Workshop on Kafka

SETTING UP PRODUCER AND CONSUMER USING KAFKA AND ZOOKEEPER

NATIONAL UNIVERSITY OF SINGAPORE | [COMPANY ADDRESS]

Table of Contents

Introduction

Apache Kafka is a publish/subscribe messaging system designed to solve this problem. It is often described as a "distributed commit log" or more recently as a "distributing streaming platform." Data within Kafka is stored durably, in order, and can be read deterministically. Not only can Kafka handle multiple consumers, but durable message retention means that consumers do not always need to work in real time. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Objective

In this workshop, participants will learn to install Kafka messaging system and create applications that use the Kafka framework to implement multiple producers and multiple consumers.

Installing Kafka

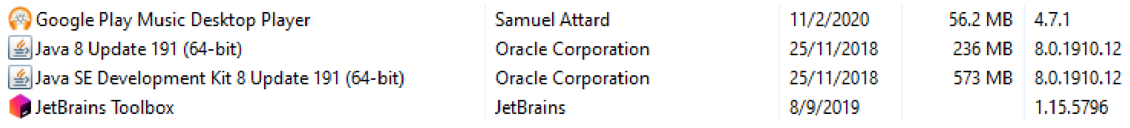
This section describes how to get started with the Apache Kafka broker, including how to set up Apache Zookeeper, which is used by Kafka for storing metadata for the brokers.

Choosing an Operating System

Apache Kafka is a Java application, and can run on many operating systems. This includes Windows, MacOS, Linux, and others. The installation steps in this workshop will be focused on setting up and using Kafka in a Windows environment, as this is the most common OS on which it is installed in ISS Context. Refer to official documentation for other OS installation.

Installing Java

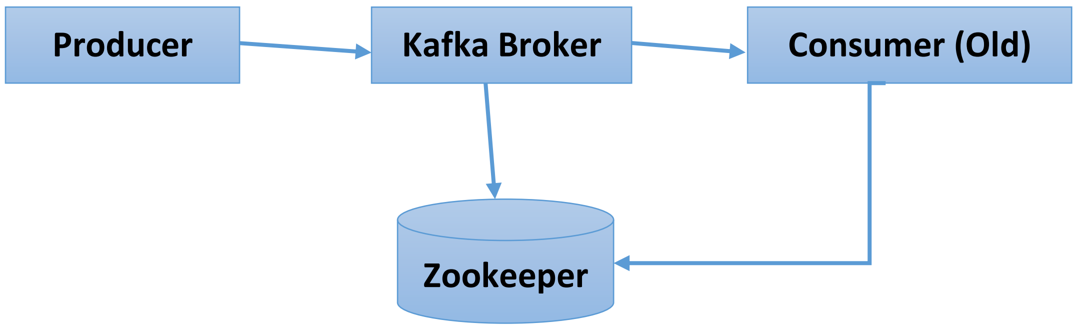
Prior to installing either Zookeeper or Kafka, we will need a Java environment set up and functioning. This should be a Java 8 version. Though Zookeeper and Kafka will work with a runtime edition of Java, it may be more convenient when developing tools and applications to have the full Java Development Kit (JDK).



Google Play Music Desktop Player	Samuel Attard	11/2/2020	56.2 MB	4.7.1
Java 8 Update 191 (64-bit)	Oracle Corporation	25/11/2018	236 MB	8.0.1910.12
Java SE Development Kit 8 Update 191 (64-bit)	Oracle Corporation	25/11/2018	573 MB	8.0.1910.12
JetBrains Toolbox	JetBrains	8/9/2019		1.15.5796

