LAB 2 Writing a Kafka Producer in Java

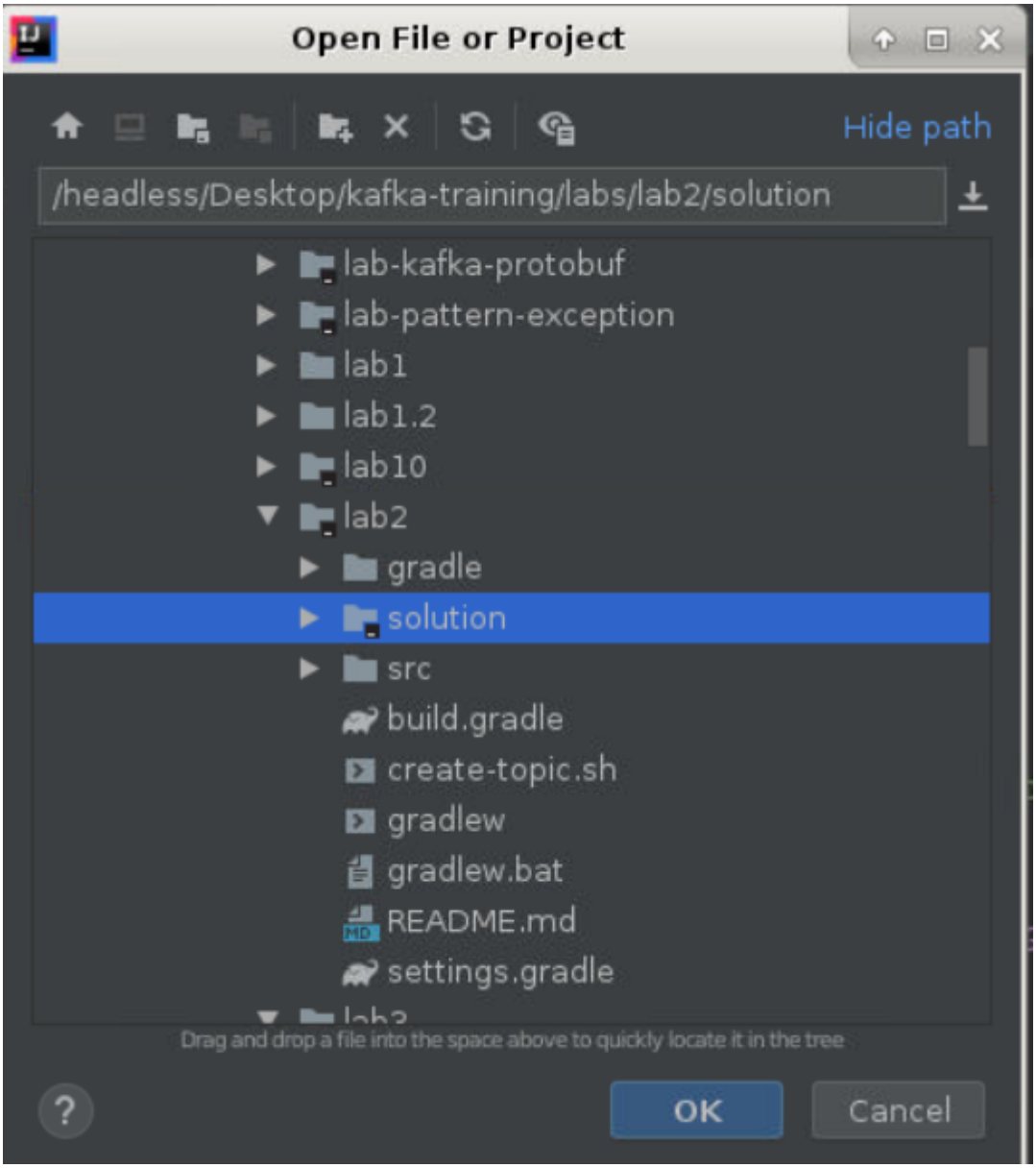
Welcome to the session 2 lab. The work for this lab is done in `~/kafka-training/labs/lab2`.

In this lab, you are going to create simple Java Kafka producer. You create a new replicated Kafka topic called `my-example-topic`, then you create a Kafka producer that uses this topic to send records. You will send records with the Kafka producer. You will send records synchronously. Later, you will send records asynchronously.

To start Kafka and ZooKeeper use the same technique you did in lab1.2. You can use the start up scripts that you wrote in lab1.2.

