Kafka Streams Hands On

Basic Operations LAB

The Basic Operations exercise demonstrates how to use Kafka Streams stateless operations such as `filter` and `mapValues`. Note that the next few steps, including setting up Confluent Cloud, creating the properties, adding the application ID, and creating the `StreamsBuilder`, apply to each exercise but will only be shown in this one.

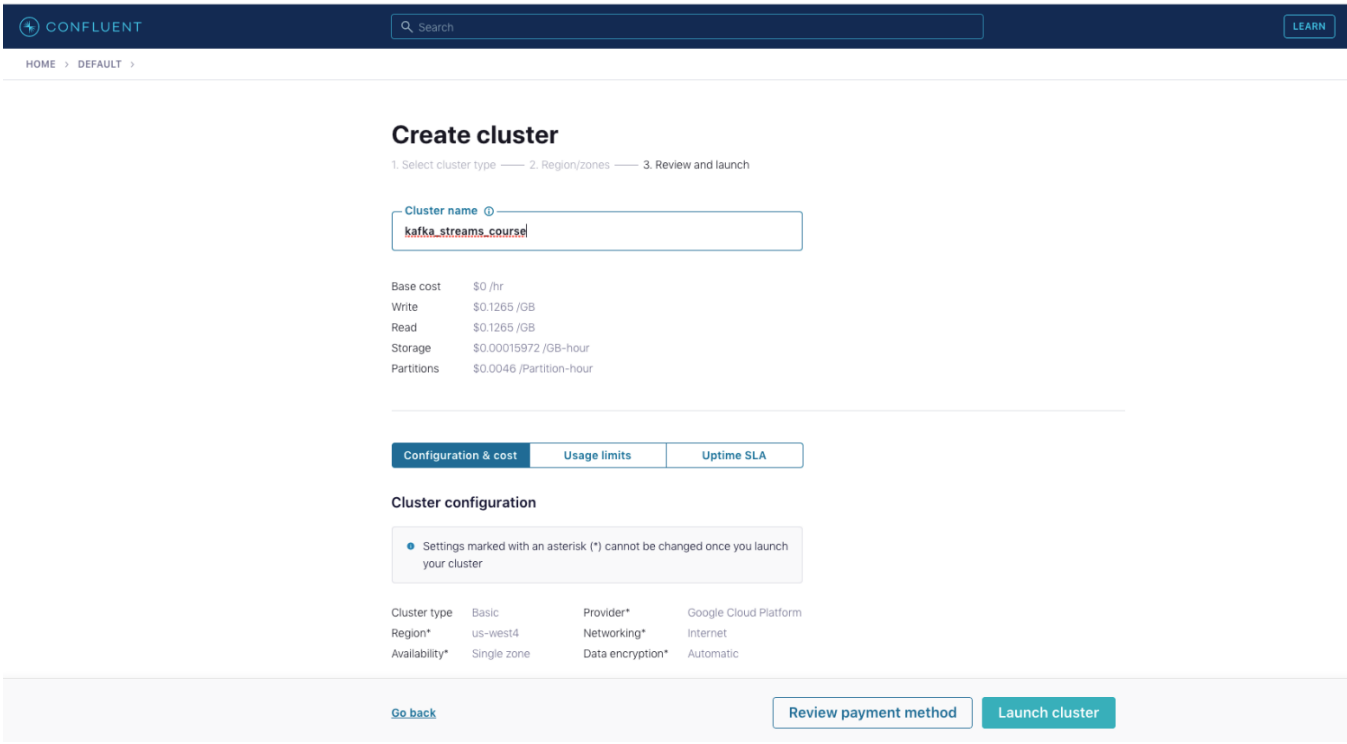
1. Clone the course's GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
```

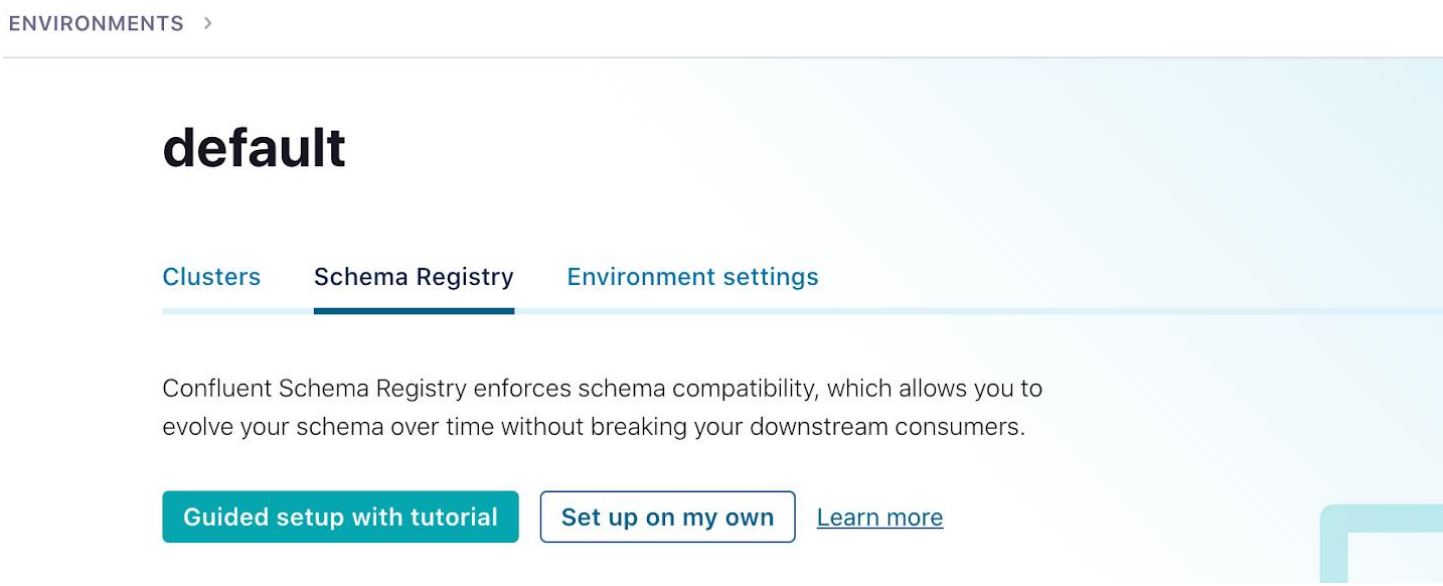
2. Change to the directory.