Installing Kafka and Zookeeper

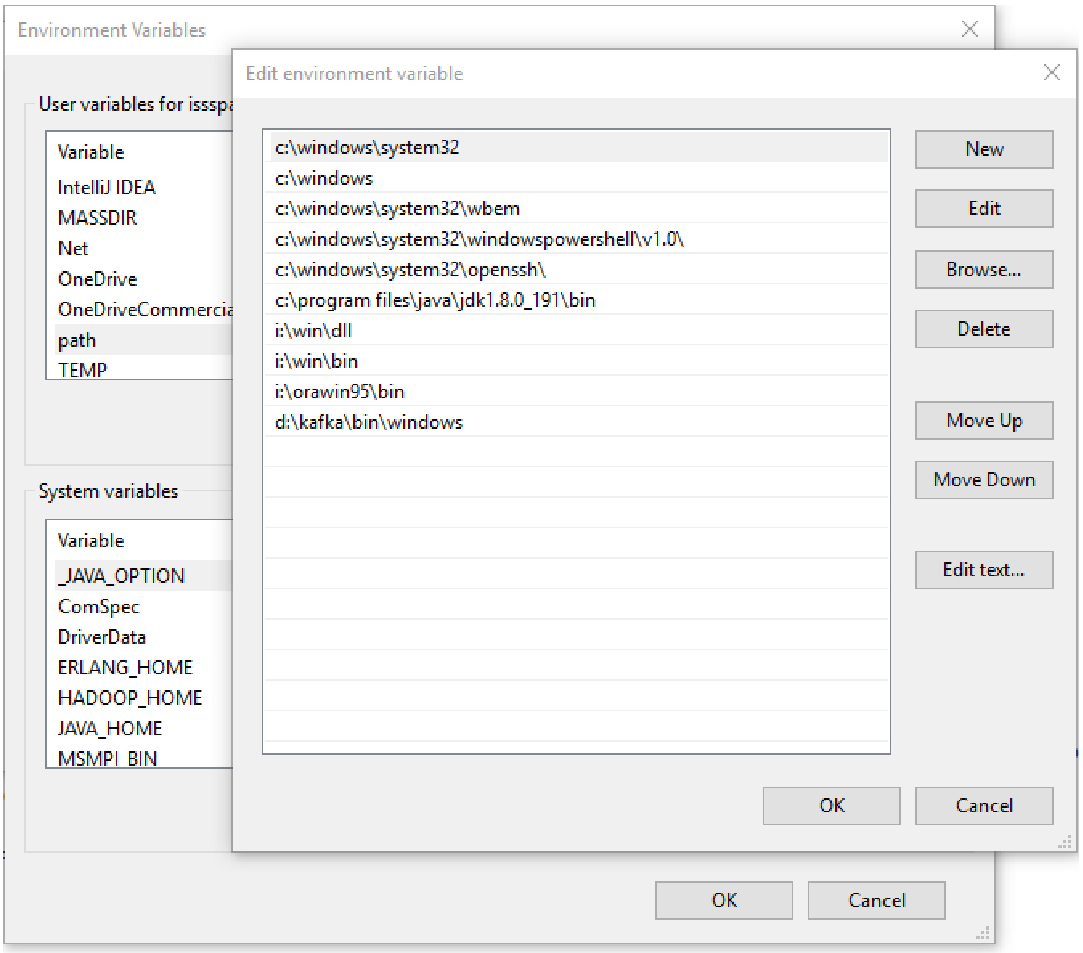
Apache Kafka 2.4.0 uses Zookeeper to store metadata about the Kafka cluster, as well as consumer client details as seen in picture below.



While it is possible to run a Zookeeper server using scripts contained in the Kafka distribution, it is trivial to install a full version of Zookeeper from the distribution. Kafka has been tested extensively with the stable 3.5.6 release of Zookeeper, which can be downloaded from apache.org.

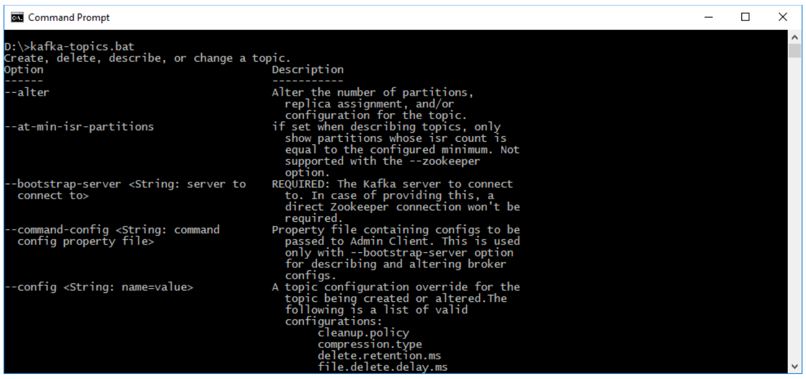
Download the current Kafka binaries from https://kafka.apache.org/downloads

