFKA_ADVERTISED_HOST_NAME: localhost
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    depends_on:
      - zookeeper
```

To allow connectivity between the local Java application and Kafka running in Docker, you need to insert the locally assigned IP address for your computer for the value of `KAFKA_ADVERTISED_HOST_NAME`.

On OSX, you can use the following command:

```
$ ifconfig | grep inet
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet6 fe80::1cf1:f9f4:6104:f2a3%en0 prefixlen 64 secured scopeid 0x4
    inet 10.0.1.4 netmask 0xffffff00 broadcast 10.0.1.255
    inet6 2605:6000:1025:407f:101b:9cd8:e973:b9dd prefixlen 64 autoconf secured
    inet6 2605:6000:1025:407f:9077:8802:8c88:e63d prefixlen 64 autoconf temporary
    inet6 fe80::cc8a:c5ff:fe43:b670%awdl0 prefixlen 64 scopeid 0x8
    inet6 fe80::7df6:ec93:ffea:367a%utun0 prefixlen 64 scopeid 0xa
```

In this case, the IP address to use is `10.0.1.4`. Make sure you *do not* use `127.0.0.1` because that will not work correctly.

On Windows, you can use the following command:

```
$ ipconfig
Ethernet adapter Local Area Connection:
Connection-specific DNS Suffix . : hsd1.ut.comcast.net.
IP Address. . . . . . . . . . . . : 192.168.201.245
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Default Gateway . . . . . . . . . : 192.168.201.1
```

Here the IP address to use is `192.168.201.245`.
After you replace the line in `docker-compose.yml`, it should look like this:

```
KAFKA_ADVERTISED_HOST_NAME: 192.168.201.245
```

Save the `docker-compose.yml` file after making this modification.

We have noticed on some configurations of Windows and Linux that the use of `KAFKA_ADVERTISED_HOST_NAME` does not work properly (the Kafka clients can't connect). We've not found the source of this problem, but in many of the cases we've seen, the use of `localhost` instead of the host IP may work. Note though, that the use of `localhost`prevents you from running multiple Kafka brokers on the same machine.

3. Start the Kafka and Zookeeper processes using Docker Compose:

```
$ docker-compose up
```

4. Open an additional terminal window in the lesson directory, `docker/`. We are going to create two topics that will be used in the Producer program. Run the following commands:

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-topics.sh --create --zookeeper
zookeeper:2181 --replication-factor 1 --partitions 1 --topic user-events
$ docker-compose exec kafka /opt/kafka/bin/kafka-topics.sh --create --zookeeper
zookeeper:2181 --replication-factor 1 --partitions 1 --topic global-events
```

5. List the topics to double check they were created without any issues.

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-topics.sh --list --zookeeper
zookeeper:2181

global-events
user-events
```

6. Open `producer/pom.xml` in your favorite text editor. At the time of this writing, the current stable version of Kafka is 1.1.0. Maven is being used for dependency management in this lab and includes the following in the `pom.xml` for Kafka:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>1.1.0</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>19.0</version>
</dependency>
```

The only other dependency being used is `guava` from Google which contains some helpful utility classes that we are going to use to load a resource file.

7. Next open `producer/src/main/java/com/example/Producer.java` in your text editor. This class is fairly simple Java application but contains all the functionality necessary to operate as a Kafka Producer. The application has two main responsibilities:
    - Initialize and configure a KafkaProducer
    - Send messsages to topics with the Producer object

To create our Producer object, we must create an instance of `org.apache.kafka.clients.producer.KafkaProducer` which requires a set of properties for initialization. While it is possible to add the properties directly in Java code, a more likely scenario is that the configuration would be externalized in a properties file. The following code instantiates a `KafkaProducer` object using `resources/producer.properties`.

```
KafkaProducer<String, String> producer;
try (InputStream props = Resources.getResource("producer.properties").openStream()) {
    Properties properties = new Properties();
    properties.load(props);
    producer = new KafkaProducer<>(properties);
}
```

