

# Lab 5.1: Creating Advanced Kafka Producers in Java

Welcome to the session 5 lab 1. The work for this lab is done in `~/kafka-training/lab5.1`.

In this lab, you are going to create an advanced Java Kafka consumer.

Please refer to the [Kafka course notes](#) for any updates or changes to this lab.

Find the latest version of this lab [here](#).

## Kafka Producers

A producer is a type of Kafka client that publishes records to Kafka cluster. The Kafka client API for Producers are thread safe. A Kafka *Producer* has a pool of buffer that holds to-be-sent records. The producer has background, I/O threads for turning records into request bytes and transmitting requests to Kafka cluster. The producer must be closed to not leak resources, i.e., connections, thread pools, buffers.

## Kafka Producer Send, Acks and Buffers

The Kafka Producer has a ***send()*** method which is asynchronous. Calling the send method adds the record to the output buffer and returns right away. The buffer is used to batch records for efficient IO and compression. The Kafka Producer configures acks to control record durability. The "all" acks setting ensures full commit of record to all replicas and is most durable and least fast setting. The Kafka Producer can automatically retry failed requests. The Producer has buffers of unsent records per topic partition (sized at ***batch.size***).

## Lab Creating an advanced Kafka Producer

### Stock Price Producer

The Stock Price Producer example has the following classes:

- StockPrice - holds a stock price has a name, dollar, and cents
- StockPriceKafkaProducer - Configures and creates KafkaProducer<String, StockPrice>, StockSender list, ThreadPool (ExecutorService), starts StockSender runnable into thread pool
- StockAppConstants - holds topic and broker list
- StockPriceSerializer - can serialize a StockPrice into byte[]
- StockSender - generates somewhat random stock prices for a given StockPrice name, Runnable, 1 thread per StockSender and shows using KafkaProducer from many threads

## StockPrice

The StockPrice is a simple domain object that holds a stock price has a name, dollar, and cents. The StockPrice knows how to convert itself into a JSON string.

`~/kafka-training/lab5.1/src/main/java/com/cloudurable/kafka/model/StockPrice.java`

### Kafka Producer: StockPrice

```
package com.cloudurable.kafka.producer.model;

import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;

    public StockPrice(final String json) {
```

```

        this(JsonFactory.fromJson(json, StockPrice.class));
    }

    public StockPrice() {
        dollars = 0;
        cents = 0;
        name = "";
    }

    public StockPrice(final String name, final int dollars, final int cents) {
        this.dollars = dollars;
        this.cents = cents;
        this.name = name;
    }

    public StockPrice(final StockPrice stockPrice) {
        this.cents = stockPrice.cents;
        this.dollars = stockPrice.dollars;
        this.name = stockPrice.name;
    }

    public int getDollars() {
        return dollars;
    }

    public int getCents() {
        return cents;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "StockPrice{" +
            "dollars=" + dollars +
            ", cents=" + cents +
            ", name='" + name + '\'' +
            '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        StockPrice that = (StockPrice) o;

```

```

        if (dollars != that.dollars) return false;
        if (cents != that.cents) return false;
        return name != null ? name.equals(that.name) : that.name == null;
    }

    @Override
    public int hashCode() {
        int result = dollars;
        result = 31 * result + cents;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }

    public String toJson() {
        return "{" +
            "\"dollars\": " + dollars +
            ", \"cents\": " + cents +
            ", \"name\": \"" + name + "\"" +
            "}";
    }
}

```

StockPrice is just a POJO.

## **ACTION - EDIT src/main/java/com/cloudurable/kafka/model/StockPrice.java and follow the instructions.**

### **StockPriceKafkaProducer**

StockPriceKafkaProducer import classes and sets up a logger. It has a createProducer method to create a KafkaProducer instance. It has a setupBootstrapAndSerializers to initialize bootstrap servers, client id, key serializer and custom serializer (StockPriceSerializer). It has a main() method that creates the producer, creates a StockSender list passing each instance the producer, and it creates a thread pool, so every stock sender gets its own thread, and then it runs each stockSender in its own thread using the thread pool.

~/kafka-training/lab5.1/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

### **Kafka Producer: StockPriceKafkaProducer imports, createProducer**

```

package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.StockAppConstants;
import com.cloudurable.kafka.producer.model.StockPrice;
import io.advantageous.boon.core.Lists;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

```

```
import java.util.concurrent.TimeUnit;

public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
        createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        return new KafkaProducer<>(props);
    }
    ...
}
```

The above code imports Kafka classes and sets up the logger and calls `createProducer` to create a `KafkaProducer`. The `createProducer()` calls `setupBootstrapAndSerializers()`.