Extract Kafka at the root of D:\ using a valid zip tool such as 7Zip or WinRaR. Setup Kafka bins in the Environment variables section by editing **path** variable as shown below



```
c:\windows\system32
c:\windows
c:\windows\system32\wbem
c:\windows\system32\windowspowershell\v1.0\
c:\windows\system32\openssh\
c:\program files\java\jdk1.8.0_191\bin
i:\win\dll
i:\win\bin
i:\orawin95\bin
d:\kafka\bin\windows
```

Try Kafka commands using kafka-topics.bat (for example) open a cmd prompt and try a kafka command such as kafka-topics.bat as shown below:



```
D:\>kafka-topics.bat
Create, delete, describe, or change a topic.
Option                                   Description
------                                   -----------
--alter                                  Alter the number of partitions,
                                           replica assignment, and/or
                                           configuration for the topic.
--at-min-isr-partitions                  if set when describing topics, only
                                           show partitions whose isr count is
                                           equal to the configured minimum. Not
                                           supported with the --zookeeper
                                           option.
--bootstrap-server <String: server to    REQUIRED: The Kafka server to connect
  connect to>                              to. In case of providing this, a
                                           direct Zookeeper connection won't be
                                           required.
--command-config <String: command        Property file containing configs to be
  config property file>                    passed to Admin Client. This is used
                                           only with --bootstrap-server option
                                           for describing and altering broker
                                           configs.
--config <String: name=value>            A topic configuration override for the
                                           topic being created or altered.The
                                           following is a list of valid
                                           configurations:
                                                cleanup.policy
                                                compression.type
                                                delete.retention.ms
                                                file.delete.delay.ms
```

Edit Zookeeper & Kafka configs using some text editors

```
zookeeper.properties:

dataDir=D:/kafka_2.12-2.4.0/data/zookeeper
```

Note: yes, the slashes are inversed

```
server.properties:

log.dirs=D:/kafka_2.12-2.4.0/data/kafka
```

Note: yes, the slashes are inversed

Starting the Kafka and Zookeeper Servers

The default configuration provided with the Kafka distribution is sufficient to run a standalone server as a proof of concept, but it will not be sufficient for most production installations. There are numerous configuration options for Kafka that control all aspects of setup and tuning. Many options can be left to the default settings, as they deal with tuning aspects of the Kafka broker that will not be applicable until we have a specific use case to work with and a specific use case that requires adjusting these settings.

Start Zookeeper in one command line:

```
zookeeper-server-start.bat D:\kafka_2.12-
2.4.0\config\zookeeper.properties
```

Start Kafka in another command line:

```
kafka-server-start.bat D:\kafka_2.12-
2.4.0\config\server.properties
```

Once the Kafka broker is started, we can verify that it is working by performing some simple operations against the cluster creating a test topic, producing some messages, and consuming the same messages.

Kafka Command Line Interface (CLI)

The Kafka distribution provides a command utility to send messages from the command line. It starts up a terminal window where everything you type is sent to the Kafka topic. In this section we will get ourselves familiar with some commands often be used when we work with Apache Kafka command line interface (CLI).

kafka-topics

The `kafka-topics.bat` tool can be used to create, alter, list, and describe topics. For example to list existing topics:

```
> kafka-topics.bat --zookeeper 127.0.0.1:2181 --list
```

To create a topic by name `first_topic`:

```
> kafka-topics.bat --zookeeper 127.0.0.1:2181 --topic
    first_topic --create --partitions 3 --replication-factor 1
```

To describe the new topic:

```
> kafka-topics.bat --zookeeper 127.0.0.1:2181 --topic
first_topic --describe
```

To create another topic:

```
> kafka-topics.bat --zookeeper 127.0.0.1:2181 --topic
second_topic --create --partitions 1 --replication-factor 1
```

To list all topics:

```
kafka-topics.bat --zookeeper 127.0.0.1:2181 –list
```

To delete a particular topic:

Do Not Try this in Windows machine as you need to physically delete folders and restart the server to recover from an access rights bug

```
kafka-topics.bat --zookeeper 127.0.0.1:2181 --topic
second_topic --delete
```

kafka-console-producer

Producers write data to topics. They understand which broker and partition to write to. In case of broker failures, producers will automatically recover. The load is balanced by multiple brokers and number of partitions.

