Sample Chapter

Kafka Streams IN ACTION

Real-time apps and microservices with the Kafka Streams API

William P. Bejeck Jr.

Foreword by Neha Narkhede

MANNING

Kafka Streams in Action

by William P. Bejeck Jr.

Chapter 3

brief contents

3 Developing Kafka Streams

This chapter covers

- Introducing the Kafka Streams API
- Building Hello World for Kafka Streams
- Exploring the ZMart Kafka Streams application in depth
- Splitting an incoming stream into multiple streams

In chapter 1, you learned about the Kafka Streams library. You learned about building a topology of processing nodes, or a graph that transforms data as it's streaming into Kafka. In this chapter, you'll learn how to create this processing topology with the Kafka Streams API.

The Kafka Streams API is what you'll use to build Kafka Streams applications. You'll learn how to assemble Kafka Streams applications; but, more important, you'll gain a deeper understanding of how the components work together and how they can be used to achieve your stream-processing goals.

3.1 The Streams Processor API

The Kafka Streams DSL is the high-level API that enables you to build Kafka Streams applications quickly. The high-level API is very well thought out, and there are methods to handle most stream-processing needs out of the box, so you can create a sophisticated stream-processing program without much effort. At the heart of the high-level API is the `KStream` object, which represents the streaming key/value pair records.

Most of the methods in the Kafka Streams DSL return a reference to a `KStream` object, allowing for a fluent interface style of programming. Additionally, a good percentage of the `KStream` methods accept types consisting of single-method interfaces allowing for the use of Java 8 lambda expressions. Taking these factors into account, you can imagine the simplicity and ease with which you can build a Kafka Streams program.

Back in 2005, Martin Fowler and Eric Evans developed the concept of the *fluent interface*—an interface where the return value of a method call is the same instance that originally called the method (https://martinfowler.com/bliki/FluentInterface.html). This approach is useful when constructing objects with several parameters, such as `Person.builder().firstName("Beth").withLastName("Smith").withOccupation("CEO")`. In Kafka Streams, there is one small but important difference: the returned `KStream` object is a new instance, not the same instance that made the original method call.

There's also a lower-level API, the Processor API, which isn't as succinct as the Kafka Streams DSL but allows for more control. We'll cover the Processor API in chapter 6. With that introduction out of the way, let's dive into the requisite Hello World program for Kafka Streams.

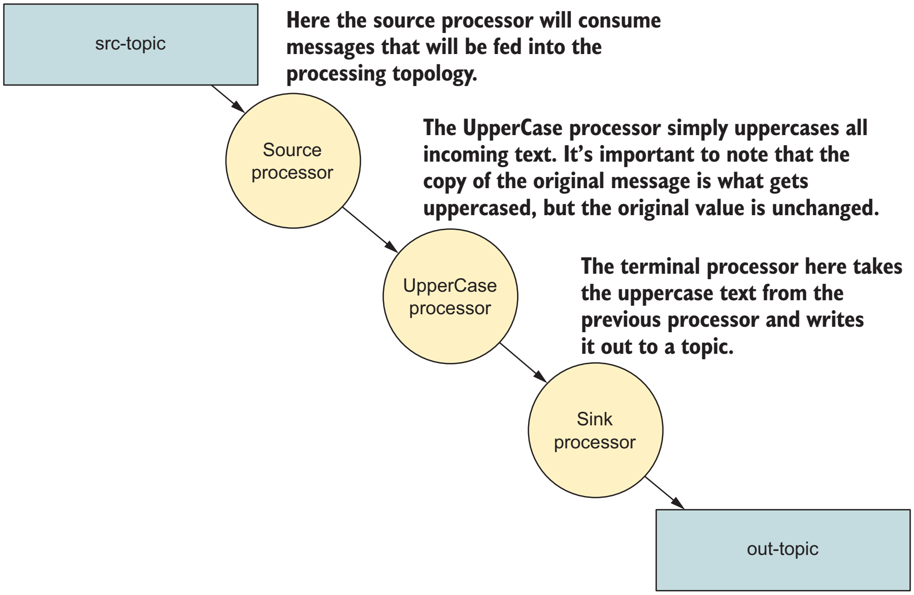
3.2 Hello World for Kafka Streams

For the first Kafka Streams example, we'll deviate from the problem outlined in chapter 1 to a simpler use case. This will get off the ground quickly so you can see how Kafka Streams works. We'll get back to the problem from chapter 1 later in section 3.1.1 for a more realistic, concrete example.

Your first program will be a toy application that takes incoming messages and converts them to uppercase characters, effectively yelling at anyone who reads the message. You'll call this the Yelling App.

Before diving into the code, let's take a look at the processing topology you'll assemble for this application. You'll follow the same pattern as in chapter 1, where you built up a processing graph topology with each node in the graph having a particular function. The main difference is that this graph will be simpler, as you can see in figure 3.1.

As you can see, you're building a simple processing graph—so simple that it resembles a linked list of nodes more than the typical tree-like structure of a graph. But there's enough here to give you strong clues about what to expect in the code. There will be a source node, a processor node transforming incoming text to uppercase, and a sink processor writing results out to a topic.



Here the source processor will consume messages that will be fed into the processing topology.

The UpperCase processor simply uppercases all incoming text. It's important to note that the copy of the original message is what gets uppercased, but the original value is unchanged.

The terminal processor here takes the uppercase text from the previous processor and writes it out to a topic.

Figure 3.1 Graph (topology) of the Yelling App

This is a trivial example, but the code shown here is representative of what you'll see in other Kafka Streams programs. In most of the examples, you'll see a similar structure:

1. Define the configuration items.
2. Create `Serde` instances, either custom or predefined.
3. Build the processor topology.
4. Create and start the `KStream`.

When we get into the more advanced examples, the principal difference will be in the complexity of the processor topology. With that in mind, it's time to build your first application.

3.2.1 *Creating the topology for the Yelling App*

The first step to creating any Kafka Streams application is to create a source node. The source node is responsible for consuming the records, from a topic, that will flow through the application. Figure 3.2 highlights the source node in the graph.

The following line of code creates the source, or parent, node of the graph.

Listing 3.1 Defining the source for the stream

```
KStream<String, String> simpleFirstStream = builder.stream("src-topic",
➥ Consumed.with(stringSerde, stringSerde));
```

The `simpleFirstStreamKStream` instance is set to consume messages written to the `src-topic` topic. In addition to specifying the topic name, you also provide `Serde`

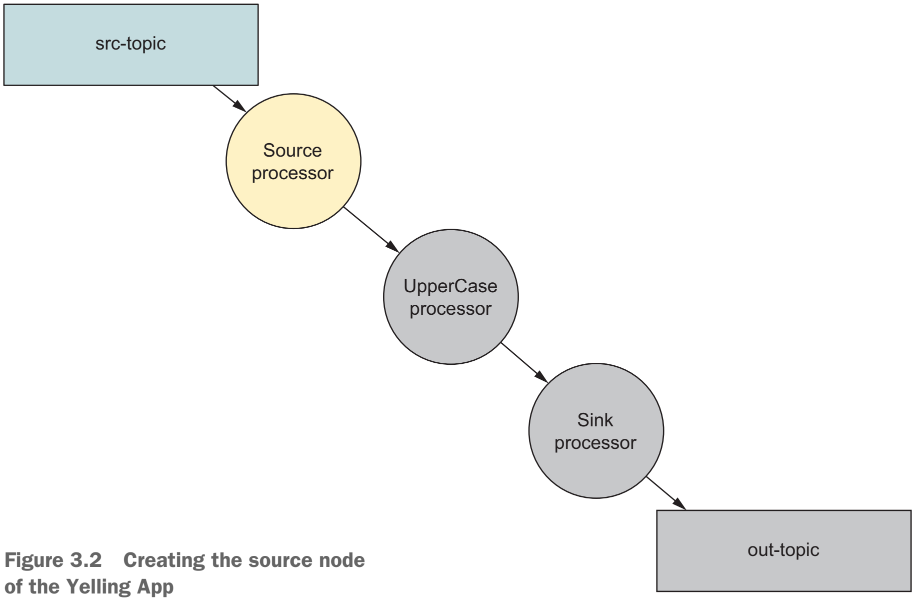


Figure 3.2 Creating the source node of the Yelling App

objects (via a `Consumed` instance) for deserializing the records from Kafka. You'll use the `Consumed` class for any optional parameters whenever you create a source node in Kafka Streams.

