Kafka Producer Client: Use Cases and Examples

This guide provides a complete, working example of a Java client that produces messages to an Apache Kafka topic. It demonstrates four common producer patterns: Fire-and-Forget, Synchronous, Asynchronous, and a high-throughput configuration.

1. Maven Project Configuration (pom.xml)

This pom.xml file includes the necessary dependencies for the Kafka client and a simple logger to get console output. It also configures the maven-assembly-plugin to create a single "fat" JAR file with all dependencies, making it easy to run the application from the command line.

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>kafka-producer-examples</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <kafka.version>4.0.0</kafka.version>
    </properties>

    <dependencies>
        <!-- Kafka Clients dependency -->
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>${kafka.version}</version>
        </dependency>

        <!-- SLF4J simple logger to get console output -->
```

```
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.7.36</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Maven Compiler Plugin -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11.0</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
      </plugin>
      <!-- Maven Assembly Plugin to create a fat JAR -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>3.6.0</version>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
          <archive>
            <manifest>
              <mainClass>com.example.ProducerExamples</mainClass>
            </manifest>
          </archive>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals>
```

```
                    <goal>single</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
  </plugins>
 </build>
</project>
```

2. Java Producer Examples (ProducerExamples.java)

This single Java class demonstrates four different producer patterns. Each method is a complete, self-contained example of a producer client.

```
package com.example;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

/**
 * A class containing various examples of Kafka Producer implementations:
 * Fire-and-Forget, Synchronous, Asynchronous, and Safe/High-Throughput.
 */
public class ProducerExamples {

   private static final String TOPIC = "producer-topic";
   private static final String BOOTSTRAP_SERVERS =
"kafka-199:9092,kafka2:9092,kafka3:9092";

   public static void main(String[] args) throws InterruptedException,
ExecutionException {
      // Run each producer example in sequence
      System.out.println("--- Running Fire-and-Forget Producer ---");
      fireAndForgetProducer();

      System.out.println("\n--- Running Synchronous Producer ---");
```

```
        synchronousProducer();

        System.out.println("\n--- Running Asynchronous Producer ---");
        asynchronousProducer();

        System.out.println("\n--- Running Safe and High-Throughput Producer ---");
        safeHighThroughputProducer();
    }

    /**
     * A simple producer that sends messages without waiting for a response.
     * This is the fastest but also the least reliable method.
     */
    public static void fireAndForgetProducer() {
        Properties props = createProducerProperties();
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // Create and send a single message
        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC,
"fire-and-forget message");
        producer.send(record);

        System.out.println("Message sent in fire-and-forget mode. Check consumer for
delivery confirmation.");

        producer.close();
    }

    /**
     * A producer that sends messages and waits for an acknowledgment from the
broker.
     * This is highly reliable but much slower due to waiting for each message.
     */
    public static void synchronousProducer() throws ExecutionException,
InterruptedException {
        Properties props = createProducerProperties();
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // Send a message and block until a response is received
```

```
        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, "sync
message");

        try {
            RecordMetadata metadata = producer.send(record).get();
            System.out.printf("Message sent successfully (sync): Topic: %s, Partition: %d,
Offset: %d\n",
                    metadata.topic(), metadata.partition(), metadata.offset());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        producer.close();
    }

    /**
     * The recommended approach. It sends messages asynchronously and uses a
callback
     * to handle the response when it arrives, allowing for high throughput.
     */
    public static void asynchronousProducer() throws InterruptedException {
        Properties props = createProducerProperties();
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, "async
message");

        // Send a message with a callback function
        producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception) {
                if (exception == null) {
                    System.out.printf("Message sent successfully (async): Topic: %s, Partition:
%d, Offset: %d\n",
                            metadata.topic(), metadata.partition(), metadata.offset());
                } else {
                    exception.printStackTrace();
                }
            }
```

```
        });

        // Wait for a moment to ensure the async callback is processed.
        Thread.sleep(1000);

        producer.close();
    }

    /**
     * A producer configured for a balance of safety and high throughput.
     * This combines asynchronous sending with specific property settings.
     */
    public static void safeHighThroughputProducer() throws InterruptedException {
        Properties props = createProducerProperties();

        // 1. Acks: 'all' or '-1' ensures the message is committed by all in-sync replicas.
        //    This is the safest setting, preventing data loss.
        props.put(ProducerConfig.ACKS_CONFIG, "all");

        // 2. Retries: The producer will automatically retry sending the message if it fails.
        props.put(ProducerConfig.RETRIES_CONFIG, 10);

        // 3. Batching: The producer groups messages together to send in batches.
        //    A larger batch size improves throughput but increases latency.
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768); // 32 KB

        // 4. Linger.ms: The producer waits for this duration to fill a batch.
        //    A higher value increases throughput but also increases latency.
        props.put(ProducerConfig.LINGER_MS_CONFIG, 20); // 20 milliseconds

        KafkaProducer<String, String> producer = new KafkaProducer<>(props);
        System.out.println("Sending 10 safe, high-throughput messages...");

        for (int i = 0; i < 10; i++) {
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC,
"high-throughput-key-" + i, "message-" + i);
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {
```

```
                if (exception == null) {
                    // Success
                } else {
                    System.err.println("Error sending message: " + exception.getMessage());
                }
            }
        });
    }

    // Ensure all messages in the batch are flushed and callbacks are processed
    producer.flush();
    producer.close();
    System.out.println("Finished sending safe messages.");
  }

  /**
   * Common properties for the producer, including bootstrap servers and serializers.
   */
  private static Properties createProducerProperties() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
BOOTSTRAP_SERVERS);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
    return props;
  }
}
```

3. Step-by-Step Execution Guide

Step 1: Project Setup

Create a project directory structure.

