

**howtoprogram** 6:05 am on January 5, 2018

Tags: Apache Kafka ( 11 ), Big Data ( 3 )

## Apache Kafka Java Client API Example

In this article, I'd like to show you how to create a producer and consumer by using **Apache Kafka Java client API**.

### 1. Overview

Apache Kafka has some built-in client tools to produce and consume messages against Apache Kafka broker. However, in term of messaging, both of them are mainly used for simple operations at server (broker) side where you may only need to:

- Create Kafka Topic
- List all topics exist in the brokers.
- Produce and consume some tests data against some topics in the cluster.

In similar to many other messaging systems, Apache Kafka provides many types of client APIs in different languages such as Java, Python, Ruby, Go..which will facilitate users in working with Kafka clusters. In this article, we will focus on Java Client API.



### 2. Prerequisites

- Apache Kafka single broker installed on local machine or remote. If it's not ready, you can install by yourself by taking a look at [install Apache Kafka on Linux](#) or [install Apache Kafka on Windows](#).
- JDK 7/8 installed on your development PC.
- Eclipse 4 (I am using Eclipse Mars 4.5)
- Maven 3

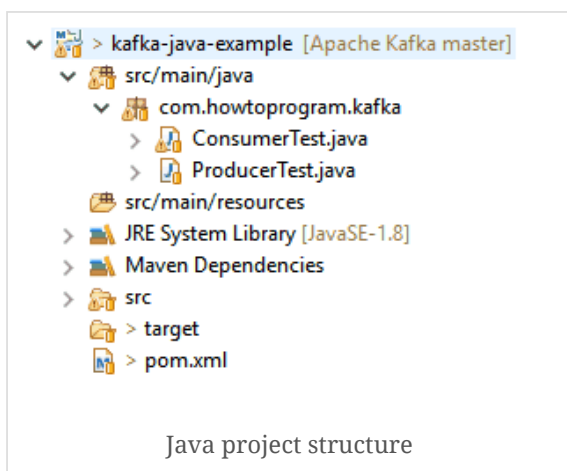
### 3. Project structure

The example source code is available on my [Github](#) project. After having the source code, we can import the source code into Eclipse and run the test.

To import:

- Menu *File* → *Import* → *Maven* → *Existing Maven Projects*
- Browse to your source code location
- Click *Finish* button to finish the importing

Here is the project structure in my Eclipse:



## 4. Maven pom.xml file

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.howtoprogram</groupId>
  <artifactId>kafka-java-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Kafka-Java-Example</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>0.9.0.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

We use the Kafka-clients-0.9.0.1 library for this example. The Java compiler 1.8

## 5. Source Code

### 5.1 Producer source code

```
package com.howtoprogram.kafka;

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class ProducerTest {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "192.168.33.10:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        Producer<String, String> producer = null;
        try {
            producer = new KafkaProducer<>(props);
            for (int i = 0; i < 100; i++) {
                String msg = "Message " + i;
                producer.send(new ProducerRecord<String, String>("HelloKafkaTopic", msg));
                System.out.println("Sent: " + msg);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            producer.close();
        }
    }
}
```

To create a producer, we just need to create a new instance of the class *KafkaProducer* and provide it a list of properties to initialize the instance:

```
producer = new KafkaProducer<>(props);
```

Below are some essential properties used in this article:

### ***bootstrap.servers***

```
props.put("bootstrap.servers", "192.168.33.10:9092");
```

***bootstrap.servers*** is the IP addresses of Kafka cluster. If you have more than 1 broker, you can put all separated by commas. For ex: 192.168.33.10:9092, 192.168.33.10:9093

Currently, my broker is installed on another PC which has IP address is **192.168.33.10** and listened on the port **9092**. If it's different with your environment, you should change that info to be matched with yours.

### ***key.serializer*** and ***value.serializer***.

```
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

Kafka message sent to Kafka cluster is combined with key(optional) and value which can be any data type. However, we will need to specify how Kafka producer should serialize those data types into binary before sending to Kafka cluster.

In this example, we will produce text messages to Kafka cluster. Therefore, we use *StringSerializer* which is a built-in serializer of Kafka client to serialize strings into binary.

Produce message to the Kafka cluster.