You now have a source node for your application, but you need to attach a processing node to make use of the data, as shown in figure 3.3. The code used to attach the

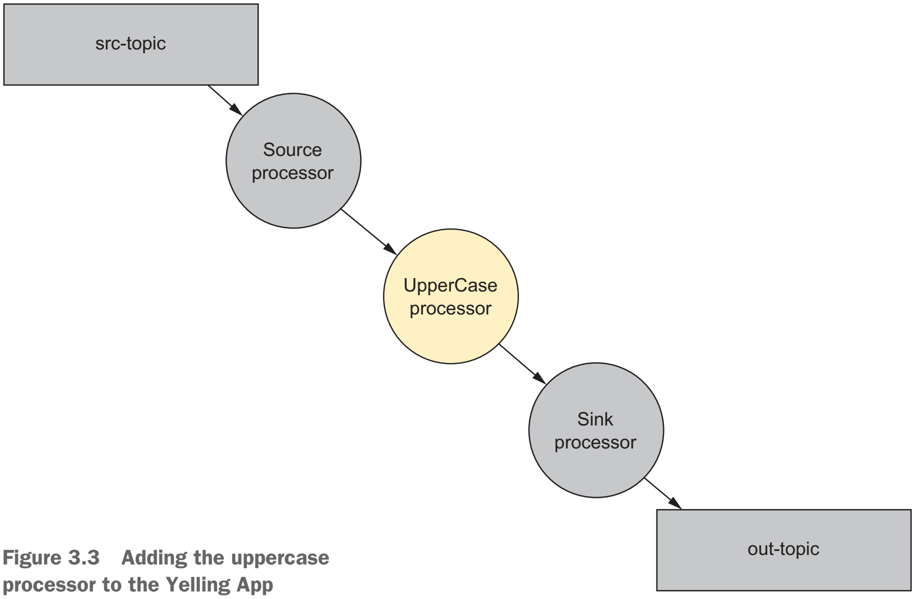


Figure 3.3 Adding the uppercase processor to the Yelling App

processor (a child node of the source node) is shown in the following listing. With this line, you create another `KStream` instance that's a child node of the parent node.

Listing 3.2 Mapping incoming text to uppercase

```
KStream<String, String> upperCasedStream =
➥ simpleFirstStream.mapValues(String::toUpperCase);
```

By calling the `KStream.mapValues` function, you're creating a new processing node whose inputs are the results of going through the `mapValues` call.

It's important to remember that you shouldn't modify the *original* value in the `ValueMapper` provided to `mapValues`. The `upperCasedStream` instance receives transformed copies of the initial value from the `simpleFirstStream.mapValues` call. In this case, it's uppercase text.

The `mapValues()` method takes an instance of the `ValueMapper<V, V1>` interface. The `ValueMapper` interface defines only one method, `ValueMapper.apply`, making it an ideal candidate for using a Java 8 lambda expression. This is what you've done here with `String::toUpperCase`, which is a method reference, an even shorter form of a Java 8 lambda expression.

> **NOTE** Many Java 8 tutorials are available for lambda expressions and method references. Good starting points can be found in Oracle's Java documentation: "Lambda Expressions" (http://mng.bz/J0Xm) and "Method References" (http://mng.bz/BaDW).

You could have used the form `s → s.toUpperCase()`, but because `toUpperCase` is an instance method on the `String` class, you can use a method reference.

Using lambda expressions instead of concrete implementations is a pattern you'll see over and over with the Streams Processor API in this book. Because most of the methods expect types that are single method interfaces, you can easily use Java 8 lambdas.

So far, your Kafka Streams application is consuming records and transforming them to uppercase. The final step is to add a sink processor that writes the results out to a topic. Figure 3.4 shows where you are in the construction of the topology.

The following code line adds the last processor in the graph.

Listing 3.3 Creating a sink node

```
upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```

The `KStream.to` method creates a sink-processing node in the topology. Sink processors write records back out to Kafka. This sink node takes records from the `upperCasedStream` processor and writes them to a topic named `out-topic`. Again, you provide `Serde` instances, this time for serializing records written to a Kafka topic. But in this case, you use a `Produced` instance, which provides optional parameters for creating a sink node in Kafka Streams.

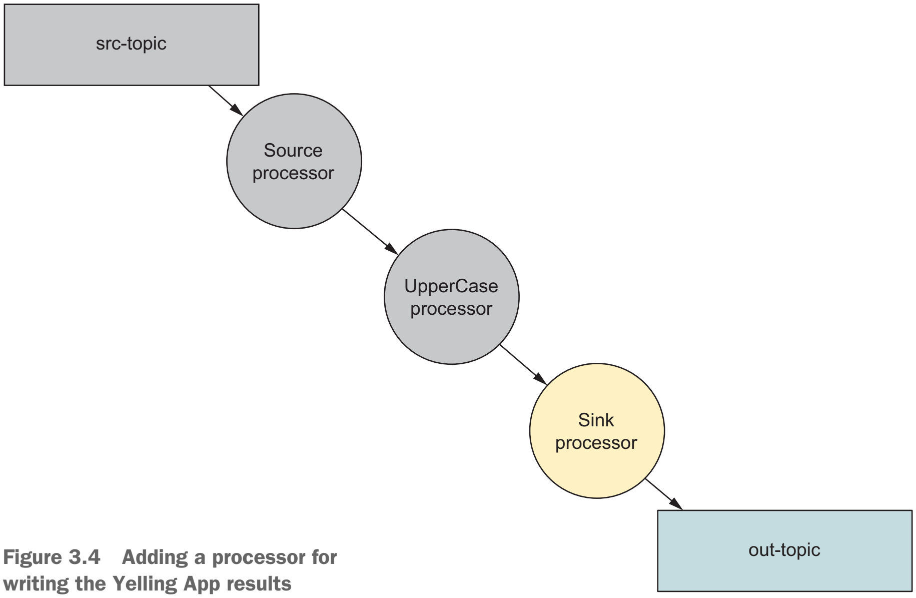


Figure 3.4 Adding a processor for writing the Yelling App results

> **NOTE** You don't always have to provide `Serde` objects to either the `Consumed` or `Produced` objects. If you don't, the application will use the serializer/deserializer listed in the configuration. Additionally, with the `Consumed` and `Produced` classes, you can specify a `Serde` for either the key or value only.

The preceding example uses three lines to build the topology:

```
KStream<String,String> simpleFirstStream =
➥ builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));
KStream<String, String> upperCasedStream =
➥ simpleFirstStream.mapValues(String::toUpperCase);
    upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```

Each step is on an individual line to demonstrate the different stages of the building process. But all methods in the `KStream` API that don't create terminal nodes (methods with a return type of `void`) return a new `KStream` instance, which allows you to use the fluent interface style of programming mentioned earlier. To demonstrate this idea, here's another way you could construct the Yelling App topology:

```
builder.stream("src-topic", Consumed.with(stringSerde, stringSerde))
➥ .mapValues(String::toUpperCase)
➥ .to("out-topic", Produced.with(stringSerde, stringSerde));
```

This shortens the program from three lines to one without losing any clarity or purpose. From this point forward, all the examples will be written using the fluent interface style unless doing so causes the clarity of the program to suffer.

You've built your first Kafka Streams topology, but we glossed over the important steps of configuration and Serde creation. We'll look at those now.

3.2.2 Kafka Streams configuration

Although Kafka Streams is highly configurable, with several properties you can adjust for your specific needs, the first example uses only two configuration settings, APPLICATION_ID_CONFIG and BOOTSTRAP_SERVERS_CONFIG:

```
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

Both settings are required because no default values are provided. Attempting to start a Kafka Streams program without these two properties defined will result in a ConfigException being thrown.

The StreamsConfig.APPLICATION_ID_CONFIG property identifies your Kafka Streams application, and it must be a unique value for the entire cluster. It also serves as a default value for the client ID prefix and group ID parameters if you don't set either value. The client ID prefix is the user-defined value that uniquely identifies clients connecting to Kafka. The group ID is used to manage the membership of a group of consumers reading from the same topic, ensuring that all consumers in the group can effectively read subscribed topics.

The StreamsConfig.BOOTSTRAP_SERVERS_CONFIG property can be a single hostname:port pair or multiple hostname:port comma-separated pairs. The value of this setting points the Kafka Streams application to the locaction of the Kafka cluster. We'll cover several more configuration items as we explore more examples in the book.

3.2.3 Serde creation

In Kafka Streams, the Serdes class provides convenience methods for creating Serde instances, as shown here:

```
Serde<String> stringSerde = Serdes.String();
```

This line is where you create the Serde instance required for serialization/deserialization using the Serdes class. Here, you create a variable to reference the Serde for repeated use in the topology. The Serdes class provides default implementations for the following types:

- String
- Byte array
- Long
- Integer
- Double

Implementations of the Serde interface are extremely useful because they contain the serializer and deserializer, which keeps you from having to specify four parameters (key serializer, value serializer, key deserializer, and value deserializer) every time you need to provide a Serde in a KStream method. In an upcoming example, you'll create a Serde implementation to handle serialization/deserialization of more-complex types.

Let's take a look at the whole program you just put together. You can find the source in src/main/java/bbejeck/chapter_3/KafkaStreamsYellingApp.java (source code can be found on the book's website here: https://manning.com/books/kafka-streams-in-action).

Listing 3.4 Hello World: the Yelling App

```
public class KafkaStreamsYellingApp {

 public static void main(String[] args) {


  Properties props = new Properties();

  props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
  props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");


  StreamsConfig streamingConfig = new StreamsConfig(props);

  Serde<String> stringSerde = Serdes.String();

  StreamsBuilder builder = new StreamsBuilder();

  KStream<String, String> simpleFirstStream = builder.stream("src-topic",
  ➥ Consumed.with(stringSerde, stringSerde));