## ACTION - EDIT

**src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java**  
**and follow the instructions in createProducer.**

~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

**Kafka Producer: StockPriceKafkaProducer imports, createProducer**

```
public class StockPriceKafkaProducer {
    private static void setupBootstrapAndSerializers(Properties props) {
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "StockPriceKafkaProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());

        //Custom Serializer - config "value.serializer"
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StockPriceSerializer.class.getName());
    }
}
```

The `setupBootstrapAndSerializers` method initializes bootstrap servers, client id, key serializer and custom serializer (`StockPriceSerializer`). The `StockPriceSerializer` will serialize `StockPrice` into bytes.

## ACTION - EDIT

**src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java**  
**and follow the instructions in setupBootstrapAndSerializers.**

~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java

**Kafka Producer: StockPriceKafkaProducer.main - start thread pool**

```
public class StockPriceKafkaProducer {
    ...
}
```

```

public static void main(String... args) throws Exception {
    //Create Kafka Producer
    final Producer<String, StockPrice> producer = createProducer();
    //Create StockSender list
    final List<StockSender> stockSenders = getStockSenderList(producer);

    //Create a thread pool so every stock sender gets it own.
    final ExecutorService executorService =
        Executors.newFixedThreadPool(stockSenders.size());

    //Run each stock sender in its own thread.
    stockSenders.forEach(executorService::submit);

}
...
}

```

The `StockPriceKafkaProducer` main method creates a Kafka producer, then creates `StockSender` list passing each instance the producer. It then creates a thread pool ( `executorService` ) and runs each `StockSender` , which is runnable, in its own thread from the thread pool.

## ACTION - EDIT

**src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java**  
**and follow the instructions in main.**

**~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java**

**Kafka Producer: `StockPriceKafkaProducer.getStockSenderList` - create list of `StockSenders`**

```

private static List<StockSender> getStockSenderList (
    final Producer<String, StockPrice> producer) {
    return Lists.list(
        new StockSender(StockAppConstants.TOPIC,
            new StockPrice("IBM", 100, 99),
            new StockPrice("IBM", 50, 10),
            producer,
            1, 10
        ),
        new StockSender(
            StockAppConstants.TOPIC,
            new StockPrice("SUN", 100, 99),
            new StockPrice("SUN", 50, 10),
            producer,
            1, 10
        ),
        ...,
        new StockSender(
            StockAppConstants.TOPIC,
            new StockPrice("FFF", 100, 99),
            new StockPrice("FFF", 50, 10),
            producer,
            1, 10
        )
    );
}

```

```

        )
    };
}

```

The `getStockSenderList` of `StockPriceKafkaProducer` just creates a list of `StockSenders`.

## **ACTION - EDIT**

**`src/main/java/com/cloudurable/kafka/producer/StockPriceKafkaProducer.java` and follow the instructions in `getStockSenderList`.**

### **StockPriceSerializer**

The `StockPriceSerializer` converts a `StockPrice` into a byte array.

`~/kafka-training/lab5.1/src/main/java/com/cloudurable/kafka/producer/StockPriceSerializer.java`

**Kafka Producer: `StockPriceSerializer` - convert `StockPrice` into a byte array**

```

package com.cloudurable.kafka.producer;
import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Serializer;
import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockPriceSerializer implements Serializer<StockPrice> {

    @Override
    public byte[] serialize(String topic, StockPrice data) {
        return data.toJson().getBytes(StandardCharsets.UTF_8);
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}

```

Notice the `StockPriceSerializer` converts a `StockPrice` into a byte array by calling `StockPrice.toJson`.

## **ACTION - EDIT**

**`src/main/java/com/cloudurable/kafka/producer/StockPriceSerializer.java` and follow the instructions.**

### **StockAppConstants**

The `StockAppConstants` defines a few constants, namely, topic name and a comma delimited list of bootstrap Kafka brokers.

`~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/StockAppConstants.java`

**Kafka Producer: `StockAppConstants` defines constants**

```
package com.cloudurable.kafka;

public class StockAppConstants {
    public final static String TOPIC = "stock-prices";
    public final static String BOOTSTRAP_SERVERS =
        "localhost:9092,localhost:9093,localhost:9094";
}
```

## StockSender

The `StockSender` uses the Kafka Producer we created earlier. The `StockSender` generates random stock prices for a given `StockPrice` name. The `StockSender` is `Runnable` and runs in its own thread. There is one thread per `StockSender`. The `StockSender` is used to show using `KafkaProducer` from many threads. The `StockSender` delays a random time duration between `delayMin` and `delayMax`, then sends a random `StockPrice` between `stockPriceHigh` and `stockPriceLow`.