The `kafka-console-producer` tool can be used to read data from standard output and write it to a Kafka topic. For example:

```
kafka-console-producer --broker-list 127.0.0.1:9092 --topic
first_topic
```

The user can also configure different acknowledgement settings discussed in lecture using (acks=0 or 1 [default] or all):

```
kafka-console-producer --broker-list 127.0.0.1:9092 --topic
first_topic --producer-property acks=all
```

We can also start a whole new topic out of blue using:

```
kafka-console-producer --broker-list 127.0.0.1:9092 --topic
new_topic
```

The first message comes with a warning as the leader is not elected actively. But subsequent messages would easily go through as the server has sorted the leader election after first producer record. We can verify the new topic presence using the following commands:

```
kafka-topics.bat --zookeeper 127.0.0.1:2181 --list

kafka-topics.bat --zookeeper 127.0.0.1:2181 --topic new_topic
--describe
```

Additional configuration settings is needed in the `server.properties` along with a server restart for default partitions to be increased from 1 to 3.

Also note that producers can choose to send messages with key. If key-null, then round robin algorithm is chosen. If key is used, then all messages with that key are sent to same partition. Key is useful for message ordering purposes.

kafka-console-consumer

Consumer reads data from a topic (identified by a name). They know which broker to read from. In case of failures, they know how to recover. Data is read in order within each partitions.

The `kafka-console-consumer` tool can be used to read data from a Kafka topic and write it to standard output. For example:

```
kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 -
-topic first_topic

kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 -
-topic new_topic
```

You won't see new messages. as the consumer starts listening only from current time. So test the consumer by adding new messages to the producer by

```
kafka-console-producer.bat --broker-list 127.0.0.1:9092 --
topic first_topic
```

In order to consume messages from beginning, try the command:

```
kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 -
-topic first_topic --from-beginning
```

kafka-consumer-groups

Kafka consumers are generally part of a group. A group is usually an IDE or Application. For example we can create a consumer group as below:

```
kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 -
-topic first_topic --group my-first-application
```

We can also use the consumer group to read all messages from the beginning.

```
kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 -
-topic first_topic --group my-first-application -from-
beginning
```

But once a group is specified, and from-beginning flag is issued once the **offset** keeps track of it and a subsequent command as below will not read from start this time.

```
kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 -
-topic first_topic --group my-first-application -from-
beginning
```

Next, we look at the kakfka-consumer-groups is command to work with all the existing consumer groups. For example, to list all the consumer-groups use:

```
kafka-consumer-groups.bat --bootstrap-server 127.0.0.1:9092 --
list
```

To find the active members in a particular group that are consuming to current streams produced, use the below command:

```
kafka-consumer-groups.bat --bootstrap-server 127.0.0.1:9092 --
describe --group my-second-application
```

Lag 0 means that all data has been read. The list also produced current offset and end offset. You can read the pending messages using the following command

```
kafka-console-consumer.bat --bootstrap-server 127.0.0.1:9092 --
topic first_topic --group my-second-application
```

Kafka Tools UI

We can also use a UI tool to work with Kafka messages. There are many tools available, the lecturer will demonstrate Kafka tool that is quite popular in the community. You can download and install the tool from:

http://www.kafkatool.com/download.html

Feel free to explore this UI tool which is an easy to pick tool. Please note that this tool is only for personal use.

Client Languages Supported

In Kafka the communication between the clients and the servers is done with a simple, high-performance, language agnostic TCP protocol. This protocol is versioned and maintains backwards compatibility with older version. We provide a Java client for Kafka in this workshop, but clients are available in many languages. Official clients managed and supported by the open source are:

- C/C++
- Python
- Go (AKA golang)