  KStream<String, String> upperCasedStream =
  ➥ simpleFirstStream.mapValues(String::toUpperCase);

  upperCasedStream.to( "out-topic",
  ➥ Produced.with(stringSerde, stringSerde));

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

  kafkaStreams.start();
  Thread.sleep(35000);
  LOG.info("Shutting down the Yelling APP now");
  kafkaStreams.close();

  }
}
```

Creates the StreamsConfig with the given properties

Properties for configuring the Kafka Streams program

Creates the Serdes used to serialize/deserialize keys and values

Creates the StreamsBuilder instance used to construct the processor topology

Creates the actual stream with a source topic to read from (the parent node in the graph)

A processor using a Java 8 method handle (the first child node in the graph)

Writes the transformed output to another topic (the sink node in the graph)

Kicks off the Kafka Streams threads

You've now constructed your first Kafka Streams application. Let's quickly review the steps involved, as it's a general pattern you'll see in most of your Kafka Streams applications:

1. Create a StreamsConfig instance.
2. Create a Serde object.

3. Construct a processing topology.
4. Start the Kafka Streams program.

Apart from the general construction of a Kafka Streams application, a key takeaway here is to use lambda expressions whenever possible, to make your programs more concise.

We'll now move on to a more complex example that will allow us to explore more of the Streams Processor API. The example will be new, but the scenario is one you're already familiar with: ZMart data-processing goals.

3.3 Working with customer data

In chapter 1, we discussed ZMart's new requirements for processing customer data, intended to help ZMart do business more efficiently. We demonstrated how you could build a topology of processors that would work on purchase records as they come streaming in from transactions in ZMart stores. Figure 3.5 shows the completed graph again.

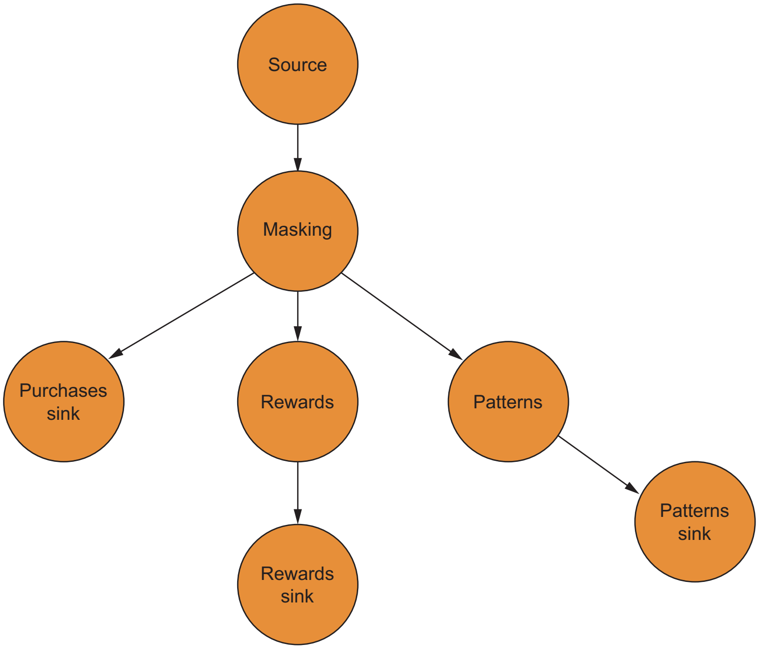


Figure 3.5 Topology for ZMart Kafka Streams program

Let's briefly review the requirements for the streaming program, which will also serve as a good description of what the program will do:

- All records need to have credit card numbers protected, in this case by masking the first 12 digits.
- You need to extract the items purchased and the ZIP code to determine purchase patterns. This data will be written out to a topic.

- You need to capture the customer's ZMart member number and the amount spent and write this information to a topic. Consumers of the topic will use this data to determine rewards.
- You need to write the entire transaction out to topic, which will be consumed by a storage engine for ad hoc analysis.

As in the Yelling App, you'll combine the fluent interface approach with Java 8 lambdas when building the application. Although it's sometimes clear that the return type of a method call is a `KStream` object, other times it may not be. Keep in mind that the majority of the methods in the `KStream` API return *new* `KStream` instances. Now, let's build a streaming application that will satisfy ZMart's business requirements.

3.3.1 Constructing a topology

Let's dive into building the processing topology. To help make the connection between the code you'll create here and the processing topology graph from chapter 1, I'll highlight the part of the graph that you're currently working on.

BUILDING THE SOURCE NODE

You'll start by building the source node and first processor of the topology by chaining two calls to the `KStream` API together (highlighted in figure 3.6). It should be fairly obvious by now what the role of the origin node is. The first processor in the topology will be responsible for masking credit card numbers to protect customer privacy.

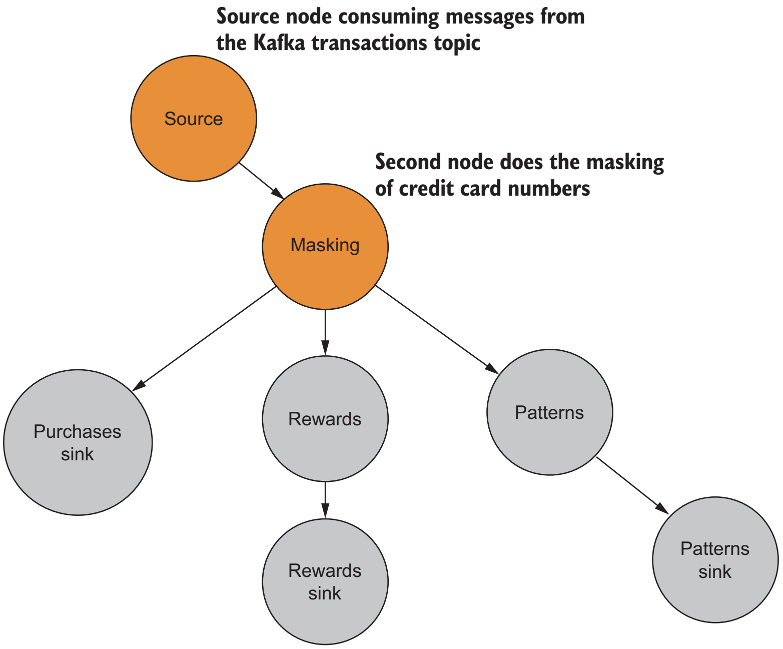

Source node consuming messages from the Kafka transactions topic
Source
Second node does the masking of credit card numbers
Masking
Purchases sink
Rewards
Patterns
Rewards sink
Patterns sink

Figure 3.6 The source processor consumes from a Kafka topic, and it feeds the masking processor exclusively, making it the source for the rest of the topology.

Listing 3.5 Building the source node and first processor