```
cd kafka-courses/kafka-streams
```

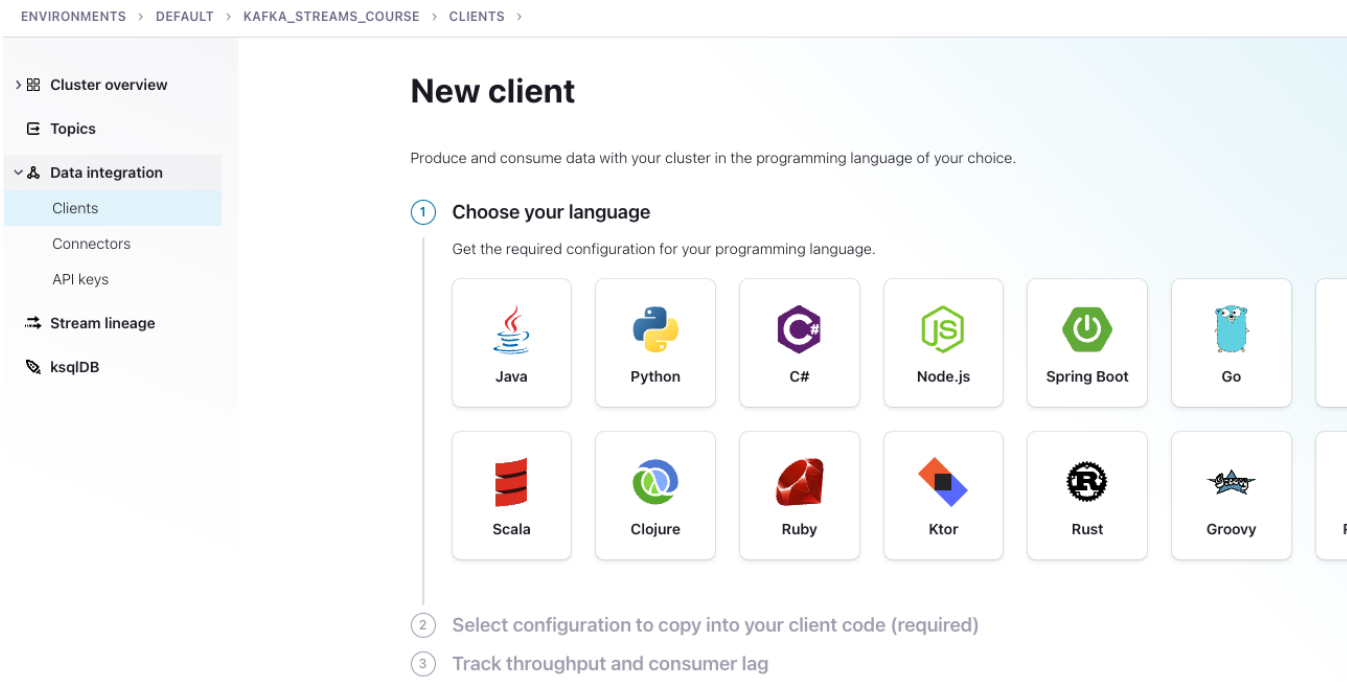
3. Compile the source with `./gradlew build`. The source code in this course is compatible with Java 11, and this module's code can be found in the source file `java/io/confluent/developer/basic/BasicStreams.java`.
4. Go to Confluent Cloud and use the promo code `STREAMS101` for $25 of free usage (details).
5. Create a new cluster in Confluent Cloud. For the purposes of all the exercise modules you can use the **Basic** type. Name the cluster **kafka_streams_course**. Note the associated costs, then click on the **Launch Cluster** button on the bottom right. (Make a mental note at this point to remember to shut down the cluster when you no longer need it; there are shutdown instructions at the end of the course).



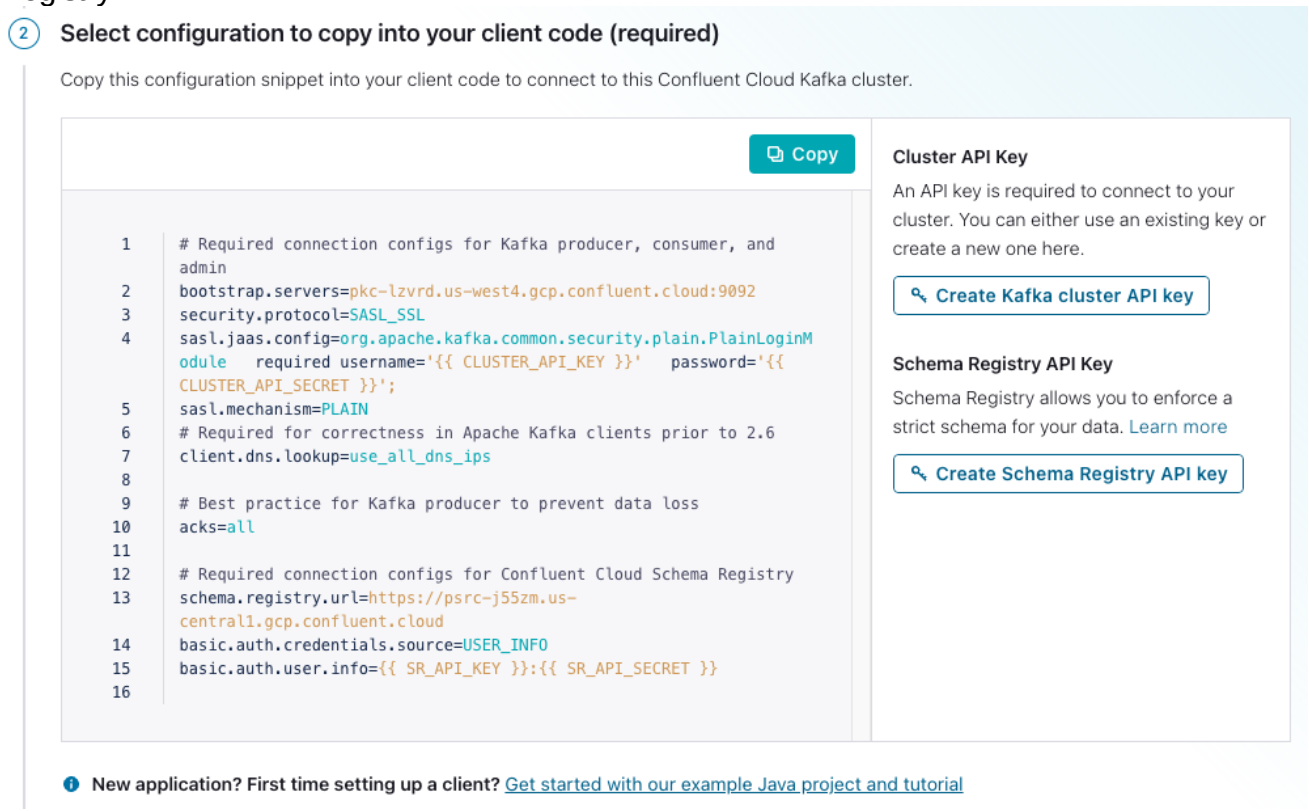
6. You'll also need to set up a Schema Registry. Click on the environment link in the upper left corner (probably DEFAULT). Then click the Schema Registry link and follow the prompts to set up a schema registry on the provider of your choice. Once that is complete, go back to your cluster.



7. Next click on **Data Integration** in the menu on the left, then select **Clients**, then the **Java** tile.



8. You'll be dropped down to a window where you can create credentials for your cluster and Confluent Schema Registry.



1. Click on **Create Kafka cluster API key**.

2. your key and secret, name the file, then click **Download and continue**. (Your credentials will populate into the configurations boilerplate.)

3. Click on **Create Schema Registry API key**.

4. your key and secret, name the file, then click **Download and continue**. (Your credentials will populate into the configurations boilerplate.)

5. Make sure **Show API keys** is selected, then the configurations in the window.

6. Create a file named `ccloud.properties` in the `src/main/resources` directory of the repo you downloaded. Then paste the configurations into the `ccloud.properties` file. Note that this file is ignored and should never get checked into GitHub.

9. Now set up properties for the exercises (you are taking a couple of minor extra steps to make sure that any sensitive information doesn't get accidentally checked into GitHub). First make sure you are in `src/main/resources`. Then run the following commands:

   - `cat streams.properties.orig > streams.properties`
   - `cat ccloud.properties >> streams.properties`

10. Next, create a `properties` object in your `BasicStreams.java` file:

```
package io.confluent.developer.basic;


import java.io.IOException;

import java.util.Properties;


public class BasicStreams {

    public static void main(String[] args) throws IOException {

    Properties streamsProps = new Properties();

    }
}
```

11. Use a `FileInputStream` to load properties from the file that includes your Confluent Cloud properties; in addition, add the application configuration ID to the properties:

```
try (FileInputStream fis = new FileInputStream("src/main/resources/streams
.properties")) {

    streamsProps.load(fis);

}
streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG, "basic-streams");
```

12. Create a `StreamsBuilder` instance, and retrieve the name of the `inputTopic` and `outputTopic` from the `Properties`:

```
StreamsBuilder builder = new StreamsBuilder()

final String inputTopic = streamsProps.getProperty("basic.input.topic");
final String outputTopic = streamsProps.getProperty("basic.output.topic");
```

13. Create an order number variable (you'll see where it comes into play soon), and then create the KStream instance (note the use of the `inputTopic` variable):

```
final String orderNumberStart = "orderNumber-";
KStream<String, String> firstStream = builder.stream(inputTopic,
Consumed.with(Serdes.String(), Serdes.String()));
```

14. Add a `peek` operator (it's expected that you don't modify the keys and values). Here, it's printing records as they come into the topology:

```
firstStream.peek((key, value) -> System.out.println("Incoming record - key
" +key +" value " + value))
```

15. Add a `filter` to drop records where the value doesn't contain an order number string:

```
.filter((key, value) -> value.contains(orderNumberStart))
```

16. Add a `mapValues` operation to extract the number after the dash:

```
.mapValues(value -> value.substring(value.indexOf("-") + 1))
```

17. Add another `filter` to drop records where the value is not greater than 1000:

```
.filter((key, value) -> Long.parseLong(value) > 1000)
```

18. Add an additional `peek` method to display the transformed records:

```
.peek((key, value) -> System.out.println("Outgoing record - key " +key +"
value " + value))
```

19. Add the `to` operator, the processor that writes records to a topic:

```
.to(outputTopic, Produced.with(Serdes.String(), Serdes.String()));
```

20. Create the Kafka Streams instance:

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),
streamsProps);
```

21. Use the utility method `TopicLoader.runProducer()` to create the required topics on the cluster and produce some sample records (we'll see this pattern throughout the exercises, but keep in mind that it's not part of a standard Kafka Streams application):

```
TopicLoader.runProducer();
```

22. Start the application:

```
kafkaStreams.start();
```

Now you can run the basic operations example with this command:

```
./gradlew runStreams -Pargs=basic
```

and your output on the console should resemble this:

```
Incoming record - key order-key value orderNumber-1001

Outgoing record - key order-key value 1001

Incoming record - key order-key value orderNumber-5000
Outgoing record - key order-key value 5000
```

Take note that it's expected to not have a corresponding output record for each input record due to the filters applied by the Kafka Steams application.

KTABLES LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code. This module's code can be found in the source file `java/io/confluent/developer/ktable/KTableExample.java`.

This exercise features the KTable version of the Streams application shown in the Basic Operations exercise.

