Kafka is great for messaging between applications, but it also allows you to transform and process data using Kafka Streams. In this lesson, we will provide an overview of what Kafka streams are. We will also implement a basic Kafka Streams application using Java.

Relevant Documentation

- Kafka Streams

Lesson Reference

1. Clone the starter project.

```
cd ~/
git clone https://github.com/linuxacademy/content-ccdak-kafka-streams.git
cd content-ccdak-kafka-streams
```

2. Note that the `kafka-client` and `kafka-streams` dependencies have already been added to `build.gradle`.

3. Edit the `main` class and implement a basic Streams application that simply copies data from the input topic to the output topic.

```
vi src/main/java/com/linuxacademy/ccdak/streams/StreamsMain.java
```

4. Here is an example of the completed `StreamsMain` class.

```
package com.linuxacademy.ccdak.streams;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;

public class StreamsMain {<pre><code>public static void main(String[] args) {
    // Set up the configuration.
    final Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "inventory-data");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
    // Since the input topic uses Strings for both key and value, set the default Serdes to String.
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

    // Get the source stream.
    final StreamsBuilder builder = new StreamsBuilder();
    final KStream&lt;String, String&gt; source = builder.stream("streams-input-topic");

    source.to("streams-output-topic");

    final Topology topology = builder.build();
    final KafkaStreams streams = new KafkaStreams(topology, props);
    // Print the topology to the console.
    System.out.println(topology.describe());
    final CountDownLatch latch = new CountDownLatch(1);

    // Attach a shutdown handler to catch control-c and terminate the application gracefully.
    Runtime.getRuntime().addShutdownHook(new Thread("streams-wordcount-shutdown-hook") {
        @Override
```

```
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (final Throwable e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
        System.exit(0);
    }</code></pre>}
```

5. Run your Streams application.

```
./gradlew run
```

6. In a separate session, use `kafka-console-producer` to publish some data to `streams-input-topic`.

```
kafka-console-producer --broker-list localhost:9092 --topic streams-input-topic --property parse.key=true
```

7. In another session, use `kafka-console-producer` to view the data being sent to `streams-output-topic` by your Java application.

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic streams-output-topic --property print.ke
```

8. If you have both the producer and consumer running, you should see your Java application pushing data to the output topic in real time.