```
KStream<String,Purchase> purchaseKStream =
➥ streamsBuilder.stream("transactions",
➥ Consumed.with(stringSerde, purchaseSerde))
➥ .mapValues(p -> Purchase.builder(p).maskCreditCard().build());
```

You create the source node with a call to the `StreamsBuilder.stream` method using a default `String` serde, a custom serde for `Purchase` objects, and the name of the topic that's the source of the messages for the stream. In this case, you only specify one topic, but you could have provided a comma-separated list of names or a regular expression to match topic names instead.

In this listing 3.5, you provide `Serdes` with a `Consumed` instance, but you could have left that out and only provided the topic name and relied on the default `Serdes` provided via configuration parameters.

The next immediate call is to the `KStream.mapValues` method, taking a `ValueMapper<V, V1>` instance as a parameter. Value mappers take a single parameter of one type (a `Purchase` object, in this case) and map that object to a to a new value, possibly of another type. In this example, `KStream.mapValues` returns an object of the same type (`Purchase`), but with a masked credit card number.

Note that when using the `KStream.mapValues` method, the original key is unchanged and isn't factored into mapping a new value. If you wanted to generate a new key/value pair or include the key in producing a new value, you'd use the `KStream.map` method that takes a `KeyValueMapper<K, V, KeyValue<K1, V1>>` instance.

HINTS ABOUT FUNCTIONAL PROGRAMMING

An important concept to keep in mind with the `map` and `mapValues` functions is that they're expected to operate without side effects, meaning the functions don't modify the object or value presented as a parameter. This is because of the functional programming aspects in the `KStream` API. Functional programming is a deep topic, and a full discussion is beyond the scope of this book, but we'll briefly look at two central principles of functional programming here.

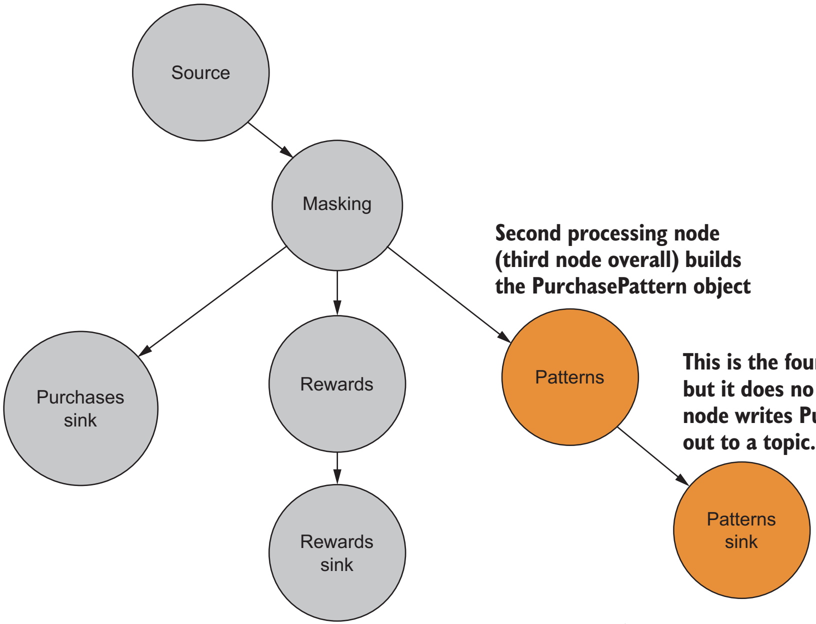
The first principle is avoiding state modification. If an object requires a change or update, you pass the object to a function, and a copy or entirely new instance is made, containing the desired changes or updates. In listing 3.5, the lambda passed to `KStream.mapValues` is used to update the `Purchase` object with a masked credit card number. The credit card field on the original `Purchase` object is left unchanged.

The second principle is building complex operations by composing several smaller single-purpose functions together. The composition of functions is a pattern you'll frequently see when working with the `KStream` API.

> **DEFINITION** For the purposes of this book, I define *functional programming* as a programming approach in which functions are first-class objects. Furthermore, functions are expected to avoid creating side effects, such as modifying state or mutable objects.

Building the second processor

Now you'll build the second processor, responsible for extracting pattern data from a topic, which ZMart can use to determine purchase patterns in regions of the country. You'll also add a sink node responsible for writing the pattern data to a Kafka topic. The construction of these is demonstrated in figure 3.7.



Second processing node (third node overall) builds the PurchasePattern object

This is the fourth node overall, but it does no processing. This node writes PurchasePattern out to a topic.

Again you see two nodes in the graph, but with the fluent style of programming in Kafka Streams, sometimes it's easy to overlook the fact that you're creating two nodes.

Figure 3.7 The second processor builds purchase-pattern information. The sink node writes the `PurchasePattern` object out to a Kafka topic.

In listing 3.6, you can see the `purchaseKStream` processor using the familiar `mapValues` call to create a new `KStream` instance. This new `KStream` will start to receive `PurchasePattern` objects created as a result of the `mapValues` call.

Listing 3.6 Second processor and a sink node that writes to Kafka

```
KStream<String, PurchasePattern> patternKStream =
➥ purchaseKStream.mapValues(purchase ->
➥ PurchasePattern.builder(purchase).build());

patternKStream.to("patterns",
➥ Produced.with(stringSerde,purchasePatternSerde));
```

Here, you declare a variable to hold the reference of the new `KStream` instance, because you'll use it to print the results of the stream to the console with a `print` call. This is very useful during development and for debugging. The purpose-patterns processor forwards the records it receives to a child node of its own, defined by the method call `KStream.to`, writing to the `patterns` topic. Note the use of a `Produced` object to provide the previously built `Serde`.

The `KStream.to` method is a mirror image of the `KStream.source` method. Instead of setting a source for the topology to read from, the `KStream.to` method defines a sink node that's used to write the data from a `KStream` instance to a Kafka topic. The `KStream.to` method also provides overloaded versions in which you can leave out the `Produced` parameter and use the default `Serdes` defined in the configuration. One of the optional parameters you can set with the `Produced` class is `StreamPartitioner`, which we'll discuss next.

Building the third processor

The third processor in the topology is the customer rewards accumulator node shown in figure 3.8, which will let ZMart track purchases made by members of their preferred customer club. The rewards accumulator sends data to a topic consumed by applications at ZMart HQ to determine rewards when customers complete purchases.

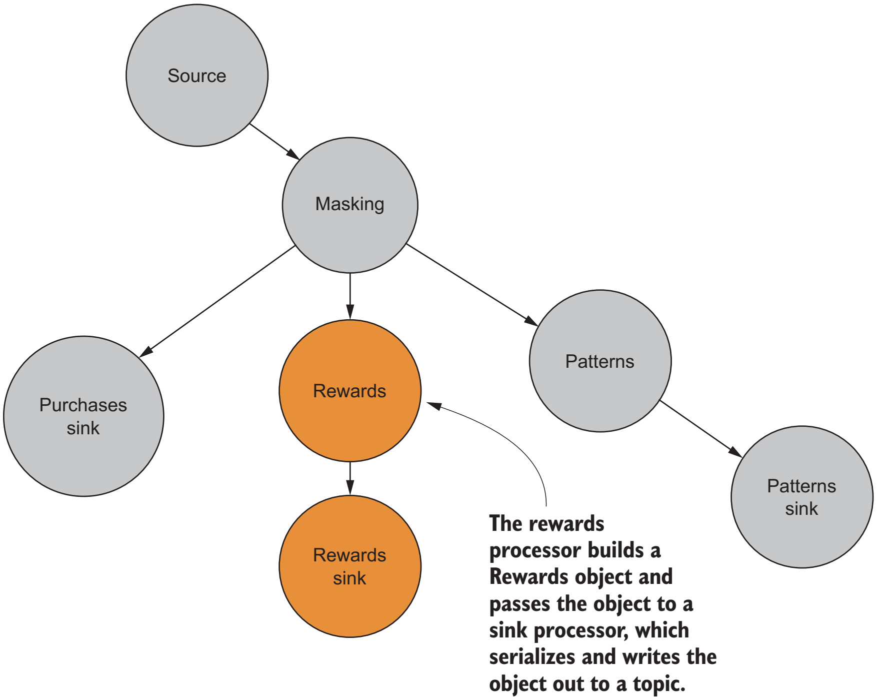

Source
Masking
Purchases sink
Rewards
Patterns
Rewards sink
Patterns sink
The rewards processor builds a Rewards object and passes the object to a sink processor, which serializes and writes the object out to a topic.

Figure 3.8 The third processor creates the `RewardAccumulator` object from the purchase data. The terminal node writes the results out to a Kafka topic.

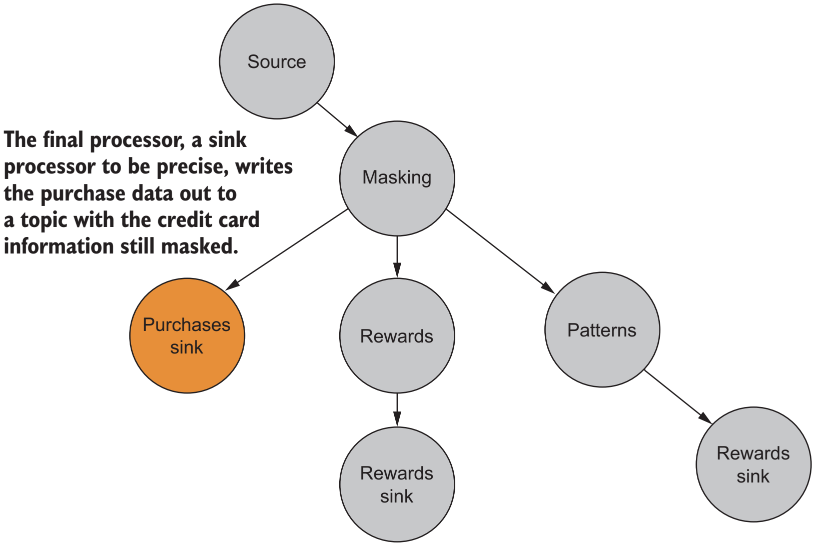
Listing 3.7 Third processor and a terminal node that writes to Kafka

```
KStream<String, RewardAccumulator> rewardsKStream =
➥ purchaseKStream.mapValues(purchase ->
➥ RewardAccumulator.builder(purchase).build());
rewardsKStream.to("rewards",
➥ Produced.with(stringSerde,rewardAccumulatorSerde));
```

You build the rewards accumulator processor using what should be by now a familiar pattern: creating a new `KStream` instance that maps the raw purchase data contained in the record to a new object type. You also attach a sink node to the rewards accumulator so the results of the rewards `KStream` can be written to a topic and used for determining customer reward levels.

BUILDING THE LAST PROCESSOR

Finally, you'll take the first `KStream` you created, `purchaseKStream`, and attach a sink node to write out the raw purchase records (with credit cards masked, of course) to a topic called `purchases`. The `purchases` topic will be used to feed into a NoSQL store such as Cassandra (http://cassandra.apache.org/), Presto (https://prestodb.io/), or Elastic Search (www.elastic.co/webinars/getting-started-elasticsearch) to perform ad hoc analysis. Figure 3.9 shows the final processor.



The final processor, a sink processor to be precise, writes the purchase data out to a topic with the credit card information still masked.

Figure 3.9 The last node writes out the entire purchase transaction to a topic whose consumer is a NoSQL data store.

Listing 3.8 Final processor