1. Start by creating a variable to store the string that we want to filter on:

```
final String orderNumberStart = "orderNumber-";
```

2. Now create the KTable instance. Note that you call `builder.table` instead of `builder.stream`; also, with the `Materialized` configuration object, you need to provide a name for the KTable in order for it to be materialized. It will use caching and will only emit the latest records for each key after a commit (which is 30 seconds, or when the cache is full at 10 MB).

```
KTable<String, String> firstKTable = builder.table(inputTopic,
    Materialized.<String, String, KeyValueStore<Bytes, byte[]>>as("ktable-store")
```

3. Add SerDes for the key and value on your `Materialized` object:

```
.withKeySerde(Serdes.String())
.withValueSerde(Serdes.String()));
```

4. Add a `filter` operator for removing records that don't contain the order number variable value:

```
firstKTable.filter((key, value) -> value.contains(orderNumberStart))
```

5. Map the values by taking a substring:

```
.mapValues(value -> value.substring(value.indexOf("-") + 1))
```

6. Then filter again by taking out records where the number value of the string is less than or equal to 1000:

```
.filter((key, value) -> Long.parseLong(value) > 1000)
```

7. Convert the KTable to a KStream:

```
.toStream()
```

8. Add a `peek` operation to view the key values from the table:

```
.peek((key, value) -> System.out.println("Outgoing record - key " +key +" value " + value))
```

9. Write the records to a topic:

```
.to(outputTopic, Produced.with(Serdes.String(), Serdes.String()));
```

10. Create a `KafkaStreams` object and run the topic data helper utility:

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsProps);
TopicLoader.runProducer();
```

11. Finally, start the application:

```
kafkaStreams.start();
```

Now you can run the `KTable` example with this command:

```
./gradlew runStreams -Pargs=ktable
```

You should let the application run for about 40 seconds, and you should see one result output:

```
Outgoing record - key order-key value 8400
```

There's a single output result because the sample data for this exercise has the same key.

Joins LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code.

This module's code can be found in the source file `java/io/confluent/developer/joins/StreamsJoin.java`.

This exercise teaches you how to join two streams into a third stream, and then join that third stream with a table.

1. Use a static helper method to get SerDes for your Avro records (in subsequent exercises, you'll abstract this into a static utility method, in the `StreamsUtils` class of the course repo):

```
static <T extends SpecificRecord> SpecificAvroSerde<T> getSpecificAvroSerd
e(Map<String, Object> serdeConfig) {

    SpecificAvroSerde<T> specificAvroSerde = new SpecificAvroSerde<>();

    specificAvroSerde.configure(serdeConfig, false);

    return specificAvroSerde;
}
```

2. Use a utility method to load the properties (you can refer to the `StreamsUtils` class within the exercise source code):

```
Properties streamsProps = StreamsUtils.loadProperties();
```

3. Get the input topic names and the output topic name from the properties:

```
String streamOneInput = streamsProps.getProperty("stream_one.input.topic")
;
String streamTwoInput = streamsProps.getProperty("stream_two.input.topic")
;
String tableInput = streamsProps.getProperty("table.input.topic");
String outputTopic = streamsProps.getProperty("joins.output.topic");
```

4. Create a HashMap of the configurations:

```
Map<String, Object> configMap =
StreamsUtils.propertiesToMap(streamsProps);
```

5. Then create the required SerDes for all streams and for the table:

```
SpecificAvroSerde<ApplianceOrder> applianceSerde = getSpecificAvroSerde(co
nfigMap);
SpecificAvroSerde<ElectronicOrder> electronicSerde = getSpecificAvroSerde(
configMap);
SpecificAvroSerde<CombinedOrder> combinedSerde = getSpecificAvroSerde(conf
igMap);
SpecificAvroSerde<User> userSerde = getSpecificAvroSerde(configMap);
```

6. Create the `ValueJoiner` for the stream-table join:

```
ValueJoiner<ApplianceOrder, ElectronicOrder, CombinedOrder> orderJoiner =
    (applianceOrder, electronicOrder) -> CombinedOrder.newBuilder()
        .setApplianceOrderId(applianceOrder.getOrderId())
        .setApplianceId(applianceOrder.getApplianceId())
```

```
        .setElectronicOrderId(electronicOrder.getOrderId())

        .setTime(Instant.now().toEpochMilli())
        .build();
```

The stream is a result of the preceding stream-stream join, but it's a left outer join, because the right-side record might not exist.

7. Create the `ApplianceOrder` stream as well as the `ElectronicOrder` stream:

```
KStream<String, ApplianceOrder> applianceStream =

    builder.stream(streamOneInput, Consumed.with(Serdes.String(), applianc
eSerde))

        .peek((key, value) -> System.out.println("Appliance stream incomin
g record key " + key + " value " + value));


KStream<String, ElectronicOrder> electronicStream =

    builder.stream(streamTwoInput, Consumed.with(Serdes.String(), electron
icSerde))
        .peek((key, value) -> System.out.println("Electronic stream
incoming record " + key + " value " + value));
```

8. From here, create the `User` table:

```
KTable<String, User> userTable = builder.table(tableInput,
Materialized.with(Serdes.String(), userSerde));
```

9. Now create the stream-stream join and call the `join` method on the `applianceStream`, the left side (or primary) stream in the join. Add the `electronicStream` as the right side (or secondary) stream in the join, and add the `orderJoiner` created before:

```
KStream<String, CombinedOrder> combinedStream =
```

```
      applianceStream.join(

            electronicStream,

            orderJoiner,
```

10. Specify a `JoinWindows` configuration of 30 minutes (a right-side record must have timestamps within 30 minutes before or after the timestamp of the left side for a join result to occur):

```
JoinWindows.of(Duration.ofMinutes(30)),
```

11. Add the `StreamJoined` configuration with SerDes for the key, left-side, and right-side objects, for the joined state stores:

```
StreamJoined.with(Serdes.String(), applianceSerde, electronicSerde))
```

12. Add a `peek` operator to view the results of the join:

```
.peek((key, value) -> System.out.println("Stream-Stream Join record key "
+ key + " value " + value));
```

13. Call the `join` method on the KStream that results from the join in previous steps, adding the `userTable` as the right side in the stream-table join. Then add `enrichmentJoiner` to add user information, if available. Add the `Joined` object with SerDes for the values of both sides of the join, add the `peek` operator to view the stream-table join results, and write the final join results to a topic:

```
combinedStream.leftJoin(

                userTable,

                enrichmentJoiner,

                Joined.with(Serdes.String(), combinedSerde, userSerde))

    .peek((key, value) -> System.out.println("Stream-Table Join record key
" + key + " value " + value))
```

```
    .to(outputTopic, Produced.with(Serdes.String(), combinedSerde));
```

14. Create the `KafkaStreams` object, and again use the `TopicLoader` helper class to create topics and produce exercise data:

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsProps
);
TopicLoader.runProducer();
```

15. Finally, start the Kafka Streams application:

```
kafkaStreams.start();
```

You run the joins example with this command:

```
./gradlew runStreams -Pargs=joins
```