Note: Lab solution is available in following directory:

`~/kafka-training/labs/lab2/solution`



You can open `Intellij_IDE_guide.pdf` to learn how to open and run java project in intelliJ.

Create Replicated Kafka Topic

Next, you need to create a replicated topic.

ACTION - EDIT `~/kafka-training/labs/lab2/create-topic.sh` and follow instructions in file.

~/kafka-training/labs/lab2/create-topic.sh

```
#!/usr/bin/env bash
cd ~/kafka-training

## Create topics
kafka/bin/kafka-topics.sh --create \
```

```
    --replication-factor 1 \
    --partitions 13 \
    --topic my-example-topic \
    --zookeeper  localhost:2181


## List created topics
kafka/bin/kafka-topics.sh --list \
    --zookeeper localhost:2181
```

Above we create a topic named `my-example-topic` with 13 partitions and a replication factor of 3. Then we list the Kafka topics.

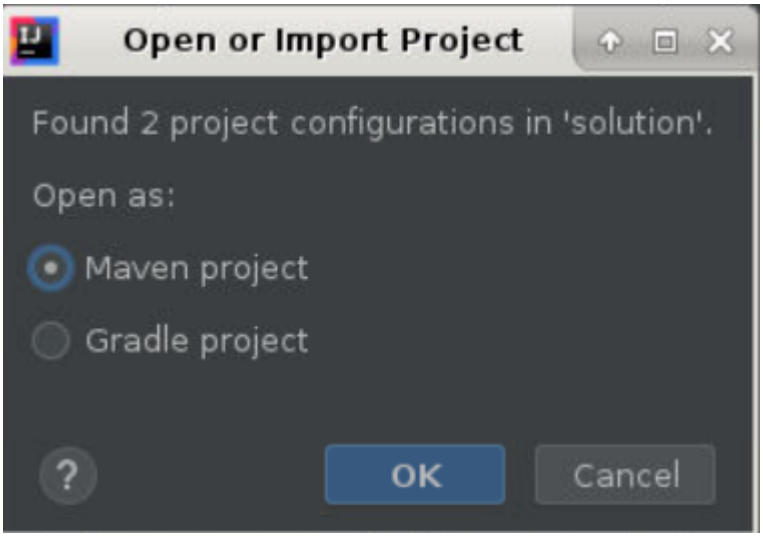
ACTION - RUN `create-topic.sh` as follows:

Output from running create-topic.sh

```
$ cd ~/kafka-training/labs/lab2
$ ./create-topic.sh
Created topic "my-example-topic".
__consumer_offsets
my-example-topic
my-failsafe-topic
```

Maven / Gradle

Lab supports both maven and gradle as a build tool. Make sure to choose preferred build tool while opening project in IntelliJ:



Note:

- Both maven and gradle are supported in next labs as well.
- Edit `pom.xml` or `build.gradle` in the next step accordingly.

Maven pom.xml

ACTION - EDIT `~/kafka-training/labs/lab2/pom.xml` and follow the instructions in file.

~/kafka-training/labs/lab2/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fenago-kafka</groupId>
  <artifactId>lab2-solution</artifactId>
  <version>1.0.0</version>
  <inceptionYear>2017</inceptionYear>
  <dependencies>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>1.1.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.2.2</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Gradle Build Script

For this example, we use gradle to build the project.

ACTION - EDIT `~/kafka-training/labs/lab2/build.gradle` and follow the instructions in file.

~/kafka-training/labs/lab2/build.gradle

```
group 'fenago-kafka'
version '1.0-SNAPSHOT'
```

```
apply plugin: 'java'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.apache.kafka:kafka-clients:1.1.0'
    compile 'ch.qos.logback:logback-classic:1.2.2'
}
```

Notice that we import the jar file `kafka-clients:1.1.0`. Apache Kafka uses `sl4j` so to setup logging we use logback (`ch.qos.logback:logback-classic:1.2.2`).

Construct a Kafka Producer

To create a Kafka producer, you will need to pass it a list of bootstrap servers (a list of Kafka brokers). You will also specify a `client.id` that uniquely identifies this Producer client. In this example, we are going to send messages with ids. The message body is a string, so we need a record value serializer as we will send the message body in the Kafka's records value field. The message id (long), will be sent as the Kafka's records key. You will need to specify a Key serializer and a value serializer, which Kafka will use to encode the message id as a Kafka record key, and the message body as the Kafka record value.