```
purchaseKStream.to("purchases", Produced.with(stringSerde, purchaseSerde));
```

Now that you've built the application piece by piece, let's look at the entire application (src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsApp.java). You'll quickly notice it's more complicated than the previous Hello World (the Yelling App) example.

Listing 3.9 ZMart customer purchase `KStream` program

```
public class ZMartKafkaStreamsApp {

  public static void main(String[] args) {
    // some details left out for clarity

    StreamsConfig streamsConfig = new StreamsConfig(getProperties());

    JsonSerializer<Purchase> purchaseJsonSerializer = new
➥ JsonSerializer<>();
    JsonDeserializer<Purchase> purchaseJsonDeserializer =
➥ new JsonDeserializer<>(Purchase.class);
    Serde<Purchase> purchaseSerde =
➥ Serdes.serdeFrom(purchaseJsonSerializer, purchaseJsonDeserializer);
    //Other Serdes left out for clarity

    Serde<String> stringSerde = Serdes.String();

    StreamsBuilder streamsBuilder = new StreamsBuilder();

    KStream<String,Purchase> purchaseKStream =
➥ streamsBuilder.stream("transactions",
➥ Consumed.with(stringSerde, purchaseSerde))
➥ .mapValues(p -> Purchase.builder(p).maskCreditCard().build());

    KStream<String, PurchasePattern> patternKStream =
➥ purchaseKStream.mapValues(purchase ->
➥ PurchasePattern.builder(purchase).build());

    patternKStream.to("patterns",
➥ Produced.with(stringSerde,purchasePatternSerde));

    KStream<String, RewardAccumulator> rewardsKStream =
➥ purchaseKStream.mapValues(purchase ->
➥ RewardAccumulator.builder(purchase).build());

     rewardsKStream.to("rewards",
➥ Produced.with(stringSerde,rewardAccumulatorSerde));

    purchaseKStream.to("purchases",
➥ Produced.with(stringSerde,purchaseSerde));

    KafkaStreams kafkaStreams =
➥ new KafkaStreams(streamsBuilder.build(),streamsConfig);
    kafkaStreams.start();

  }
```

Creates the Serde; the data format is JSON.

Builds the source and first processor

Builds the PurchasePattern processor

Builds the RewardAccumulator processor

Builds the storage sink, the topic used by the storage consumer

NOTE I've left out some details in listing 3.9 for clarity. The code examples in the book aren't necessarily meant to stand on their own. The source code that accompanies this book provides the full examples.

As you can see, this example is a little more involved than the Yelling App, but it has a similar flow. Specifically, you still performed the following steps:

- Create a `StreamsConfig` instance.
- Build one or more `Serde` instances.
- Construct the processing topology.
- Assemble all the components and start the Kafka Streams program.

In this application, I've mentioned using a `Serde`, but I haven't explained why or how you create them. Let's take some time now to discuss the role of the `Serde` in a Kafka Streams application.

3.3.2 Creating a custom Serde

Kafka transfers data in byte array format. Because the data format is JSON, you need to tell Kafka how to convert an object first into JSON and then into a byte array when it sends data to a topic. Conversely, you need to specify how to convert consumed byte arrays into JSON, and then into the object type your processors will use. This conversion of data to and from different formats is why you need a `Serde`. Some serdes are provided out of the box by the Kafka client dependency, (`String`, `Long`, `Integer`, and so on), but you'll need to create custom serdes for other objects.

In the first example, the Yelling App, you only needed a serializer/deserializer for strings, and an implementation is provided by the `Serdes.String()` factory method. In the ZMart example, however, you need to create custom `Serde` instances, because the types of the objects are arbitrary. We'll look at what's involved in building a `Serde` for the `Purchase` class. We won't cover the other `Serde` instances, because they follow the same pattern, just with different types.

Building a `Serde` requires implementations of the `Deserializer<T>` and `Serializer<T>` interfaces. We'll use the implementations in listings 3.10 and 3.11 throughout the examples. Also, you'll use the Gson library from Google to convert objects to and from JSON. Here's the serializer, which you can find in src/main/java/bbejeck/util/serializer/JsonSerializer.java.

Listing 3.10 Generic serializer

```
public class JsonSerializer<T> implements Serializer<T> {

    private Gson gson = new Gson();

    @Override
    public void configure(Map<String, ?> map, boolean b) {

    }
```

Creates the Gson object

```
    @Override
    public byte[] serialize(String topic, T t) {
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));
    }

    @Override
    public void close() {

    }
}
```

Serializes an object to bytes

For serialization, you first convert an object to JSON, and then get the bytes from the string. To handle the conversions from and to JSON, the example uses Gson (https://github.com/google/gson).

For the deserializing process, you take different steps: create a new string from a byte array, and then use Gson to convert the JSON string into a Java object. This generic deserializer can be found in src/main/java/bbejeck/util/serializer/JsonDeserializer.java.

Listing 3.11 Generic deserializer

```
public class JsonDeserializer<T> implements Deserializer<T> {

    private Gson gson = new Gson();
    private Class<T> deserializedClass;

    public JsonDeserializer(Class<T> deserializedClass) {
        this.deserializedClass = deserializedClass;
    }
```

Creates the Gson object

Instance variable of Class to deserialize

```
    public JsonDeserializer() {
    }

    @Override
    @SuppressWarnings("unchecked")
    public void configure(Map<String, ?> map, boolean b) {
        if(deserializedClass == null) {
            deserializedClass = (Class<T>) map.get("serializedClass");
        }
    }

    @Override
    public T deserialize(String s, byte[] bytes) {
         if(bytes == null){
             return null;
         }

         return gson.fromJson(new String(bytes),deserializedClass);

    }
```

Deserializes bytes to an instance of expected Class

```
    @Override
    public void close() {

    }
}
```

Now, let's go back to the following lines from listing 3.9:

```
JsonDeserializer<Purchase> purchaseJsonDeserializer =
➥ new JsonDeserializer<>(Purchase.class);
JsonSerializer<Purchase> purchaseJsonSerializer =
➥ new JsonSerializer<>();
Serde<Purchase> purchaseSerde =
➥ Serdes.serdeFrom(purchaseJsonSerializer,purchaseJsonDeserializer);
```

Creates the Deserializer for the Purchase class

Creates the Serializer for the Purchase class

Creates the Serde for Purchase objects

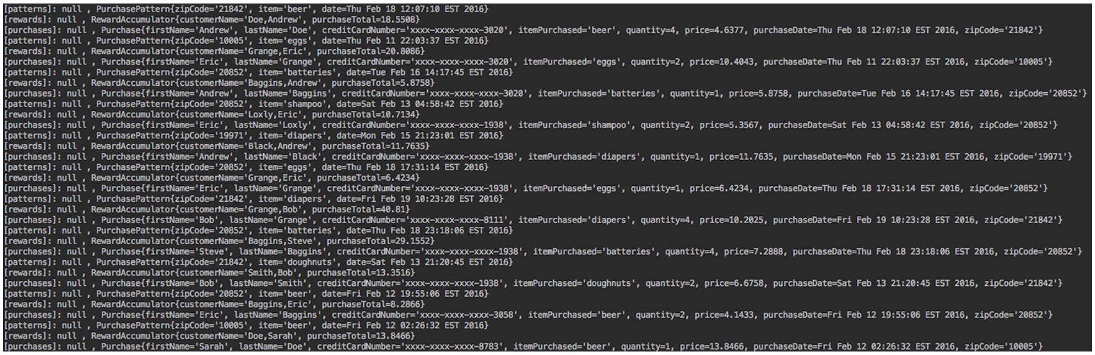
As you can see, a `Serde` object is useful because it serves as a container for the serializer and deserializer for a given object.

We've covered a lot of ground so far in developing a Kafka Streams application. We still have much more to cover, but let's pause for a moment and talk about the development process itself and how you can make life easier for yourself while developing a Kafka Streams application.

3.4 Interactive development

You've built the graph to process purchase records from ZMart in a streaming fashion, and you have three processors that write out to individual topics. During development it would certainly be possible to have a console consumer running to view results, but it would be good to have a more convenient solution, like the ability to watch data flowing through the topology in the console, as shown in figure 3.10.

There's a method on the `KStream` interface that can be useful during development: the `KStream.print` method, which takes an instance of the `Printed<K, V>` class.