Open `resources/producer.properties` and you can see that the configuration is minimal:

```
acks=all
retries=0
bootstrap.servers=localhost:9092
key.serializer=org.apache.kafka.common.serialization.StringSerializer
value.serializer=org.apache.kafka.common.serialization.StringSerializer
```

- `acks` is the number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. A setting of `all` is the strongest guarantee available.
- Setting `retries` to a value greater than 0 will cause the client to resend any record whose send fails with a potentially transient error.
- `bootstrap.servers` is our required list of host/port pairs to connect to Kafka. In this case, we only have one server. The Docker Compose file exposes the Kafka port so that it can be accessed through `localhost`.
- `key.serializer` is the serializer class for key that implements the `Serializer` interface.

- `value.serializer` is the serializer class for value that implements the `Serializer` interface.

There are many configuration options available for Kafka producers that should be explored for a production environment.

8. Once you have a Producer instance, you can post messages to a topic using the ProducerRecord. A `ProducerRecord` is a key/value pair that consists of a topic name to which the record is being sent, an optional partition number, an optional key, and required value.

   In our lab, we are going to send 100000 messages in a loop. Each iteration of the loop will consist of sending a message to the `user-events` topic with a key/value pair. Every 100 iterations, we also send a randomized message to the `global-events` topic. The message sent to `global-events` does not have a key specified which normally means that Kafka will assign a partition in a round-robin fashion but our topic only contains 1 partition.

   After the loop is complete, it is important to call `producer.close()` to end the producer lifecycle. This method blocks the process until all the messages are sent to the server. This is called in the `finally` block to guarantee that it is called. It is also possible to use a Kafka producer in a try-with-resources statement.

```
try {
    for (int i = 0; i < 100000; i++) {
        producer.send(new ProducerRecord<String, String>(
              "user-events", // topic
              "user_id_" + i, // key
              "some_value_" + System.nanoTime())); // value

        if (i % 100 == 0) {
            String event = global_events[(int) (Math.random() *
global_events.length)] + "_" + System.nanoTime();

            producer.send(new ProducerRecord<String, String>(
                "global-events", // topic
                event)); // value

            producer.flush();
            System.out.println("Sent message number " + i);
        }
    }
} catch (Throwable throwable) {
    System.out.println(throwable.getStackTrace());
} finally {
    producer.close();
```

```
}
```

9. Now we are ready to compile and run the lab. In a terminal, change to the `lab` directory and run the following Maven command using a Docker image:

```
$ mvn clean package
```

10. For convenience, the project is set up so that the `package` target produces a single executable: `target/producer`. Run the producer to send messages to our two topics -- `user-events` and `global-events`.

```
$ target/producer
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
Sent message number 0
Sent message number 100
...
Sent message number 99800
Sent message number 99900
```

11. *Don't* stop the Kafka and Zookeeper servers because they will be used in the next section focusing on the Consumer API.

Conclusion

We have now successfully sent a series of messages to Kafka using the Producer API. In the next section, we will write a consumer program to process the messages from Kafka.

3. Developing Kafka Applications - Consumer API

In this section, you will create a Kafka Consumer using the Java API. This is the continuation of the previous lab in which a Kafka Producer was created to send messages to two topics -- `user-events` and `global-events`.

Objectives

1. Understand what the consumer Java code is doing
2. Compile and run the consumer program
3. Observe the interaction between producer and consumer programs

Prerequisites

Like the first part of this of this lab, we will use a Maven Docker image to compile the Kafka consumer Java application. You should have a text editor available with Java syntax highlighting for clarity. You will need a basic understanding of Java programming to follow the lab although coding will not be required. You should have already completed the previous Kafka Producer lab so that there are messages ready in the Kafka server for the Consumer to process.