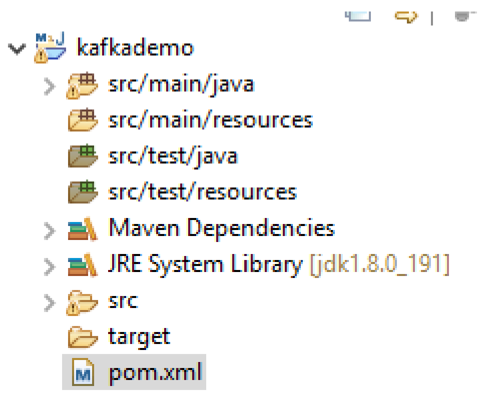
- Erlang
- .NET
- Clojure
- Ruby
- Node.js
- Proxy (HTTP REST, etc)
- Perl
- stdin/stdout
- PHP
- Rust
- Alternative Java
- Storm
- Scala DSL
- Clojure
- Swift

Kafka Project using Eclipse

git code available at:

Project Settings

In this workshop we will use eclipse and maven. But you are free to pick any other of your favourite IDE of equivalent capabilities. We have created a Java Maven Project as shown below:



```
kafkademo
  src/main/java
  src/main/resources
  src/test/java
  src/test/resources
  Maven Dependencies
  JRE System Library [jdk1.8.0_191]
  src
  target
  pom.xml
```

The pom file is filled in as below

```
<project  .   . >
      <modelVersion>4.0.0</modelVersion>
      <groupId>sg.edu.iss</groupId>
      <artifactId>kafkademo</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <dependencies>
      <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-
clients -->
            <dependency>
```

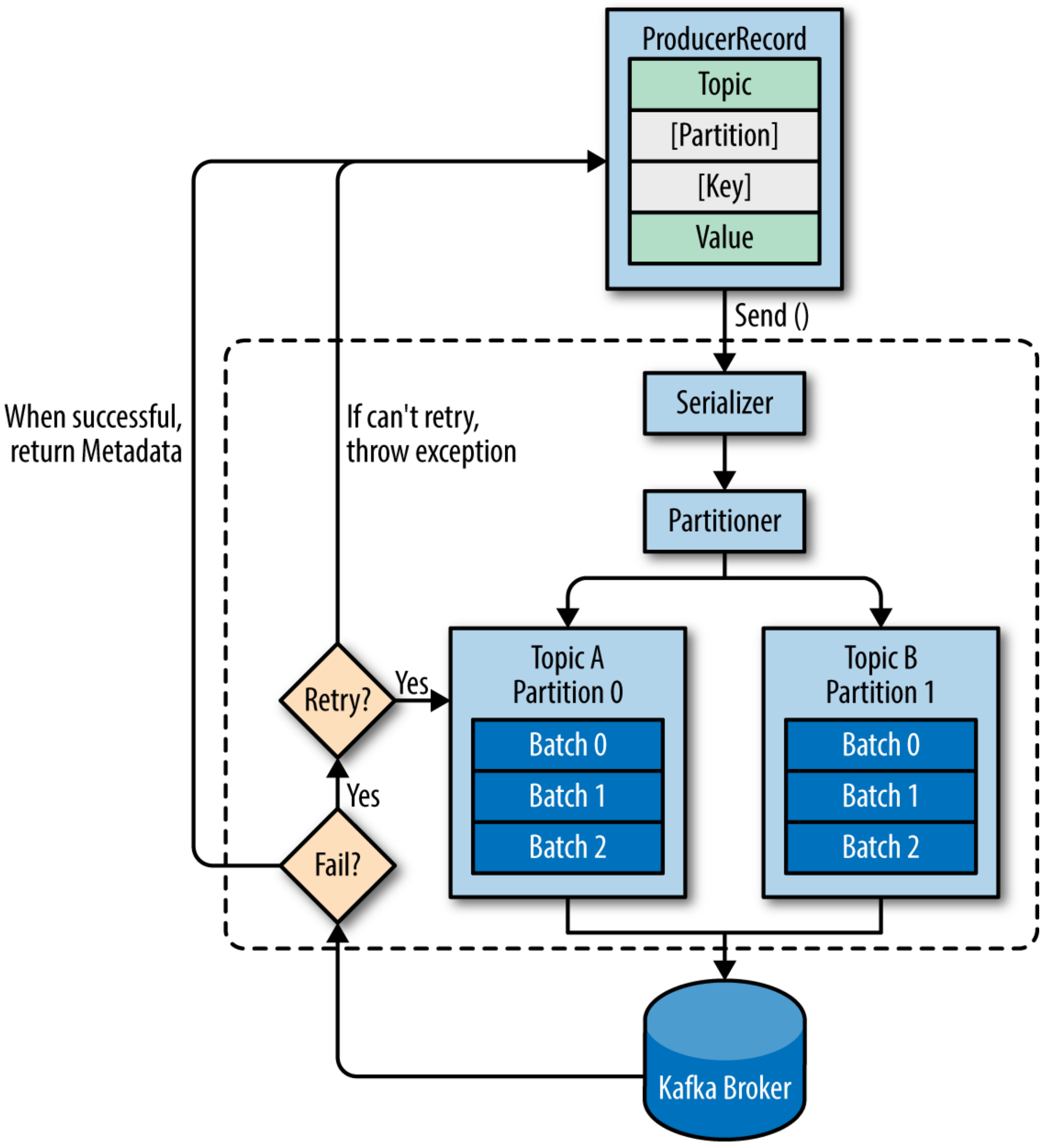
```
                  <groupId>org.apache.kafka</groupId>
                  <artifactId>kafka-clients</artifactId>
                  <version>2.4.0</version>
            </dependency>
      <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-
streams -->
            <dependency>
                  <groupId>org.apache.kafka</groupId>
                  <artifactId>kafka-streams</artifactId>
                  <version>2.4.0</version>
            </dependency>

      <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-simple -->
            <dependency>
                  <groupId>org.slf4j</groupId>
                  <artifactId>slf4j-simple</artifactId>
                  <version>1.7.30</version>
            </dependency>

      <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
            <dependency>
                  <groupId>com.google.code.gson</groupId>
                  <artifactId>gson</artifactId>
                  <version>2.8.5</version>
            </dependency>
      </dependencies>

</project>
```