```
[patterns]: null , PurchasePattern{zipCode='21842', item='beer', date=Thu Feb 18 12:07:10 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Doe,Andrew', purchaseTotal=18.5508}
[purchases]: null , Purchase{firstName='Andrew', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-3020', itemPurchased='beer', quantity=4, price=4.6377, purchaseDate=Thu Feb 18 12:07:10 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='10005', item='eggs', date=Thu Feb 11 22:03:37 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Eric', purchaseTotal=20.8086}
[purchases]: null , Purchase{firstName='Eric', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-3020', itemPurchased='eggs', quantity=2, price=10.4043, purchaseDate=Thu Feb 11 22:03:37 EST 2016, zipCode='10005'}
[patterns]: null , PurchasePattern{zipCode='20852', item='batteries', date=Tue Feb 16 14:17:45 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Baggins,Andrew', purchaseTotal=5.8758}
[purchases]: null , Purchase{firstName='Andrew', lastName='Baggins', creditCardNumber='xxxx-xxxx-xxxx-3020', itemPurchased='batteries', quantity=1, price=5.8758, purchaseDate=Tue Feb 16 14:17:45 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='20852', item='shampoo', date=Sat Feb 13 04:58:42 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Loxly,Eric', purchaseTotal=10.7134}
[purchases]: null , Purchase{firstName='Eric', lastName='Loxly', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='shampoo', quantity=2, price=5.3567, purchaseDate=Sat Feb 13 04:58:42 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='19971', item='diapers', date=Mon Feb 15 21:23:01 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Black,Andrew', purchaseTotal=11.7635}
[purchases]: null , Purchase{firstName='Andrew', lastName='Black', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='diapers', quantity=1, price=11.7635, purchaseDate=Mon Feb 15 21:23:01 EST 2016, zipCode='19971'}
[patterns]: null , PurchasePattern{zipCode='20852', item='eggs', date=Thu Feb 18 17:31:14 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Eric', purchaseTotal=6.4234}
[purchases]: null , Purchase{firstName='Eric', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='eggs', quantity=1, price=6.4234, purchaseDate=Thu Feb 18 17:31:14 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='21842', item='diapers', date=Fri Feb 19 10:23:28 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Bob', purchaseTotal=40.81}
[purchases]: null , Purchase{firstName='Bob', lastName='Grange', creditCardNumber='xxxx-xxxx-xxxx-8111', itemPurchased='diapers', quantity=4, price=10.2025, purchaseDate=Fri Feb 19 10:23:28 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='20852', item='batteries', date=Thu Feb 18 23:18:06 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Baggins,Steve', purchaseTotal=29.1552}
[purchases]: null , Purchase{firstName='Steve', lastName='Baggins', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='batteries', quantity=4, price=7.2888, purchaseDate=Thu Feb 18 23:18:06 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='21842', item='doughnuts', date=Sat Feb 13 21:20:45 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Smith,Bob', purchaseTotal=13.3516}
[purchases]: null , Purchase{firstName='Bob', lastName='Smith', creditCardNumber='xxxx-xxxx-xxxx-1938', itemPurchased='doughnuts', quantity=2, price=6.6758, purchaseDate=Sat Feb 13 21:20:45 EST 2016, zipCode='21842'}
[patterns]: null , PurchasePattern{zipCode='20852', item='beer', date=Fri Feb 12 19:55:06 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Baggins,Eric', purchaseTotal=8.2866}
[purchases]: null , Purchase{firstName='Eric', lastName='Baggins', creditCardNumber='xxxx-xxxx-xxxx-3058', itemPurchased='beer', quantity=2, price=4.1433, purchaseDate=Fri Feb 12 19:55:06 EST 2016, zipCode='20852'}
[patterns]: null , PurchasePattern{zipCode='10005', item='beer', date=Fri Feb 12 02:26:32 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Doe,Sarah', purchaseTotal=13.8466}
[purchases]: null , Purchase{firstName='Sarah', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-8783', itemPurchased='beer', quantity=1, price=13.8466, purchaseDate=Fri Feb 12 02:26:32 EST 2016, zipCode='10005'}
```

Figure 3.10 A great tool while you're developing is the capacity to print the data that's output from each node to the console. To enable printing to the console, just replace any of the `to` methods with a call to `print`.

`Printed` provides two static methods allowing you print to stdout, `Printed.toSysOut()`, or to write results to a file, `Printed.toFile(filePath)`.

Additionally, you can label your printed results by chaining the `withLabel()` method, allowing you to print an initial header with the records. This is useful when you're dealing with results from different processors. It's important that your objects provide a meaningful `toString` implementation to create useful results when printing your stream either to the console or a file.

Finally, if you don't want to use `toString`, or you want to customize how Kafka Streams prints records, there's the `Printed.withKeyValueMapper` method, which takes a `KeyValueMapper` instance so you can format your records in any way you want. The same caveat I mentioned earlier—that you shouldn't modify the original records—applies here as well.

In this book, I focus on printing records to the console for all examples. Here are some examples of using `KStream.print` in listing 3.11:

Sets up to print the RewardAccumulator transformation to the console

Sets up to print the PurchasePattern transformation to the console

```
patternKStream.print(Printed.<String, PurchasePattern>toSysOut()
➥  .withLabel("patterns"));

rewardsKStream.print(Printed.<String, RewardAccumulator>toSysOut()
➥  .withLabel("rewards"));

purchaseKStream.print(Printed.<String, Purchase>toSysOut()
➥  .withLabel("purchases"));
```

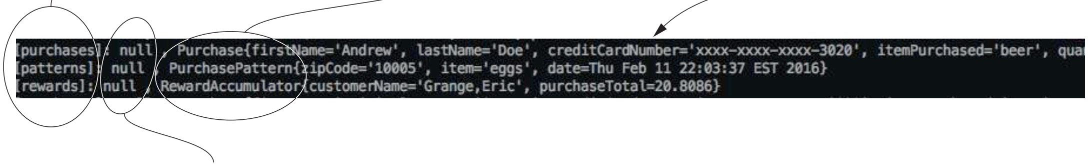
Prints the purchase data to the console

Let's take a quick look at the output you'll see on the screen (figure 3.11) and how it can help you during development. With printing enabled, you can run the Kafka Streams application directly from your IDE as you make changes, stop and start the application, and confirm that the output is what you expect. This is no substitute for unit and integration tests, but viewing streaming results directly as you develop is a great tool.

Name(s) given to the print statement, helpful to make this the same as the topic

The values for the records. Note that these are JSON strings and the Purchase, PurchasePattern, and RewardAccumulator objects defined toString methods to get this rendering on the console.

Note the masked credit card number!



```
[purchases]: null , Purchase{firstName='Andrew', lastName='Doe', creditCardNumber='xxxx-xxxx-xxxx-3020', itemPurchased='beer', qua
[patterns]: null , PurchasePattern{zipCode='10005', item='eggs', date=Thu Feb 11 22:03:37 EST 2016}
[rewards]: null , RewardAccumulator{customerName='Grange,Eric', purchaseTotal=20.8086}
```

The keys for the records, which are null in this case

Figure 3.11 This a detailed view of the data on the screen. With printing to the console enabled, you'll quickly see if your processors are working correctly.

One downside of using the `print()` method is that it creates a terminal node, meaning you can't embed it in a chain of processors. You need to have a separate statement. However, there's also the `KStream.peek` method, which takes a `ForeachAction` instance as a parameter and returns a new `KStream` instance. The `ForeachAction` interface has one method, `apply()`, which has a return type of `void`, so nothing from `KStream.peek` is forwarded downstream, making it ideal for operations like printing. You can embed it in a chain of processors without the need for a separate print statement. You'll see the `KStream.peek` method used in this manner in other examples in the book.

3.5 Next steps

At this point, you have your Kafka Streams purchase-analysis program running well. Other applications have also been developed to consume the messages written to the `patterns`, `rewards`, and `purchases` topics, and the results for ZMart have been good. But alas, no good deed goes unpunished. Now that the ZMart executives can see what your streaming program can provide, a slew of new requirements come your way.

3.5.1 New requirements

You now have new requirements for each of the three categories of results you're producing. The good news is that you'll still use the same source data. You're being asked to refine, and in some cases further break down, the data you're providing. The new requirements may be able to be applied to current topics, or they may require you to create entirely new topics:

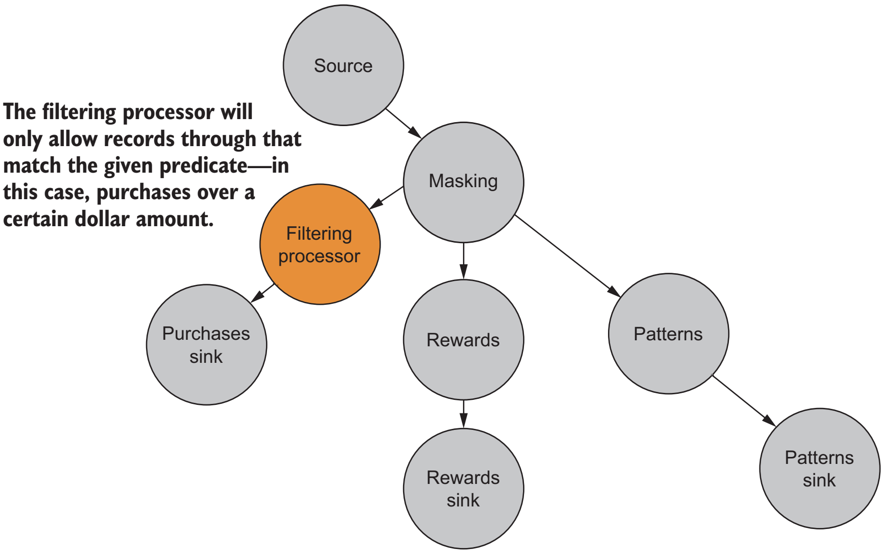
- Purchases under a certain dollar amount need to be filtered out. Upper management isn't much interested in the small purchases for general daily articles.
- ZMart has expanded and has bought an electronics chain and a popular coffee house chain. All purchases from these new stores will flow through the streaming application you've set up. You need to send the purchases from these new subsidiaries to their topics.
- The NoSQL solution you've chosen stores items in key/value format. Although Kafka also uses key/value pairs, the records coming into your Kafka cluster don't have keys defined. You need to generate a key for each record before the topology forwards it to the `purchases` topic.

More requirements will inevitably come your way, but you can start to work on the current set of new requirements now. If you look through the `KStream` API, you'll be relieved to see that there are several methods already defined that will make fulfilling these new demands easy.

NOTE From this point forward, all code examples are pared down to the essentials to maximize clarity. Unless there's something new to introduce, you can assume that the configuration and setup code remain the same. These truncated examples aren't meant to stand alone—the full code listing for this example can be found in src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsAdvancedReqsApp.java.

Filtering purchases

Let's start with filtering out purchases that don't reach the minimum threshold. To remove low-dollar purchases, you'll need to insert a filter-processing node between the `KStream` instance and the sink node. You'll update the processor topology graph as shown in figure 3.12.



The filtering processor will only allow records through that match the given predicate—in this case, purchases over a certain dollar amount.

Figure 3.12 You're placing a processor between the masking processor and the terminal node that writes to Kafka. This filtering processor will drop purchases under a given dollar amount.

You can use the `KStream` method, which takes a `Predicate<K,V>` instance as a parameter. Although you're chaining method calls together here, you're creating a new processing node in the topology.

Listing 3.12 Filtering on `KStream`