Common Kafka imports and constants

Next, we will import the Kafka packages and define a constant for the topic and a constant to define the list of bootstrap servers that the producer will connect.

KafkaProducerExample.java - imports and constants

~/kafka-training/labs/lab2/src/main/java/com/fenago/kafka/KafkaProducerExample.java

```
package com.fenago.kafka;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.LongSerializer;
import org.apache.kafka.common.serialization.StringSerializer;
import java.util.Properties;

public class KafkaProducerExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";
```

Notice that `KafkaProducerExample` imports `LongSerializer` which gets configured as the Kafka record key serializer, and imports `StringSerializer` which gets configured as the record value serializer. The constant `BOOTSTRAP_SERVERS` is set to `localhost:9092,localhost:9093,localhost:9094` which is the three

Kafka servers that we started up in the last lesson. Go ahead and make sure all three Kafka servers are running. The constant `TOPIC` is set to the replicated Kafka topic that we just created.

ACTION - EDIT src/main/java/com/fenago/kafka/KafkaProducerExample.java and add constants as above.

Create Kafka Producer to send records

Now, that we imported the Kafka classes and defined some constants, let's create a Kafka producer.

KafkaProducerExample.java - Create Producer to send Records

~/kafka-training/labs/lab2/src/main/java/com/fenago/kafka/KafkaProducerExample.java

```
public class KafkaProducerExample {
    ...
    private static Producer<Long, String> createProducer() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                                        BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                                    LongSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                                  StringSerializer.class.getName());
        return new KafkaProducer<>(props);
    }
```

To create a Kafka producer, you use `java.util.Properties` and define certain properties that we pass to the constructor of a `KafkaProducer`.

Above `KafkaProducerExample.createProducer` sets the `BOOTSTRAP_SERVERS_CONFIG` ("bootstrap.servers) property to the list of broker addresses we defined earlier. `BOOTSTRAP_SERVERS_CONFIG` value is a comma separated list of host/port pairs that the `Producer` uses to establish an initial connection to the Kafka cluster. The producer uses of all servers in the cluster no matter which ones we list here. This list only specifies the initial Kafka brokers used to discover the full set of servers of the Kafka cluster. If a server in this list is down, the producer will just go to the next broker in the list to discover the full topology of the Kafka cluster.

The `CLIENT_ID_CONFIG` ("client.id") is an id to pass to the server when making requests so the server can track the source of requests beyond just IP/port by passing a producer name for things like server-side request logging.

The `KEY_SERIALIZER_CLASS_CONFIG` ("key.serializer") is a Kafka Serializer class for Kafka record keys that implements the Kafka Serializer interface. Notice that we set this to `LongSerializer` as the message ids in our example are longs.

The `VALUE_SERIALIZER_CLASS_CONFIG` ("value.serializer") is a Kafka Serializer class for Kafka record values that implements the Kafka Serializer interface. Notice that we set this to `StringSerializer` as the message body in our example are strings.

ACTION - EDIT KafkaProducerExample.java and finish createProducer.

Send records synchronously with Kafka Producer

Kafka provides a synchronous send method to send a record to a topic. Let's use this method to send some message ids and messages to the Kafka topic we created earlier.

KafkaProducerExample.java - Send Records Synchronously

~/kafka-training/labs/lab2/src/main/java/com/fenago/kafka/KafkaProducerExample.java

```
public class KafkaProducerExample {
  ...

  static void runProducer(final int sendMessageCount) throws Exception {
      final Producer<Long, String> producer = createProducer();
      long time = System.currentTimeMillis();

      try {
          for (long index = time; index < time + sendMessageCount; index++) {
              final ProducerRecord<Long, String> record =
                      new ProducerRecord<>(TOPIC, index,
                                  "Hello Mom " + index);

              RecordMetadata metadata = producer.send(record).get();

              long elapsedTime = System.currentTimeMillis() - time;
              System.out.printf("sent record(key=%s value=%s) " +
                              "meta(partition=%d, offset=%d) time=%d\n",
                      record.key(), record.value(), metadata.partition(),
                      metadata.offset(), elapsedTime);

          }
      } finally {
          producer.flush();
          producer.close();
      }
  }
  ...
```

The above just iterates through a for loop, creating a `ProducerRecord` sending an example message (`"Hello Mom " + index`) as the `record value` and the for loop `index` as the `record key` . For each iteration, `runProducer` calls the `send` method of the `producer` (`RecordMetadata metadata = producer.send(record).get()`). The send method returns a Java `Future` .