Instructions

1. Open `consumer/src/main/java/com/example/Consumer.java` in your favorite text editor. Like the `Producer` we saw in the previous lab, this is a fairly simple Java class but can be expanded upon in a real application. For example, after processing the incoming records from Kafka, you would probably want to do something interesting like store them in HBase for later analysis. This application has two main responsibilities:
    - Initialize and configure a KafkaConsumer
    - Poll for new records in an infinite loop

The first thing to notice is that a `KafkaConsumer` requires a set of properties upon creation just like a `KafkaProducer`. You can add these properties directly to code but a better

solution is to externalize them in a properties file. The following code instantiates a `KafkaConsumer` object using `resources/consumer.properties`.

```
KafkaConsumer<String, String> consumer;
try (InputStream props = Resources.getResource("consumer.properties").openStream()) {
    Properties properties = new Properties();
    properties.load(props);
    consumer = new KafkaConsumer<>(properties);
}
```

2. Open `resources/consumer.properties` and you see that the required configuration is minimal like the producer.

```
bootstrap.servers=localhost:9092
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
group.id=test
auto.offset.reset=earliest
```

   - `bootstrap.servers` is our required list of host/port pairs to connect to Kafka. In this case, we only have one server.
   - `key.deserializer` is the deserializer class for key that implements the `Deserializer` interface.
   - `value.deserializer` is the deserializer class for value that implements the `Deserializer` interface.
   - `group.id` is a string that uniquely identifies the group of consumer processes to which this consumer belongs. In our case, we are just using the value of `test` for an example.
   - `auto.offset.reset` determines what to do when there is no initial offset in Zookeeper or Kafka from which to read records. The first time that a consumer is run will be the first time that the Kafka broker has seen the consumer group that the consumer is using. The default behavior is to position newly created consumer groups at the end of existing data which means that the producer data that we ran previously would not be read. By setting this to `earliest`, we are telling the consumer to reset the offset to the smallest offset.

   Like the producer, there are many configuration options available for Kafka consumer that should be explored for a production environment.

3. Open `consumer/src/main/java/com/example/Consumer.java` again. A consumer can subscribe to one ore more topics. In this lab, the consumer will listen to messages from two topics with the following code:
   ```
   consumer.subscribe(Arrays.asList("user-events", "global-events"));
   ```

4. Once the consumer has subscribed to the topics, the consumer can poll for new messages in the following loop:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);

    for (ConsumerRecord<String, String> record : records) {
        switch (record.topic()) {
            case "user-events":
                System.out.println("Received user-events message - key: " +
record.key() + " value: " + record.value());
                break;
            case "global-events":
                System.out.println("Received global-events message - value: " +
record.value());
                break;
            default:
                throw new IllegalStateException("Shouldn't be possible to get
message on topic " + record.topic());
        }
    }
}
```

For each iteration of the loop, the consumer will fetch records for the topics. On each poll, the consumer will use the last consumed offset as the starting offset and fetch sequentially. The `poll` method takes a timeout in milliseconds to spend waiting if data is not available in the buffer.

The returned object of the `poll` method is an `Iterable` that contains all the new records. From there our example lab just uses a `switch` statement to process each type of topic. In a real application, you would do something more interesting here than output the results to `stdout`.

5. Now we are ready to compile and run the lab. In a terminal, change to the `lab` directory and run the following Maven targets:

```
$ docker run -it --rm --name lesson -v "$PWD":/usr/src/lesson -w /usr/src/lesson
maven:3-jdk-8 mvn clean package
```

6. For convenience, the project is set up so that the `package` target produces a single executable: `target/consumer`. Run the consumer:

```
$ target/consumer
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
Received user-events message - key: user_id_0 value: some_value_148511272601285
Received user-events message - key: user_id_1 value: some_value_148511557371815
Received user-events message - key: user_id_2 value: some_value_148511557456741
....
```

After the consumer has processed all of the messages, start the producer again in another terminal window and you will see the consumer output the messages almost immediately. The consumer will run indefinitely until you press `Ctrl-C` in the terminal window.

7. Finally, change back into the `docker/` directory to shut down the Kafka and Zookeeper servers.

```
$ docker-compose down
```

Conclusion

We have now seen in action a basic producer that sends messages to the Kafka broker and then a consumer to process them. The examples we've shown here can be incorporated into a larger, more sophisticated application.

Congratulations, this lab is complete!