The output for the exercise should like this:

```
Appliance stream incoming record key 10261998 value {"order_id": "remodel-1", "ap
pliance_id": "dishwasher-1333", "user_id": "10261998", "time": 1622148573134}

Electronic stream incoming record 10261999 value {"order_id": "remodel-2", "elect
ronic_id": "laptop-5333", "user_id": "10261999", "price": 0.0, "time": 1622148573
146}

Electronic stream incoming record 10261998 value {"order_id": "remodel-1", "elect
ronic_id": "television-2333", "user_id": "10261998", "price": 0.0, "time": 162214
8573136}

Stream-Stream Join record key 10261998 value {"electronic_order_id": "remodel-1",
"appliance_order_id": "remodel-1", "appliance_id": "dishwasher-1333", "user_name"
: "", "time": 1622148582747}
```

```
Stream-Table Join record key 10261998 value {"electronic_order_id": "remodel-1",
"appliance_order_id": "remodel-1", "appliance_id": "dishwasher-1333", "user_name"
: "Elizabeth Jones", "time": 1622148582747}

Appliance stream incoming record key 10261999 value {"order_id": "remodel-2", "ap
pliance_id": "stove-2333", "user_id": "10261999", "time": 1622148573134}

Stream-Stream Join record key 10261999 value {"electronic_order_id": "remodel-2",
"appliance_order_id": "remodel-2", "appliance_id": "stove-2333", "user_name": "",
"time": 1622148582853}
Stream-Table Join record key 10261999 value {"electronic_order_id": "remodel-2",
"appliance_order_id": "remodel-2", "appliance_id": "stove-2333", "user_name": "",
"time": 1622148582853}
```

Aggregations LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code. This module's code can be found in the source file `java/io/confluent/developer/aggregate/StreamsAggregate.java`.

This hands-on exercise demonstrates stateful operations in Kafka Streams, specifically aggregation, using a simulated stream of electronic purchases. You'll see the incoming records on the console along with the aggregation results.

1. To begin, let's extract the names of the topics from the configuration, which we've already loaded via a static helper method. Then we'll convert the properties to a HashMap and use another utility method to create the specific record `AvroSerde`.

```
String inputTopic = streamsProps.getProperty("aggregate.input.topic");
```

```
String outputTopic = streamsProps.getProperty("aggregate.output.topic");

Map<String, Object> configMap = StreamsUtils.propertiesToMap(streamsProps)
;


SpecificAvroSerde<ElectronicOrder> electronicSerde =
StreamsUtils.getSpecificAvroSerde(configMap);
```

2. Create the electronic orders stream:

```
KStream<String, ElectronicOrder> electronicStream =

    builder.stream(inputTopic, Consumed.with(Serdes.String(), electronicSe
rde))
            .peek((key, value) -> System.out.println("Incoming record - key
" +key +" value " + value));
```

3. Execute a `groupByKey` followed by `aggregate` (initialize the aggregator with "0.0," a double value):

```
electronicStream.groupByKey().aggregate(() -> 0.0,
```

4. Now add the aggregator implementation, which takes each order and adds the price to a running `total`, a sum of all electronic orders. Also add a `Materialized`, which is necessary to provide state store SerDes since the value type has changed.

```
(key, order, total) -> total + order.getPrice(),
Materialized.with(Serdes.String(), Serdes.Double()))
```

Call `toStream()` on the KTable that results from the aggregation operation, add a `peek` operation to print the results of the aggregation, and then add a `.to` operator to write the results to a topic:

```
.toStream()
```

```
.peek((key, value) -> System.out.println("Outgoing record - key " +key +"
value " + value))
.to(outputTopic, Produced.with(Serdes.String(), Serdes.Double()));
```

5. Create the `KafkaStreams` object, and again use the `TopicLoader` helper class to create topics and produce exercise data:

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsProps
);
TopicLoader.runProducer();
```

6. Finally, start the Kafka Streams application, making sure to let it run for more than 30 seconds:

```
kafkaStreams.start();
```

To run the aggregation example use this command:

```
./gradlew runStreams -Pargs=aggregate
```

You'll see the incoming records on the console along with the aggregation results:

```
Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "10261998", "price": 2000.0, "time": 1622149038018}
Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1033737373", "price": 1999.23, "time": 1622149048018}
Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1026333", "price": 4500.0, "time": 1622149058018}
Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622149070018}
Outgoing record - key HDTV-2333 value 9833.21
```

NOTE that you'll need to let the application run for ~40 seconds to see the aggregation result.

Windowing LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code. This module's code can be found in the source file `java/io/confluent/developer/windows/StreamsWindows.java`

In this hands-on exercise, you will write a windowed aggregation. Note that the application you'll build uses the default timestamp extractor FailOnInvalidTimestamp.

1. Use the following code as a starter for your windowing code:

```
public class StreamsWindows {

    public static void main(String[] args) throws IOException {

        Properties streamsProps = StreamsUtils.loadProperties();

        streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG, "windowed-st
reams");

        StreamsBuilder builder = new StreamsBuilder();

        String inputTopic = streamsProps.getProperty("windowed.input.topic
");

        String outputTopic = streamsProps.getProperty("windowed.output.top
ic");
```

```
        Map<String, Object> configMap = StreamsUtils.propertiesToMap(strea
msProps);
        SpecificAvroSerde<ElectronicOrder> electronicSerde =
StreamsUtils.getSpecificAvroSerde(configMap);
```

2. Use `builder.stream` to create a `KStream` named `electronicStream`, with a `peek` operator to observe incoming events:

```
KStream<String, ElectronicOrder> electronicStream =

    builder.stream(inputTopic, Consumed.with(Serdes.String(), electronicSe
rde))
            .peek((key, value) -> System.out.println("Incoming record - key
" +key +" value " + value));
```

3. Perform a `groupByKey`, and add a one-hour tumbling window with a grace period of five minutes:

```
electronicStream.groupByKey()

.windowedBy(TimeWindows.of(Duration.ofHours(1)).grace(Duration.ofMinutes(5
)))
```

4. Create the aggregation, initialize it to zero, and then add the aggregator implementation, which calculates a running sum of electronic purchase events.