```
mkdir -p kafka-producer-examples/src/main/java/com/example
```

Copy the pom.xml and ProducerExamples.java content into the corresponding files.

Step 2: Create the Kafka Topic

Create the topic that the producer will write to.

```
sudo -u kafka /usr/local/kafka/bin/kafka-topics.sh --create \
  --topic producer-topic \
  --partitions 3 \
  --replication-factor 3 \
  --bootstrap-server kafka-199:9092
```

Step 3: Compile and Package the Application

Navigate to the project root and use Maven to build the executable JAR file.

```
cd /path/to/your/kafka-producer-examples/
mvn clean package
```

This will create kafka-producer-examples-1.0-SNAPSHOT-jar-with-dependencies.jar in the target/ directory.

Step 4: Run the Producer Application

Execute the JAR file to run all the producer examples sequentially.

```
java -jar target/kafka-producer-examples-1.0-SNAPSHOT-jar-with-dependencies.jar
```

You should see output on the console showing the execution of each producer type.

Step 5: Verify Messages with a Consumer

Open a new terminal and use the console consumer to see the messages being written to the topic.

```
sudo -u kafka /usr/local/kafka/bin/kafka-console-consumer.sh \
  --topic producer-topic \
  --bootstrap-server kafka-199:9092 \
  --from-beginning \
  --property print.key=true
```

You should see all the messages from the different producer examples appear in the consumer's terminal.

###############CODE explained ##########################

That's a great question. Let's break down the `ProducerExamples.java` code in detail. This program is a perfect showcase of the different ways a Java client can interact with Kafka.

At its core, this class is designed to demonstrate four distinct producer patterns, each with its own trade-offs regarding performance and reliability.

Key Components

First, let's look at the main setup in the `ProducerExamples` class:

- **`createProducerProperties()`**: This is a helper method that bundles the fundamental configurations needed for any producer. It sets the Kafka **bootstrap servers** (`kafka-199:9092,kafka2:9092,kafka3:9092`) and specifies the **serializers** for the key and value (`StringSerializer`). A serializer converts Java objects into bytes so they can be sent over the network.
- **`main()`**: This is the entry point of the program. It simply calls each of the four example producer methods in a sequential order, allowing you to see the output of each approach.

---

1. Fire-and-Forget Producer (`fireAndForgetProducer()`)

This is the simplest and fastest way to send a message.

- **How it works**: The producer sends a `ProducerRecord` and immediately moves on without waiting for a response from the Kafka broker.
- **Code**: The key line is `producer.send(record)`. There is no `.get()` call or `Callback` function attached to it.
- **Pros**: Extremely fast because there's no waiting.
- **Cons**: No reliability. You have no way of knowing if the message was

successfully delivered or if it was lost due to a network error or broker failure. This method is generally **not recommended** for most production applications.

---

2. Synchronous Producer (`synchronousProducer()`)

This method ensures the message is successfully delivered to the broker before the program continues.

- **How it works**: After sending the message, the producer's thread blocks and waits for a response from the broker. This response is an acknowledgment that the message has been written to the partition.
- **Code**: The `producer.send(record).get()` call is the key. The `.get()` method is what makes this a blocking, synchronous operation. It waits for the `Future` object returned by `send()` to complete.
- **Pros**: Highly reliable. You know for sure that the message was delivered.
- **Cons**: Very slow. Because it waits for each individual message to be acknowledged, it has low throughput and is not suitable for high-volume applications.

---

3. Asynchronous Producer (`asynchronousProducer()`)

This is the most common and recommended approach, as it balances reliability with high performance.

- **How it works**: The producer sends the message and returns immediately. It doesn't wait for a response. Instead, it provides a **`Callback` function** that will be executed by a separate thread once the broker acknowledges the message.
- **Code**: `producer.send(record, new Callback() {...})`. Inside the `onCompletion` method of the callback, you can handle the success or failure of the message delivery.
- **Pros**: High throughput because the main thread doesn't wait. It's also reliable because the callback will tell you whether the message was successful or not.
- **Cons**: It can be slightly more complex to manage, as you need to handle logic in a separate callback method. The `Thread.sleep()` in the example is just to ensure the callback has time to execute before the program exits; it's not something you'd typically do in a real application.

---

4. Safe and High-Throughput Producer (`safeHighThroughputProducer()`)

This example demonstrates how to configure the asynchronous producer for optimal performance and data safety in a production environment.

- **How it works**: It uses the asynchronous sending pattern but configures several key properties to maximize both throughput and durability.
- **Code and Key Configurations**:
    - `ProducerConfig.ACKS_CONFIG, "all"`: This is a crucial setting for **durability**. It tells the producer to wait for all in-sync replicas to acknowledge the message. If any replica fails, the producer will not receive an acknowledgment, ensuring no data is lost.
    - `ProducerConfig.RETRIES_CONFIG, 10`: The producer will automatically **retry** sending a message up to 10 times if a temporary failure occurs, such as a broker being temporarily unavailable.
    - `ProducerConfig.BATCH_SIZE_CONFIG, 32768`: To improve **throughput**, the producer doesn't send messages one by one. It buffers them and sends them in batches. This setting controls the maximum size of a batch.
    - `ProducerConfig.LINGER_MS_CONFIG, 20`: The producer will wait for up to 20 milliseconds to fill a batch before sending it, even if the batch size limit hasn't been reached. This further optimizes throughput by giving messages time to accumulate.
    - `producer.flush()`: This is called at the end to force all remaining buffered messages to be sent before the application closes.

By combining the asynchronous pattern with these configurations, you get the best of both worlds: a reliable producer that can handle a high volume of messages efficiently.