```
for (int i = 0; i < 100; i++) {  
    String msg = "Message " + i;  
    producer.send(new ProducerRecord<String, String>("HelloKafkaTopic", msg));  
    System.out.println("Sent:" + msg);  
}
```

Above source code will produce 100 messages which value are “Message 1”, “Message 2”,... “Message 99” to the ***HelloKafkaTopic*** topic.

## 5.2. Consumer source code

```
package com.howtoprogram.kafka;  
  
import java.util.Arrays;  
import java.util.Properties;  
  
import org.apache.kafka.clients.consumer.ConsumerRecord;  
import org.apache.kafka.clients.consumer.ConsumerRecords;  
import org.apache.kafka.clients.consumer.KafkaConsumer;  
  
public class ConsumerTest {  
  
    public static void main(String[] args) {  
        Properties props = new Properties();  
        props.put("bootstrap.servers", "192.168.33.10:9092");  
        props.put("group.id", "group-1");  
        props.put("enable.auto.commit", "true");  
        props.put("auto.commit.interval.ms", "1000");  
        props.put("auto.offset.reset", "earliest");  
        props.put("session.timeout.ms", "30000");  
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
  
        KafkaConsumer<String, String> kafkaConsumer = new KafkaConsumer<>(props);  
        kafkaConsumer.subscribe(Arrays.asList("HelloKafkaTopic"));
```

```
while (true) {
    ConsumerRecords<String, String> records = kafkaConsumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, value = %s", record.offset(), record.value());
        System.out.println();
    }
}
```

To create Kafka consumer, we also just need to create an instance of **KafkaConsumer** class and provide it a list of properties. Some essential properties for the consumer include:

### ***bootstrap.servers***

The Kafka cluster IP addresses. In the same with Producer, we can put a list of brokers IP addresses, ports separated by commas in here.

### ***group.id***

It is the group id of processes which the consumer belonged to.

### ***key.deserializer and value.deserializer***

They are deserializers used by Kafka consumer to deserialize the binary data received from Kafka cluster to our desire data types. In this example, because the producer produces string message, our consumer use **StringDeserializer** which is a built-in deserializer of Kafka client API to deserialize the binary data to the string.

Our consumer subscribes the topic: **HelloKafkaTopic**

```
kafkaConsumer.subscribe(Arrays.asList("HelloKafkaTopic"));
```

Consume messages from the cluster

```
while (true) {
    ConsumerRecords<String, String> records = kafkaConsumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset = %d, value = %s", record.offset(), record.value());
        System.out.println();
    }
}
```

Our consumer polls the clusters for the new messages. The method takes a timeout =100 milliseconds which is the time the consumers wait if there is no message from the cluster. Once found the messages, we will print their offsets, values line by line.

## 6. Execute the source code

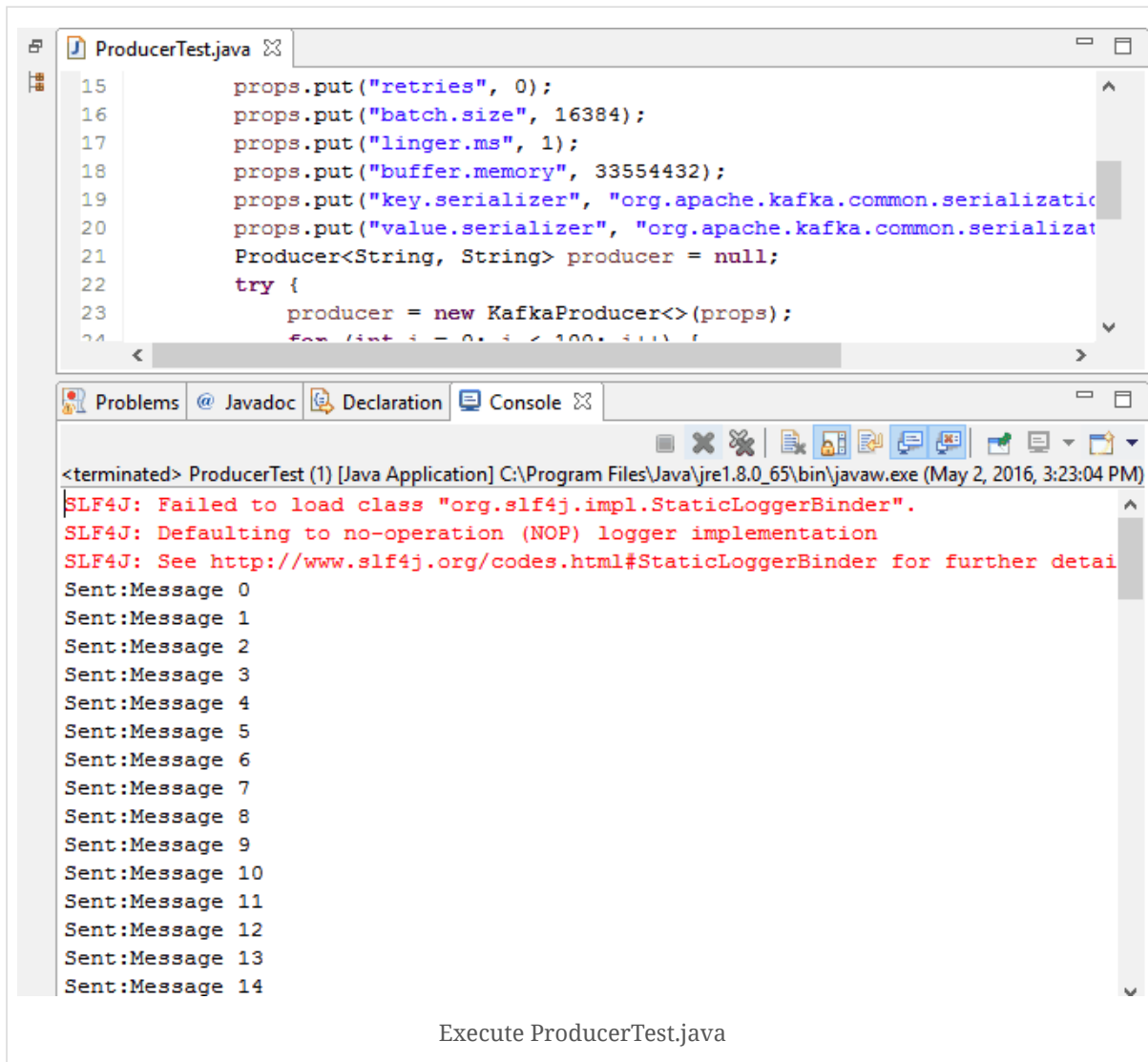
### 6.1. ProducerTest.java

From the Eclipse:

Open the **ProducerTest.java**

**Right-click → Run as → Java Application**

You will see the output on the console as below:



The screenshot shows the Eclipse IDE with the `ProducerTest.java` file open. The code defines properties for a Kafka producer and sends 15 messages. The console output shows the execution of the program, including an SLF4J warning and the messages being sent.