The response RecordMetadata has 'partition' where the record was written and the 'offset' of the record in that partition.

Notice the call to flush and close. Kafka will auto flush on its own, but you can also call flush explicitly which will send the accumulated records now. It is polite to close the connection when we are done.

ACTION - EDIT KafkaProducerExample.java and finish runProducer.

Running the Kafka Producer

Next you define the `main` method.

KafkaProducerExample.java - Running the Producer

~/kafka-training/labs/lab2/src/main/java/com/fenago/kafka/KafkaProducerExample.java

```
public static void main(String... args) throws Exception {
    if (args.length == 0) {
        runProducer(5);
    } else {
        runProducer(Integer.parseInt(args[0]));
    }
}
```

The `main` method just calls `runProducer`.

ACTION - EDIT KafkaProducerExample.java and finish main method.

ACTION - RUN KafkaProducerExample from the IDE

Send records asynchronously with Kafka Producer

Kafka provides an asynchronous send method to send a record to a topic. Let's use this method to send some message ids and messages to the Kafka topic we created earlier. The big difference here will be that we use a lambda expression to define a callback.

KafkaProducerExample.java - Send Records Asynchronously with Kafka Producer

~/kafka-training/labs/lab2/src/main/java/com/fenago/kafka/KafkaProducerExample.java

```
static void runProducer(final int sendMessageCount) throws InterruptedException {
    final Producer<Long, String> producer = createProducer();
    long time = System.currentTimeMillis();
    final CountDownLatch countDownLatch = new CountDownLatch(sendMessageCount);

    try {
        for (long index = time; index < time + sendMessageCount; index++) {
            final ProducerRecord<Long, String> record =
                    new ProducerRecord<>(TOPIC, index, "Hello Mom " + index);
            producer.send(record, (metadata, exception) -> {
                long elapsedTime = System.currentTimeMillis() - time;
                if (metadata != null) {
                    System.out.printf("sent record(key=%s value=%s) " +
                                    "meta(partition=%d, offset=%d) time=%d\n",
                            record.key(), record.value(), metadata.partition(),
                            metadata.offset(), elapsedTime);
                } else {
                    exception.printStackTrace();
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await(25, TimeUnit.SECONDS);
    }finally {
        producer.flush();
        producer.close();
    }
}
```

Notice the use of a CountDownLatch so we can send all N messages and then wait for them all to send.

ACTION - EDIT KafkaProducerExample.java and change runProducer method to use async API

ACTION - RUN KafkaProducerExample from the IDE

Async Interface Callback and Async Send Method

Kafka defines a Callback interface that you use for asynchronous operations. The callback interface allows code to execute when the request is complete. The callback executes in a background I/O thread so it should be fast (don't block it). The `onCompletion(RecordMetadata metadata, Exception exception)` gets called when the asynchronous operation completes. The `metadata` gets set (not null) if the operation was a success, and the exception gets set (not null) if the operation had an error.

The async `send` method is used to send a record to a topic, and the provided callback gets called when the `send` is acknowledged. The `send` method is asynchronous, and when called returns immediately once the record gets stored in the buffer of records waiting to post to the Kafka broker. The `send` method allows sending many records in parallel without blocking to wait for the response after each one.

> *Since the send call is asynchronous it returns a Future for the RecordMetadata that will be assigned to this record. Invoking get() on this future will block until the associated request completes and then return the metadata for the record or throw any exception that occurred while sending the record. KafkaProducer*

Conclusion Kafka Producer example

You created a simple example that creates a Kafka Producer. First, you created a new replicated Kafka topic; then you created Kafka Producer in Java that uses the Kafka replicated topic to send records. You sent records with the Kafka Producer using async and sync send methods.

Review Kafka Producer

What does the Callback lambda do?

The callback gets notified when the request is complete.

What will happen if the first server is down in the bootstrap list? Can the producer still connect to the other Kafka brokers in the cluster?

The producer will try to contact the next broker in the list. Any of the brokers once contacted, will let the producer know about the entire Kafka cluster. The Producer will connect as long as at least one of the brokers in the list is running. If you have 100 brokers and two of the brokers in a list of three servers in the bootstrap list are down, the producer can still use the 98 remaining brokers.

When would you use Kafka async send vs. sync send?

If you were already using an async code (Akka, QBit, Reakt, Vert.x) base, and you wanted to send records quickly.

Why do you need two serializers for a Kafka record?

One of the serializers is for the Kafka record key, and the other is for the Kafka record value.