```
 .aggregate(() -> 0.0,
                (key, order, total) -> total + order.getPrice(),
```

5. Add SerDes for the types in the aggregation, which are used by the state store. Then, add a suppress operator in order to not emit any updates until the window closes:

```
Materialized.with(Serdes.String(), Serdes.Double()))
```

```
.suppress(untilWindowCloses(unbounded()))
```

This gives a single result for the windowed aggregation.
(The `unbounded` parameter means that the buffer will continue to consume memory as needed until the window closes.) Suppression is optional, particularly if you want to see any intermediate results.

6. Convert the `KTable` to a `KStream`, `map` the windowed key to an underlying record key, add a `peek` operator to view the underlying result, and add a `to` operator to write the results to a topic:

```
.toStream()

.map((wk, value) -> KeyValue.pair(wk.key(),value))

.peek((key, value) -> System.out.println("Outgoing record - key " +key +"
value " + value))
.to(outputTopic, Produced.with(Serdes.String(), Serdes.Double()));
```

To run the windowing example use this command:

```
./gradlew runStreams -Pargs=windows
```

Your output should look something like this:

```
Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "10261998", "price": 2000.0, "time": 1622152480629}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1033737373", "price": 1999.23, "time": 1622153380629}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1026333", "price": 4500.0, "time": 1622154280629}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622155180629}
```

```
Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622156260629}

Incoming record - key SUPER-WIDE-TV-2333 value {"order_id": "instore-1", "electro
nic_id": "SUPER-WIDE-TV-2333", "user_id": "1038884844", "price": 5333.98, "time":
1622156260629}

Incoming record - key SUPER-WIDE-TV-2333 value {"order_id": "instore-1", "electro
nic_id": "SUPER-WIDE-TV-2333", "user_id": "1038884844", "price": 4333.98, "time":
1622158960629}

Outgoing record - key HDTV-2333 value 2000.0

Outgoing record - key HDTV-2333 value 9167.189999999999
Outgoing record - key SUPER-WIDE-TV-2333 value 5333.98
```

Let the code run for a minimum of 40 seconds to see the final results for the different keys.

Note that the example uses simulated timestamps for the different keys, so the results only approximate what you would see in production.

Time Concepts LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code.

This module's code can be found in the source file `java/io/confluent/developer/time/StreamsTimestampExtractor.java`.

In this hands-on exercise, learn how to use a custom `TimestampExtractor` to drive the behavior of a Kafka Streams application, using timestamps embedded in the records themselves.

1. Use the following code as a starter for your time exercise code:

```
public class StreamsTimestampExtractor {

      public static void main(String[] args) throws IOException {


            Properties streamsProps = StreamsUtils.loadProperties();

            streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG,
"extractor-windowed-streams");


            StreamsBuilder builder = new StreamsBuilder();

            String inputTopic = streamsProps.getProperty("extractor.input.
topic");

            String outputTopic = streamsProps.getProperty("extractor.outpu
t.topic");

            Map<String, Object> configMap = StreamsUtils.propertiesToMap(s
treamsProps);


            SpecificAvroSerde<ElectronicOrder> electronicSerde =

                StreamsUtils.getSpecificAvroSerde(configMap);


            KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),
streamsProps);

            TopicLoader.runProducer();

            kafkaStreams.start();

    }
}
```

2. Above the `main` method, create an instance of a `TimestampExtractor`, implementing the `extract` method and retrieving the `ElectronicOrder` object from the `ConsumerRecord` value field; then extract and return the timestamp embedded in the `ElectronicOrder':

```
static class OrderTimestampExtractor implements TimestampExtractor {

      @Override

      public long extract(ConsumerRecord<Object, Object> record, long part
itionTime) {

            ElectronicOrder order = (ElectronicOrder)record.value();

            System.out.println("Extracting time of " + order.getTime() + "
from " + order);

            return order.getTime();

        }
    }
```

3. Just above your `kafkaStreams` instance, create a `KStream`, and make the familiar `builder.stream` call:

```
final KStream<String, ElectronicOrder> electronicStream =

      builder.stream(inputTopic,
```

4. Add the `Consumed` configuration object with SerDes for deserialization, but with a twist: You're also providing a `TimestampExtractor`. (You could also specify the `TimestampExtractor` by configurations, but then it would be global for all streams in the application.)

```
Consumed.with(Serdes.String(), electronicSerde)

    .withTimestampExtractor(new OrderTimestampExtractor()))

    .peek((key, value) -> System.out.println("Incoming record - key " +key
+" value " + value));
```

5. Create a tumbling window aggregation. Keep in mind that the timestamps from `ElectronicOrder` are what drive the window opening and closing.

```
electronicStream.groupByKey().windowedBy(TimeWindows.of(Duration.ofHours(1
)))
```

6. Call the `aggregate` method, initializing the aggregate to "0.0" and adding the aggregator instance that sums all prices for the total spent over one hour, based on the timestamp of the record itself. Add SerDes for the state store via a `Materialized`, and convert the KTable from the aggregation into a KStream.

```
.aggregate(() -> 0.0,

           (key, order, total) -> total + order.getPrice(),

           Materialized.with(Serdes.String(), Serdes.Double()))
.toStream()
```

7. Use a `map` processor to unwrap the windowed key and return the underlying key of the aggregation, and use a `peek` processor to print the aggregation results to the console. Finally, write the results out to a topic.

```
.map((wk, value) -> KeyValue.pair(wk.key(),value))

.peek((key, value) -> System.out.println("Outgoing record - key " +key +"
value " + value))
.to(outputTopic, Produced.with(Serdes.String(), Serdes.Double()));
```

8. As with the other aggregation applications, let this one run for at least 40 seconds.

To run this example use the following command:

```
./gradlew runStreams -Pargs=time
```

Your output will include statements from the `TimestampExtractor` and it should look something like this:

```
Extracting time of 1622155705696 from {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "10261998", "price": 2000.0, "time": 1622155705696}

Extracting time of 1622156605696 from {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1033737373", "price": 1999.23, "time": 1622156605696}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "10261998", "price": 2000.0, "time": 1622155705696}

Extracting time of 1622157505696 from {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1026333", "price": 4500.0, "time": 1622157505696}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1033737373", "price": 1999.23, "time": 1622156605696}

Extracting time of 1622158405696 from {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622158405696}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1026333", "price": 4500.0, "time": 1622157505696}

Extracting time of 1622159485696 from {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622159485696}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622158405696}

Incoming record - key HDTV-2333 value {"order_id": "instore-1", "electronic_id":
"HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622159485696}