~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

### Kafka Producer: StockSender imports, Runnable

```
package com.cloudurable.kafka.producer;

import com.cloudurable.kafka.producer.model.StockPrice;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Date;
import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class StockSender implements Runnable{
    ...
}
```

The `StockSender` imports Kafka Producer, `ProducerRecord`, `RecordMetadata`, and `StockPrice`. It implements `Runnable`, and can be submitted to an `ExecutionService` (thread pool).

~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

### Kafka Producer: StockSender fields

```
public class StockSender implements Runnable{

    private final StockPrice stockPriceHigh;
    private final StockPrice stockPriceLow;
    private final Producer<String, StockPrice> producer;
    private final int delayMinMs;
    private final int delayMaxMs;
    private final Logger logger = LoggerFactory.getLogger(StockSender.class);
    private final String topic;

    public StockSender(final String topic, final StockPrice stockPriceHigh,
```

```

        final StockPrice stockPriceLow,
        final Producer<String, StockPrice> producer,
        final int delayMinMs,
        final int delayMaxMs) {
    this.stockPriceHigh = stockPriceHigh;
    this.stockPriceLow = stockPriceLow;
    this.producer = producer;
    this.delayMinMs = delayMinMs;
    this.delayMaxMs = delayMaxMs;
    this.topic = topic;
}
}

```

The `StockSender` takes a topic, high & low stockPrice, producer, and delay min & max.

~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

#### Kafka Producer: StockSender run method

```

public class StockSender implements Runnable{

    ...
    public void run() {
        final Random random = new Random(System.currentTimeMillis());
        int sentCount = 0;

        while (true) {
            sentCount++;
            final ProducerRecord <String, StockPrice> record =
                createRandomRecord(random);
            final int delay = randomIntBetween(random, delayMaxMs, delayMinMs);

            try {
                final Future<RecordMetadata> future = producer.send(record);
                if (sentCount % 100 == 0) {displayRecordMetaData(record, future);}
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                if (Thread.interrupted()) {
                    break;
                }
            } catch (ExecutionException e) {
                logger.error("problem sending record to producer", e);
            }
        }
    }
}

```

The `StockSender` run methods in a forever loop creates random record, sends the record, waits random time, and then repeats.

~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java

#### Kafka Producer: StockSender createRandomRecord

```

public class StockSender implements Runnable{

    ...

```



```
private final int randomIntBetween(final Random random,
                                   final int max,
                                   final int min) {
    return random.nextInt(max - min + 1) + min;
}

private ProducerRecord<String, StockPrice> createRandomRecord(
    final Random random) {

    final int dollarAmount = randomIntBetween(random,
        stockPriceHigh.getDollars(), stockPriceLow.getDollars());

    final int centAmount = randomIntBetween(random,
        stockPriceHigh.getCents(), stockPriceLow.getCents());

    final StockPrice stockPrice = new StockPrice(
        stockPriceHigh.getName(), dollarAmount, centAmount);

    return new ProducerRecord<>(topic, stockPrice.getName(),
        stockPrice);
}
}
```

The StockSender createRandomRecord method uses randomIntBetween. The createRandomRecord creates StockPrice and then wraps StockPrice in ProducerRecord.

```
~/kafka-training/lab5.1/solution/src/main/java/com/cloudurable/kafka/producer/StockSender.java
```

### Kafka Producer: StockSender displayRecordMetaData

```
public class StockSender implements Runnable{

    ...

    private void displayRecordMetaData(final ProducerRecord<String, StockPrice> record,
                                      final Future<RecordMetadata> future)
        throws InterruptedException, ExecutionException {

        final RecordMetadata recordMetadata = future.get();
        logger.info(String.format("\n\t\t\tkey=%s, value=%s " +
                                "\n\t\t\tsent to topic=%s part=%d off=%d at time=%s",
                                record.key(),
                                record.value().toJson(),
                                recordMetadata.topic(),
                                recordMetadata.partition(),
                                recordMetadata.offset(),
                                new Date(recordMetadata.timestamp())
                            ));

    }

    ...

}
```

Every 100 records `StockSender` `displayRecordMetaData` method gets called, which prints out record info, and `recordMetadata` info: key, JSON value, topic, partition, offset, time. The `displayRecordMetaData` uses the `Future` from the call to `producer.send()`.

## ACTION - EDIT

**src/main/java/com/cloudurable/kafka/producer/StockSender.java and follow**

## the instructions.

### Running the example

To run the example, you need to run ZooKeeper, then run the three Kafka Brokers. Once that is running, you will need to run create-topic.sh. And lastly run the `StockPriceKafkaProducer` from the IDE.

First run ZooKeeper.