Producer

Apache Kafka ships with built-in client APIs that developers can use when developing applications that interact with Kafka.

There are many reasons an application might need to write messages to Kafka: recording user activities for auditing or analysis, recording metrics, storing log messages, recording information from smart appliances, communicating asynchronously with other applications, buffering information before writing to a database, and much more. While the producer APIs are very simple, there is a bit more that goes on under the hood of the producer when we send data. The diagram below explains how data is sent to Kafka.

We start producing messages to Kafka by creating a `ProducerRecord`, which must include the topic we want to send the record to and a value. Optionally, we can also specify a key and/or a partition. Once we send the `ProducerRecord`, the first thing the producer will do is serialize the key and value objects to `ByteArrays` so they can be sent over the network. Next, the data is sent to a partitioner. Once a partition is selected, the producer knows which topic and partition the record will go to. When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a `RecordMetadata` object with the topic, partition, and the offset of the record within the partition.



Producer Code

The first step in writing messages to Kafka is to create a producer object with the properties we want to pass to the producer. A Kafka producer has three mandatory properties:

1. `bootstrap.servers` List of `host:port` pairs of brokers that the producer will use to establish initial connection to the Kafka cluster.
2. `key.serializer` Name of a class that will be used to serialize the keys of the records we will produce to Kafka.
3. `value.serializer` Name of a class that will be used to serialize the values of the records we will produce to Kafka.

The following code snippet shows how to create a new producer by setting just the mandatory parameters and using defaults for everything else:

```
Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
"broker1:9092,broker2:9092");
kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");
producer = new KafkaProducer<String, String>(kafkaProps);
```

Sending a Message to Kafka

The simplest way to send a message is as follows:

```
ProducerRecord<String, String> record =     new
   ProducerRecord<>("CustomerCountry", "Precision Products",
    "France");
try {
    producer.send(record);
} catch (Exception e) {
    e.printStackTrace();
}
```

To send message synchronously we issue the command:

```
producer.send(record).get();
```

To send message asynchronously we issue the command:

```
producer.send(record, new DemoProducerCallback());
```

Feel free to explore configuration parameters such as ack, compression, retries, batch size, linger, client id, request per connection, timeout and buffer / block settings. Also explore retry guarantees.

Consumer Code

There is a need to scale consumption from topics. Just like multiple producers can write to the same topic, we need to allow multiple consumers to read from the same topic, splitting the data between them.