```
15 props.put("retries", 0);
16 props.put("batch.size", 16384);
17 props.put("linger.ms", 1);
18 props.put("buffer.memory", 33554432);
19 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
20 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
21 Producer<String, String> producer = null;
22 try {
23     producer = new KafkaProducer<>(props);
24     for (int i = 0; i < 100; i++) {
```

Console Output:

```
<terminated> ProducerTest (1) [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (May 2, 2016, 3:23:04 PM)
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 0
Sent:Message 1
Sent:Message 2
Sent:Message 3
Sent:Message 4
Sent:Message 5
Sent:Message 6
Sent:Message 7
Sent:Message 8
Sent:Message 9
Sent:Message 10
Sent:Message 11
Sent:Message 12
Sent:Message 13
Sent:Message 14
```

Execute ProducerTest.java

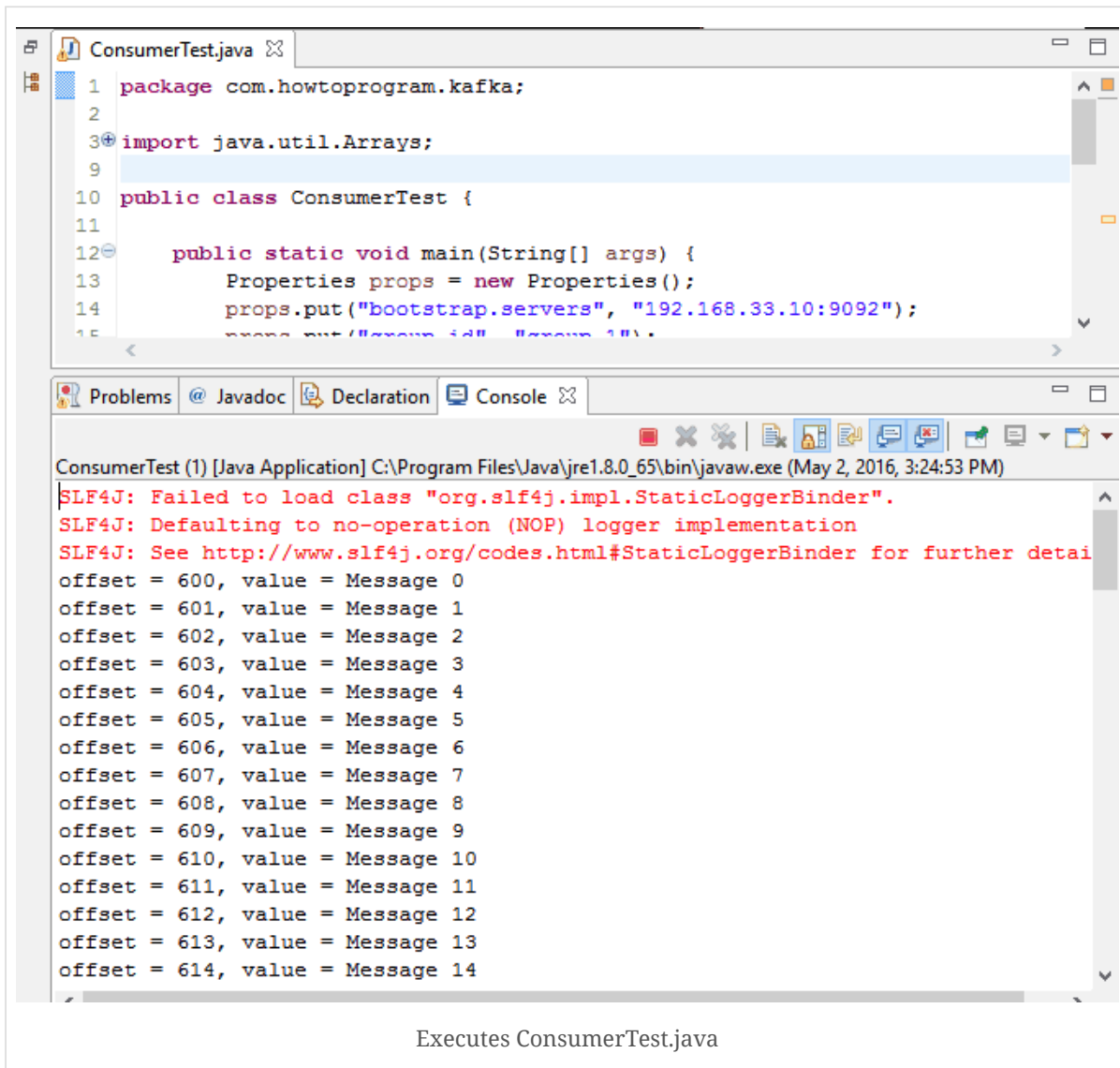
## 6.2. ConsumerTest.java

From the Eclipse:

Open the *ConsumerTest.java*

Right click → Run as → Java Application

You will see the output on the console as below:



The screenshot shows an IDE window with the file `ConsumerTest.java` open. The code defines a `ConsumerTest` class with a `main` method that sets up properties for a Kafka consumer, including `bootstrap.servers` and `zookeepers`. Below the code editor, the `Console` tab is active, displaying the output of the Java application. The output shows an SLF4J warning about a missing `StaticLoggerBinder` class, followed by a series of log messages indicating that the consumer is successfully receiving and processing 15 messages (offsets 600 to 614).

```
1 package com.howtoprogram.kafka;
2
3 import java.util.Arrays;
4
5
6
7
8
9
10 public class ConsumerTest {
11
12     public static void main(String[] args) {
13         Properties props = new Properties();