Outgoing record - key HDTV-2333 value 2000.0
Outgoing record - key HDTV-2333 value 9167.189999999999
```

Processor API LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code.

This module's code can be found in the source file `java/io/confluent/developer/processor/ProcessorApi.java`.

In this exercise, you will create an aggregation that calls a punctuation every 30 seconds. You'll use a `ProcessorSupplier` and `Processor` instance, whereby the `Processor` will contain all of your stream processing logic.

1. Create the `ProcessorSupplier` implementation:

```
static class TotalPriceOrderProcessorSupplier implements    ProcessorSuppl
ier<String, ElectronicOrder, String, Double> {

    final String storeName;
}
```

2. Add a constructor. The `get()` method implementation is important, since it returns a new processor instance each time it's called. You'll also declare a variable for `ProcessorContext` and `KeyValueStore`, and implement the `init` method, which is called by Kafka Streams when the application is starting up. In the `init` method, store a reference to the Processor context, get a reference to the state store by name, and store it in the `storeName` variable declared above. Then use the processor context to schedule a punctuation that fires every 30 seconds, based on stream time.

```
@Override
public Processor<String, ElectronicOrder, String, Double> get() {
    return new Processor<>() {
        private ProcessorContext<String, Double> context;
```

```
        private KeyValueStore<String, Double> store;

        @Override

        public void init(ProcessorContext<String, Double> context) {

            this.context = context;

            store = context.getStateStore(storeName);

            this.context.schedule(Duration.ofSeconds(30), PunctuationType.
STREAM_TIME, this::forwardAll);

         }
    }
```

3. Implement the `forwardAll` method, beginning by opening an iterator for all records in the store. (It's important to close iterators when you're done with them; it's best to use them within a `try-with-resources` block, so that closing is automatic.)

```
private void forwardAll(final long timestamp) {
    try (KeyValueIterator<String, Double> iterator = store.all()) {
```

4. Iterate over all of the records, and get a `KeyValue` from the iterator inside your loop. In addition, create a new `Record` instance. Then forward the `Record` to any child nodes.

```
while (iterator.hasNext()) {

    final KeyValue<String, Double> nextKV = iterator.next();

    final Record<String, Double> totalPriceRecord = new Record<>(nextKV.ke
y, nextKV.value, timestamp);

    context.forward(totalPriceRecord)

    }

  }
}
```

5. Implement the `Process` method on the `Processor` interface by first getting the `key` from the `Record`, then using the `key` to see if there is a value in the state store. If it's null, initialize it to "0.0". Add the current price from the record to the total, and place the new value in the store with the given key.

```
@Override
public void process(Record<String, ElectronicOrder> record) {
    final String key = record.key();
    Double currentTotal = store.get(key);
    if (currentTotal == null) {
        currentTotal = 0.0;
    }
    Double newTotal = record.value().getPrice() + currentTotal;
    store.put(key, newTotal);
}
```

6. We're not quite done with the `ProcessorSupplier` implementation, but we have some details to attend to first. Define the `storeName` variable and create a `StoreBuilder`, which you'll need for creating the state store. In the `StoreBuilder`, set the store type to `persistent` and use the `storeName` variable for the name of the store. Add SerDes for the key/value types in the store (Kafka Streams stores everything as byte arrays in state stores).

```
final static String storeName = "total-price-store";
static StoreBuilder<KeyValueStore<String, Double>> totalPriceStoreBuilder
= Stores.keyValueStoreBuilder(
    Stores.persistentKeyValueStore(storeName),
    Serdes.String(),
    Serdes.Double());
```

7. With `StoreBuilder` complete, now override the `Stores` method on the Processor interface, which gives the Processor access to the store:

```
@Override
public Set<StoreBuilder<?>> stores() {
    return Collections.singleton(totalPriceStoreBuilder);
}
```

8. Now build a topology for the streaming application. This will take a few more steps since we're using the Processor API and not the Kafka Streams DSL. Begin by creating an instance and adding a source node (you need to provide the names for the source node, SerDes, and input topic):

```
final Topology topology = new Topology();
topology.addSource(
    "source-node",
    stringSerde.deserializer(),
    electronicSerde.deserializer(),
    inputTopic);
```

9. Next, add a processor to the topology. Provide a name for the `Processor`, add the `ProcessorSupplier` instance you created before, and set the parent name(s) for the `Processor` (you can specify multiple names).

```
topology.addProcessor(
    "aggregate-price",
    new TotalPriceOrderProcessorSupplier(storeName),
    "source-node");
```

10. Complete the topology by adding a sink node, specifying its name and then adding an output topic, SerDes, and parent name(s):

```
topology.addSink(
    "sink-node",
    outputTopic,
    stringSerde.serializer(),
    doubleSerde.serializer(),
    "aggregate-price");
```

11. Finally, instantiate the `kafkaStreams` object, add the utility method for creating topics and providing sample data, and start the application.

```
final KafkaStreams kafkaStreams = new KafkaStreams(topology, streamsProps)
;
    TopicLoader.runProducer();
    kafkaStreams.start();
```

You run this example with the following command:

```
./gradlew runStreams -Pargs=processor
```

Your results should look like this:

```
Processed incoming record - key HDTV-2333 value {"order_id": "instore-1", "electr
onic_id": "HDTV-2333", "user_id": "10261998", "price": 2000.0, "time": 1622156159
867}
Punctuation forwarded record - key HDTV-2333 value 2000.0
Processed incoming record - key HDTV-2333 value {"order_id": "instore-1", "electr
onic_id": "HDTV-2333", "user_id": "1033737373", "price": 1999.23, "time": 1622156
194867}
```

```
Punctuation forwarded record - key HDTV-2333 value 3999.23

Processed incoming record - key HDTV-2333 value {"order_id": "instore-1", "electr
onic_id": "HDTV-2333", "user_id": "1026333", "price": 4500.0, "time": 16221562298
67}

Punctuation forwarded record - key HDTV-2333 value 8499.23

Processed incoming record - key HDTV-2333 value {"order_id": "instore-1", "electr
onic_id": "HDTV-2333", "user_id": "1038884844", "price": 1333.98, "time": 1622156
264867}
Punctuation forwarded record - key HDTV-2333 value 9833.21
```

Note that the timestamps are simulated to provide more activity, and the observed behavior could be different in a production environment.

Error Handling LAB

If you haven't already, clone the course GitHub repository and load it into your favorite IDE or editor.