The first step to start consuming records is to create a `KafkaConsumer` instance. Creating a `KafkaConsumer` is very similar to creating a `KafkaProducer`—we create a Java `Properties` instance with the properties we want to pass to the consumer. To start we just need to use the three mandatory properties:

`bootstrap.servers`, `key.deserializer`, and `value.deserializer`. The following code snippet shows how to create a `KafkaConsumer`:

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer =
    new KafkaConsumer<String, String>(props);
```

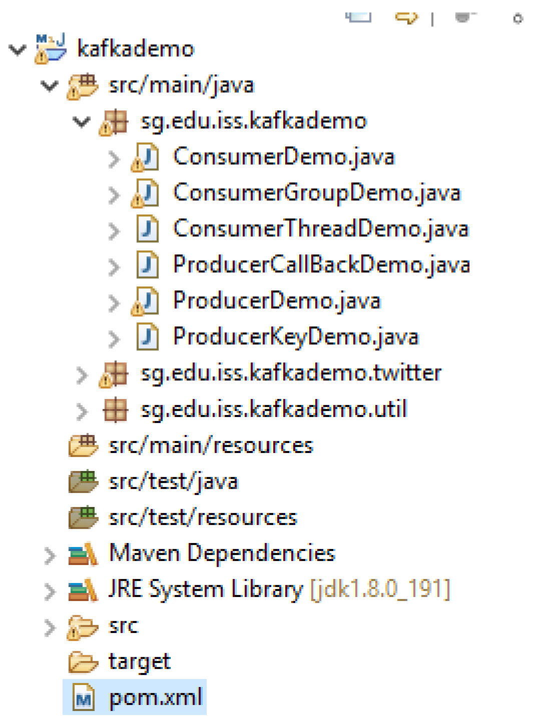
Once we create a consumer, the next step is to subscribe to one or more topics. The `subcribe()` method takes a list of topics as a parameter, so it's pretty simple to use:

```
consumer.subscribe(Collections.singletonList("customerCountries
"));
```

At the heart of the consumer API is a simple loop for polling the server for more data. Once the consumer subscribes to topics, the poll loop handles all details of coordination, partition rebalances, heartbeats, and data fetching, leaving the developer with a clean API that simply returns available data from the assigned partitions. The main body of a consumer will look as follows:

```
try {
    while (true) {
        ConsumerRecords<String, String> records =
consumer.poll(100);
      for (ConsumerRecord<String, String> record : records)
       {
           log.debug("topic = %s, partition = %d, offset = %d,"
                  customer = %s, country = %s\n",
                  record.topic(), record.partition(),
                  record.offset(),
                  record.key(), record.value());
              int updatedCount = 1;
              if (custCountryMap.countainsKey(record.value())) {
                  updatedCount =
                    custCountryMap.get(record.value()) + 1;
              }
              custCountryMap.put(record.value(), updatedCount)
              JSONObject json = new JSONObject(custCountryMap);
              System.out.println(json.toString(4))
        }
    }
} finally {
    consumer.close();
}
```

Additionally, we can develop custom serializers and data formats. Look into sample producer codes provided in the default package:



```
kafkademo
  src/main/java
    sg.edu.iss.kafkademo
      ConsumerDemo.java
      ConsumerGroupDemo.java
      ConsumerThreadDemo.java
      ProducerCallBackDemo.java
      ProducerDemo.java
      ProducerKeyDemo.java
    sg.edu.iss.kafkademo.twitter
    sg.edu.iss.kafkademo.util
  src/main/resources
  src/test/java
  src/test/resources
  Maven Dependencies
  JRE System Library [jdk1.8.0_191]
  src
  target
  pom.xml
```

Summary

We began this workshop with a complete walk through of Kafka console commands. We explored a simple UI tool. We then created a simple example of a producer. We added to the simple example by adding error handling and experimenting with synchronous and asynchronous producing. We then explored the most important producer configuration parameters and saw how they modify the behaviour of the producers. Then we proceeded with a practical example of a consumer subscribing to a topic and continuously reading events. We then looked into the most important consumer configuration parameters and how they affect consumer behaviour.

You have accomplished enough learning on this new distributed scalable messaging platform Kafka. Time to take a well-deserved break.