14         props.put("bootstrap.servers", "192.168.33.10:9092");
15         props.put("zookeepers", "192.168.33.10:2181");
```

ConsumerTest (1) [Java Application] C:\Program Files\Java\jre1.8.0\_65\bin\javaw.exe (May 2, 2016, 3:24:53 PM)

```
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
offset = 600, value = Message 0
offset = 601, value = Message 1
offset = 602, value = Message 2
offset = 603, value = Message 3
offset = 604, value = Message 4
offset = 605, value = Message 5
offset = 606, value = Message 6
offset = 607, value = Message 7
offset = 608, value = Message 8
offset = 609, value = Message 9
offset = 610, value = Message 10
offset = 611, value = Message 11
offset = 612, value = Message 12
offset = 613, value = Message 13
offset = 614, value = Message 14
```

Executes ConsumerTest.java

## 7. Conclusion

This article shows you how to create a very simple producer which produces string messages and a consumer to consume those messages. Both are implemented with Apache Kafka 0.9 Java client API. This example is very basic and simple. I hope it will help those who want to look for some basic tutorial to getting started with Apache Kafka especially version 1.0.

Below are the articles related to Apache Kafka topic. If you're interested in them, you can refer to the following links:

[Apache Kafka Tutorial](#)

[Getting started with Apache Kafka 0.9](#)

[Apache Kafka Command Line Interface](#)

[Create Multi-threaded Apache Kafka Consumer](#)

[Using Apache Kafka Docker](#)

[Apache Kafka Command Line Interface](#)

[Apache Kafka Connect Example](#)

[Apache Kafka Connect MQTT Source Tutorial](#)

[Apache Flume Kafka Source And HDFS Sink Tutorial](#)

[Spring Kafka Tutorial](#)



Set IT Service Prices that  
**Maximize Your Profit**



AdChoices 

Start  
Calculating  
>>