#### Running ZooKeeper with run-zookeeper.sh (Run in a new terminal)

```
~/kafka-training

$ cat run-zookeeper.sh
#!/usr/bin/env bash
cd ~/kafka-training

kafka/bin/zookeeper-server-start.sh \
    kafka/config/zookeeper.properties

$ ./run-zookeeper.sh
```

Now run the first Kafka Broker.

#### Running the 1st Kafka Broker (Run in a new terminal)

```
~/kafka-training/lab5.1

$ cat bin/start-1st-server.sh
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-0.properties"

$ bin/start-1st-server.sh
```

Now run the second Kafka Broker.

#### Running the 2nd Kafka Broker (Run in a new terminal)

```
~/kafka-training/lab5.1

$ cat bin/start-2nd-server.sh
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-1.properties"

$ bin/start-2nd-server.sh
```

Now run the third Kafka Broker.

#### Running the 3rd Kafka Broker (Run in a new terminal)

```
~/kafka-training/lab5.1

$ cat bin/start-3rd-server.sh
#!/usr/bin/env bash
CONFIG=`pwd`/config
cd ~/kafka-training
## Run Kafka
kafka/bin/kafka-server-start.sh \
    "$CONFIG/server-2.properties"

$ bin/start-3rd-server.sh
```

Once all brokers are running, run create-topic.sh as follows.

### Running create topic

```
~/kafka-training/lab5.1

$ cat bin/create-topic.sh
#!/usr/bin/env bash

cd ~/kafka-training

kafka/bin/kafka-topics.sh \
    --create \
    --zookeeper localhost:2181 \
    --replication-factor 3 \
    --partitions 3 \
    --topic stock-prices \
    --config min.insync.replicas=2

$ bin/create-topic.sh
Created topic "stock-prices".
```

The create-topics script creates a topic. The name of the topic is stock-prices. The topic has three partitions. The created topic has a replication factor of three.

For the config only the broker id and log directory changes.

### config/server-0.properties

```
broker.id=0
listeners=PLAINTEXT://localhost:9092
log.dirs=./logs/kafka-0
...
```

Run the StockPriceKafkaProducer from your IDE. You should see log messages from StockSender(s) with StockPrice name, JSON value, partition, offset, and time.

---

## Kafka Tutorial

This comprehensive *Kafka tutorial* covers Kafka architecture and design. The *Kafka tutorial* has example Java Kafka producers and Kafka consumers. The *Kafka tutorial* also covers Avro and Schema Registry.

[Complete Kafka Tutorial: Architecture, Design, DevOps and Java Examples.](#)

- [Kafka Tutorial Part 1: What is Kafka?](#)
  - [Kafka Tutorial Part 2: Kafka Architecture](<http://cloudurable.com/blog/kafka-architecture/index.html>) "This Kafka tutorial discusses the structure of Kafka. Kafka consists of Records, Topics, Consumers, Producers, Brokers, Logs, Partitions, and Clusters. Records can have key, value and timestamp. Kafka Records are immutable. A Kafka Topic is a stream of records - "/orders", "/user-signups". You can think of a Topic as a feed name. It covers the structure of and purpose of topics, log, partition, segments, brokers, producers, and consumers")
  - [Kafka Tutorial Part 3: Kafka Topic Architecture](#)
  - [Kafka Tutorial Part 4: Kafka Consumer Architecture](#)
  - [Kafka Tutorial Part 5: Kafka Producer Architecture](#)
  - [Kafka Tutorial Part 6: Using Kafka from the command line](#)
  - [Kafka Tutorial Part 7: Kafka Broker Failover and Consumer Failover](#)
  - [Kafka Tutorial Part 8: Kafka Ecosystem](#)
  - [Kafka Tutorial Part 9: Kafka Low-Level Design](#)
  - [Kafka Tutorial Part 10: Kafka Log Compaction Architecture](#)
  - [Kafka Tutorial Part 11: Writing a Kafka Producer example in Java](#)
  - [Kafka Tutorial Part 12: Writing a Kafka Consumer example in Java](#)
  - [Kafka Tutorial Part 13: Writing Advanced Kafka Producer Java examples](#)
  - Kafka Tutorial 14: Writing Advanced Kafka Consumer Java examples
  - [Kafka Tutorial Part 15: Kafka and Avro](#)
  - [Kafka Tutorial Part 16: Kafka and Schema Registry](#)
  - [Kafka Tutorial](#)
- 

#### About Cloudurable

We hope you enjoyed this article. Please provide [feedback](#). Cloudurable provides [Kafka training](#), [Kafka consulting](#), [Kafka support](#) and helps [setting up Kafka clusters in AWS](#).