0.8.0 Producer Example

**** PLEASE NOTE **** The recommended producer is from latest stable release using the new Java producer http://kafka.apache.org/082/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html

Once you have confirmed you have a basic Kafka cluster setup (see 0.8 Quick Start) it is time to write some code!

Producers

The Producer class is used to create new messages for a specific Topic and optional Partition.

If using Java you need to include a few packages for the Producer and supporting classes:

```
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
```

The first step in your code is to define properties for how the Producer finds the cluster, serializes the messages and if appropriate directs the message to a specific Partition.

These properties are defined in the standard Java Properties object:

```
Properties props = new Properties();

props.put("metadata.broker.list", "broker1:9092,broker2:9092");
props.put("serializer.class", "kafka.serializer.StringEncoder");
props.put("partitioner.class", "example.producer.SimplePartitioner");
props.put("request.required.acks", "1");

ProducerConfig config = new ProducerConfig(props);
```

The first property, “metadata.broker.list” defines where the Producer can find a one or more Brokers to determine the Leader for each topic. This does not need to be the full set of Brokers in your cluster but should include at least two in case the first Broker is not available. No need to worry about figuring out which Broker is the leader for the topic (and partition), the Producer knows how to connect to the Broker and ask for the meta data then connect to the correct Broker.

The second property “serializer.class” defines what Serializer to use when preparing the message for transmission to the Broker. In our example we use a simple String encoder provided as part of Kafka. Note that the encoder must accept the same type as defined in the KeyedMessage object in the next step.

It is possible to change the Serializer for the Key (see below) of the message by defining "key.serializer.class" appropriately. By default it is set to the same value as "serializer.class".

The third property "partitioner.class" defines what class to use to determine which Partition in the Topic the message is to be sent to. This is optional, but for any non-trivial implementation you are going to want to implement a partitioning scheme. More about the implementation of this class later. If you include a value for the key but haven't defined a partitioner.class Kafka will use the default partitioner. If the key is null, then the Producer will assign the message to a random Partition.

The last property "request.required.acks" tells Kafka that you want your Producer to require an acknowledgement from the Broker that the message was received. Without this setting the Producer will 'fire and forget' possibly leading to data loss. Additional information can be found here

Next you define the Producer object itself:

```
Producer<String, String> producer = new Producer<String, String>(config);
```

Note that the Producer is a Java Generic and you need to tell it the type of two parameters. The first is the type of the Partition key, the second the type of the message. In this example they are both Strings, which also matches to what we defined in the Properties above.

Now build your message:

```
Random rnd = new Random();

long runtime = new Date().getTime();

String ip = “192.168.2.” + rnd.nextInt(255);

String msg = runtime + “,www.example.com,” + ip;
```

In this example we are faking a message for a website visit by IP address. First part of the comma-separated message is the timestamp of the event, the second is the website and the third is the IP address of the requester. We use the Java Random class here to make the last octet of the IP vary so we can see how Partitioning works.

Finally write the message to the Broker:

```
KeyedMessage<String, String> data = new KeyedMessage<String, String>("page_visits", ip, msg);

producer.send(data);
```

The “page_visits” is the Topic to write to. Here we are passing the IP as the partition key. Note that if you do not include a key, even if you've defined a partitioner class, Kafka will assign the message to a random partition.

Full Source:

```
import java.util.*;

import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class TestProducer {
    public static void main(String[] args) {
        long events = Long.parseLong(args[0]);
        Random rnd = new Random();

        Properties props = new Properties();
        props.put("metadata.broker.list", "broker1:9092,broker2:9092 ");
        props.put("serializer.class", "kafka.serializer.StringEncoder");
        props.put("partitioner.class", "example.producer.SimplePartitioner");
        props.put("request.required.acks", "1");

        ProducerConfig config = new ProducerConfig(props);

        Producer<String, String> producer = new Producer<String, String>(config);

        for (long nEvents = 0; nEvents < events; nEvents++) {
               long runtime = new Date().getTime();
               String ip = “192.168.2.” + rnd.nextInt(255);
               String msg = runtime + “,www.example.com,” + ip;
               KeyedMessage<String, String> data = new KeyedMessage<String, String>("page_visits", ip, msg);
               producer.send(data);
        }
        producer.close();
    }
}
```

Partitioning Code:

```
import kafka.producer.Partitioner;
import kafka.utils.VerifiableProperties;

public class SimplePartitioner implements Partitioner {
    public SimplePartitioner (VerifiableProperties props) {

    }

    public int partition(Object key, int a_numPartitions) {
        int partition = 0;
        String stringKey = (String) key;
        int offset = stringKey.lastIndexOf('.');
        if (offset > 0) {
           partition = Integer.parseInt( stringKey.substring(offset+1)) % a_numPartitions;
        }
       return partition;
  }

}
```

The logic takes the key, which we expect to be the IP address, finds the last octet and does a modulo operation on the number of partitions defined within Kafka for the topic. The benefit of this partitioning logic is all web visits from the same source IP end up in the same Partition. Of course so do other IPs, but your consumer logic will need to know how to handle that.

Before running this, make sure you have created the Topic page_visits. From the command line:

```
bin/kafka-create-topic.sh --topic page_visits --replica 3 --zookeeper localhost:2181 --partition 5
```

Make sure you include a --partition option so you create more than one.

Now compile and run your Producer and data will be written to Kafka.

To confirm you have data, use the command line tool to see what was written:

```
 bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic page_visits --from-beginning
```

Maven

```
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka_2.9.2</artifactId>
      <version>0.8.1.1</version>
      <scope>compile</scope>
      <exclusions>
        <exclusion>
          <artifactId>jmxri</artifactId>
          <groupId>com.sun.jmx</groupId>
        </exclusion>
        <exclusion>
          <artifactId>jms</artifactId>
          <groupId>javax.jms</groupId>
        </exclusion>
        <exclusion>
          <artifactId>jmxtools</artifactId>
          <groupId>com.sun.jdmk</groupId>
        </exclusion>
      </exclusions>
    </dependency>
```