```
KStream<Long, Purchase> filteredKStream =
  ➥ purchaseKStream((key, purchase) ->
  ➥ purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);
```

This code filters purchases that are less than $5.00 and selects the purchase date as a long value for a key.

The `Predicate` interface has one method defined, `test()`, which takes two parameters—the key and the value—although, at this point, you only need to use the value. Again, you can use a Java 8 lambda in place of a concrete type defined in the `KStream` API.

DEFINITION If you're familiar with functional programming, you should feel right at home with the `Predicate` interface. If the term *predicate* is new to you, it's nothing more than a given statement, such as `x < 100`. An object either matches the predicate statement or doesn't.

Additionally, you want to use the purchase timestamp as a key, so you use the `selectKey` processor, which uses the `KeyValueMapper` mentioned in section 3.4 to extract the purchase date as a long value. I cover details about selecting the key in the section "Generating a key."

A mirror-image function, `KStreamNot`, performs the same filtering functionality but in reverse. Only records that *don't* match the given predicate are processed further in the topology.

SPLITTING/BRANCHING THE STREAM

Now you need to split the stream of purchases into separate streams that can write to different topics. Fortunately, the `KStream.branch` method is perfect. The `KStream.branch` method takes an arbitrary number of `Predicate` instances and returns an array of `KStream` instances. The size of the returned array matches the number of predicates supplied in the call.

In the previous change, you modified an existing leaf on the processing topology. With this requirement to branch the stream, you'll create brand-new leaf nodes on the graph of processing nodes, as shown in figure 3.13.

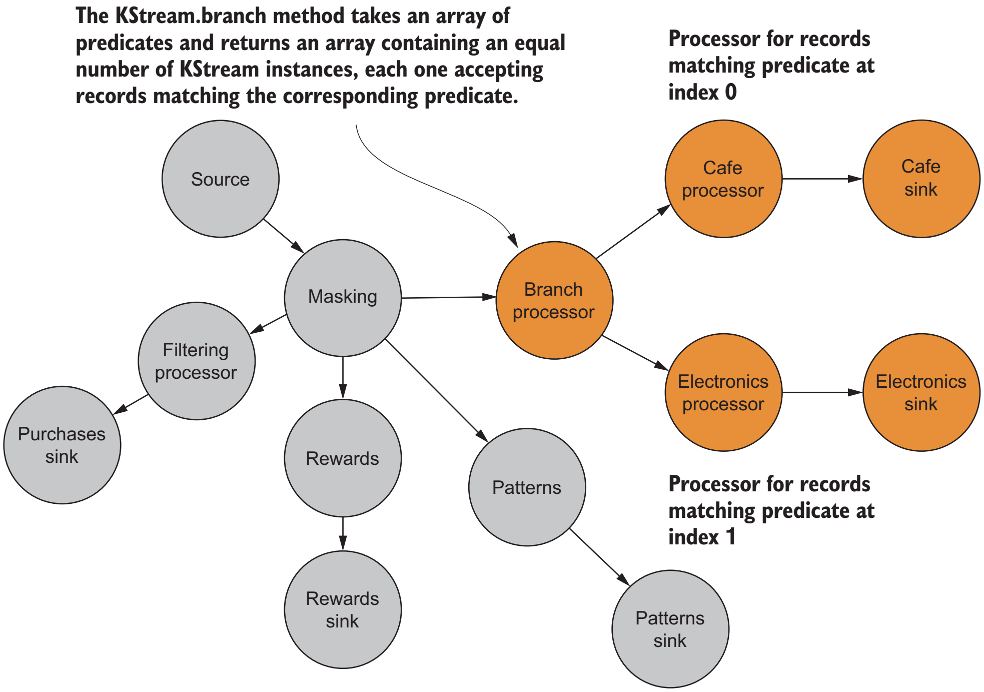

The KStream.branch method takes an array of predicates and returns an array containing an equal number of KStream instances, each one accepting records matching the corresponding predicate.
Processor for records matching predicate at index 0
Source
Cafe processor
Cafe sink
Masking
Branch processor
Filtering processor
Electronics processor
Electronics sink
Purchases sink
Rewards
Patterns
Processor for records matching predicate at index 1
Rewards sink
Patterns sink

Figure 3.13 The branch processor splits the stream into two: one stream consists of purchases from the cafe, and the other stream contains purchases from the electronics store.

As records from the original stream flow through the branch processor, each record is matched against the supplied predicates in the order that they're provided. The processor assigns records to a stream on the first match; no attempts are made to match additional predicates.

The branch processor drops records if they don't match any of the given predicates. The order of the streams in the returned array matches the order of the predicates provided to the `branch()` method. A separate topic for each department may not be the only approach, but we'll stick with this for now. It satisfies the requirement, and it can be revisited later.

Listing 3.13 Splitting the stream

```
Predicate<String, Purchase> isCoffee =
➥ (key, purchase) ->
➥ purchase.getDepartment().equalsIgnoreCase("coffee");

Predicate<String, Purchase> isElectronics =
➥ (key, purchase) ->
➥ purchase.getDepartment().equalsIgnoreCase("electronics");

int coffee = 0;
int electronics = 1;

KStream<String, Purchase>[] kstreamByDept =
➥ purchaseKStream.branch(isCoffee, isElectronics);

kstreamByDept[coffee].to( "coffee",
     Produced.with(stringSerde, purchaseSerde));
kstreamByDept[electronics].to("electronics",
➥ Produced.with(stringSerde, purchaseSerde));
```

Creates the predicates as Java 8 lambdas

Labels the expected indices of the returned array

Calls branch to split the original stream into two streams

Writes the results of each stream out to a topic

> **WARNING** The example in listing 3.13 sends records to several different topics. Although Kafka can be configured to automatically create topics when it attempts to produce or consume for the first time from nonexistent topics, it's not a good idea to rely on this mechanism. If you rely on autocreating topics, the topics are configured with default values from the server.config properties file, which may or may not be the settings you need. You should always think about what topics you'll need, the level of partitions, and the replication factor ahead of time, and create them before running your Kafka Streams application.

In listing 3.13, you define the predicates ahead of time, because passing four lambda expression parameters would be a little unwieldy. The indices of the returned array are also labeled, to maximize readability.

This example demonstrates the power and flexibility of Kafka Streams. You've been able to take the original stream of purchase transactions and split them into four streams with very few lines of code. Also, you're starting to build up a more complex processing topology, all while reusing the same source processor.

Splitting vs. partitioning streams

Although *splitting* and *partitioning* may seem like similar ideas, they're unrelated in Kafka and Kafka Streams. Splitting a stream with the `KStream.branch` method results in creating one or more streams that could ultimately send records to another topic. Partitioning is how Kafka distributes messages for one topic across servers, and aside from configuration tuning, it's the principal means of achieving high throughput in Kafka.

So far, so good. You've met two of the three new requirements with ease. Now it's time to implement the last additional requirement, generating a key for the purchase record to be stored.

GENERATING A KEY

Kafka messages are in key/value pairs, so all records flowing through a Kafka Streams application are key/value pairs as well. But there's no requirement stating that keys can't be null. In practice, if there's no need for a particular key, having a null key will reduce the overall amount of data that travels the network. All the records flowing into the ZMart Kafka Streams application have null keys.

That's been fine, until you realize that your NoSQL storage solution stores data in key/value format. You need a way to create a key from the `Purchase` data before it gets written out to the `purchases` topic. You certainly could use `KStream.map` to generate a key and return a new key/value pair (where only the key would be new), but there's a more succinct `KStream.selectKey` method that returns a new `KStream` instance that produces records with a new key (possibly a different type) and the same value. This change to the processor topology is similar to filtering, in that you add a processing node between the filter and the sink processor, shown in figure 3.14.

Listing 3.14 Generating a new key