```
git clone https://github.com/fenago/kafka-courses.git
cd kafka-courses/kafka-streams
```

The source code in this course is compatible with Java 11. Compile the source with `./gradlew build` and follow along in the code. This module's code can be found in the source file `java/io/confluent/developer/errors/StreamsErrorHandling.java`.

In this exercise, you'll essentially take the Basic Operations exercise and add error handling code to it.

1. Begin with the following code, adding to it as necessary. (Note that the static Boolean `throwErrorNow` exists for simulation purposes only, and the `streamWithErrorHandling` filter's `mapValues` is set up to throw an exception the first time it encounters a record for transient error simulation purposes only.)

```
public class StreamsErrorHandling {

    //This is for learning purposes only!
```

```
    static boolean throwErrorNow = true;


    public static void main(String[] args) throws IOException {
        final Properties streamsProps = StreamsUtils.loadProperties();
        streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-err
or-handling");


        StreamsBuilder builder = new StreamsBuilder();
        final String inputTopic = streamsProps.getProperty("error.input.to
pic");
        final String outputTopic = streamsProps.getProperty("error.output.
topic");


        final String orderNumberStart = "orderNumber-";
        KStream<String, String> streamWithErrorHandling =
            builder.stream(inputTopic, Consumed.with(Serdes.String(), Serd
es.String()))
                .peek((key, value) -> System.out.println("Incoming record
- key " +key +" value " + value));


        streamWithErrorHandling.filter((key, value) -> value.contains(orde
rNumberStart))
            .mapValues(value -> {
                if (throwErrorNow) {
                    throwErrorNow = false;
                    throw new IllegalStateException("Retryable transient e
rror");
```

```
                }
                return value.substring(value.indexOf("-") + 1);
            })
            .filter((key, value) -> Long.parseLong(value) > 1000)
            .peek((key, value) -> System.out.println("Outgoing record - ke
y " +key +" value " + value))
            .to(outputTopic, Produced.with(Serdes.String(), Serdes.String(
)));


    }
}
```

2. Begin by creating a `DeserializationExceptionHandler` above your `main` method to handle any serialization errors. (We'll discuss `errorCounter` in the next step.)

```
public static class StreamsDeserializationErrorHandler implements Deserial
izationExceptionHandler {
    int errorCounter = 0;
```

Underneath that, implement a `handle` method. (Note that each of the various error handling interfaces provides a `handle` method.)

```
@Override
public DeserializationHandlerResponse handle(ProcessorContext context,
                         ConsumerRecord<byte[], byte[]> record,
                         Exception exception) {
    if (errorCounter++ < 25) {
        return DeserializationHandlerResponse.CONTINUE;
```

```
    }

    return DeserializationHandlerResponse.FAIL;
}
```

The `errorCounter` variable counts errors, and if there are fewer than 25, your program will continue processing. Once the counter exceeds 25, the program will stop processing.

Complete the implementation by adding a no-op configure method:

```
@Override

    public void configure(Map<String, ?> configs) { }
}
```

3. Under the first error handler, create an instance of the `ProductionExceptionHandler` interface. As with the other error handler, you'll add a `handle` method with the condition for continued processing: If it's a `RecordTooLargeException`, then the application will continue processing, but if it's any other exception, you'll shut down the application.

```
public static class StreamsRecordProducerErrorHandler implements Productio
nExceptionHandler {

    @Override

    public ProductionExceptionHandlerResponse handle(ProducerRecord<byte[]
, byte[]> record, Exception exception) {

        if (exception instanceof RecordTooLargeException ) {

            return ProductionExceptionHandlerResponse.CONTINUE;

        }

        return ProductionExceptionHandlerResponse.FAIL;
    }
```

Add a no-op configure method, as before:

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

4. Next, add a `StreamsUncaughtExceptionHandler`. As with the other error handlers, you'll implement the `handle` method, checking whether the exception is an instance of `StreamsException`, which basically means that it's a wrapped user code exception. If it is, you'll extract the underlying exception.

```
public static class StreamsCustomUncaughtExceptionHandler implements Strea
msUncaughtExceptionHandler {

    @Override
    public StreamThreadExceptionResponse handle(Throwable exception) {
```

You also need code to evaluate the exception; you're looking for a specific exception that is transient in nature and that justifies replacing the stream thread.

```
if (exception instanceof StreamsException) {
    Throwable originalException = exception.getCause();
    if (originalException.getMessage().equals("Retryable transient error")
) {
        return StreamThreadExceptionResponse.REPLACE_THREAD;
    }
}
    return StreamThreadExceptionResponse.SHUTDOWN_CLIENT;
}
}
```

If that specific error is returned, we return a `REPLACE_THREAD` response. Otherwise, we shut down the instance with a `SHUTDOWN_CLIENT` response.

5. Now add the code to wire up the Kafka Streams application with your error handlers.

   First, add the `DeserializationExceptionHandler` implementation class to the configurations:

```
streamsProps.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG, StreamsDeserializationErrorHandler.class);
```

   Then add the `ProductionExceptionHandler` implementation class to the configurations:

```
streamsProps.put(StreamsConfig.DEFAULT_PRODUCTION_EXCEPTION_HANDLER_CLASS_CONFIG, StreamsRecordProducerErrorHandler.class);
```

6. Under the `streamWithErrorHandling` filter, create your `KafkaStreams` instance:

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsProps);
```

7. Then set the `StreamsUncaughtExceptionHandler` instance:

```
kafkaStreams.setUncaughtExceptionHandler(new StreamsCustomUncaughtExceptionHandler());
```

   Note that this is set directly on the `KafkaStreams` object and not in the configurations.

8. As with the other exercises, create the utility class and the code to run your application:

```
TopicLoader.runProducer();

kafkaStreams.start();
```

9. Start your application with this command

```
./gradlew runStreams -Pargs=errors
```

Then you'll quickly see a big, ugly stack trace. But the `StreamsUncaughtExceptionHandler` will do its job, and you'll see the application recover and print some processing output on the console.

10. This is the final exercise in the course, so make sure to delete your Confluent Cloud cluster. To do this, go to **Cluster settings** on the left-hand side menu, then click **Delete cluster**. Enter your cluster name, then select **Continue**.