```
KeyValueMapper<String, Purchase, Long> purchaseDateAsKey =
➥ (key, purchase) -> purchase.getPurchaseDate().getTime();

KStream<Long, Purchase> filteredKStream =
➥ purchaseKStream((key, purchase) ->
➥ purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);

filteredKStream.print(Printed.<Long, Purchase>
➥ toSysOut().withLabel("purchases"));
filteredKStream.to("purchases",
➥ Produced.with(Serdes.Long(),purchaseSerde));
```

The KeyValueMapper extracts the purchase date and converts to a long.

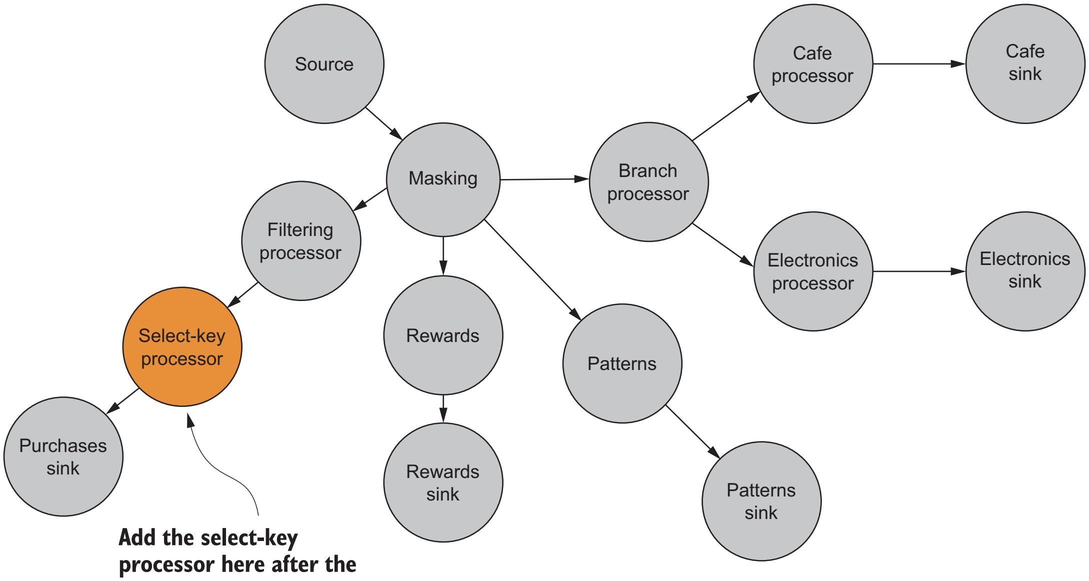
Filters out purchases and selects the key in one statement

Prints the results to the console

Materializes the results to a Kafka topic

To create the new key, you take the purchase date and convert it to a long. Although you could pass a lambda expression, it's assigned to a variable here to help with readability.

Also, note that you need to change the serde type used in the `KStream.to` method, because you've changed the type of the key.



Add the select-key processor here after the filtering, as you only need to generate keys for records that will be written out to the purchases topic.

Figure 3.14 The NoSQL data store will use the purchase date as a key for the data it stores. The new `selectKey` processor will extract the purchase date to be used as a key, right before you write the data to Kafka.

This is a simple example of mapping to a new key. Later, in another example, you'll select keys to enable joining separate streams. Also, all the examples up until this point have been stateless, but there are several options for stateful transformations as well, which you'll see a little later on.

3.5.2 Writing records outside of Kafka

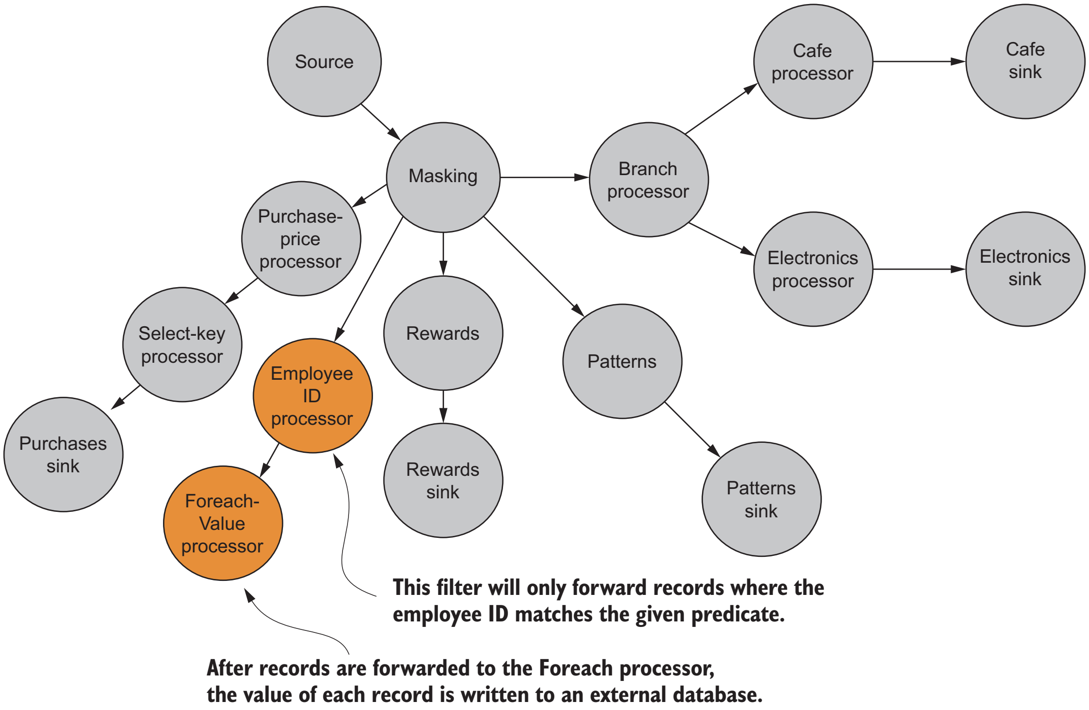
The security department at ZMart has approached you. Apparently, in one of the stores, there's a suspicion of fraud. There have been reports that a store manager is entering invalid discount codes for purchases. Security isn't sure what's going on, but they're asking for your help.

The security folks don't want this information to go into a topic. You talk to them about securing Kafka, about access controls, and about how you can lock down access to a topic, but the security folks are standing firm. These records need to go into a relational database where they have full control. You sense this is a fight you can't win, so you relent and resolve to get this task done as requested.

FOREACH ACTIONS

The first thing you need to do is create a new `KStream` that filters results down to a single employee ID. Even though you have a large amount of data flowing through your topology, this filter will reduce the volume to a tiny amount.

Here, you'll use `KStream` with a predicate that looks to match a specific employee ID. This filter will be completely separate from the previous filter, and it'll be attached to the source `KStream` instance. Although it's entirely possible to chain filters, you won't do that here; you want full access to the data in the stream for this filter.



This filter will only forward records where the employee ID matches the given predicate.

After records are forwarded to the Foreach processor, the value of each record is written to an external database.

Figure 3.15 To write purchases involving a given employee outside of the Kafka Streams application, you'll first add a `filter` processor to extract purchases by employee ID, and then you'll use a `foreach` operator to write each record to an external relational database.

Next, you'll use a `KStream.foreach` method, as shown in figure 3.15. `KStream.foreach` takes a `ForeachAction<K, V>` instance, and it's another example of a terminal node. It's a simple processor that uses the provided `ForeachAction` instance to perform an action on each record it receives.

Listing 3.15 `Foreach` operations

```
ForeachAction<String, Purchase> purchaseForeachAction = (key, purchase) ->
➥ SecurityDBService.saveRecord(purchase.getPurchaseDate(),
➥ purchase.getEmployeeId(), purchase.getItemPurchased());
```

```
purchaseKStream.filter((key, purchase) ->
➥ purchase.getEmployeeId()
➥ .equals("source code has 000000"))
➥ .foreach(purchaseForeachAction);
```

`ForeachAction` uses a Java 8 lambda (again), and it's stored in a variable, `purchaseForeachAction`. This requires an extra line of code, but the clarity gained by doing so more than makes up for it. On the next line, another `KStream` instance sends the filtered results to the `ForeachAction` defined directly above it.

Note that `KStream.foreach` is stateless. If you need state to perform some action for each record, you can use the `KStream.process` method. The `KStream.process` method will be discussed in the next chapter when you add state to a Kafka Streams application.

If you step back and look at what you've accomplished so far, it's pretty impressive, considering the amount of code written. Don't get too comfortable, though, because upper management at ZMart has taken notice of your productivity. More changes and refinements to the purchase-streaming analysis program are coming.

Summary

- You can use the `KStream.mapValues` function to map incoming record values to new values, possibly of a different type. You also learned that these mapping changes shouldn't modify the original objects. Another method, `KStream.map`, performs the same action but can be used to map both the key and the value to something new.
- A predicate is a statement that accepts an object as a parameter and returns `true` or `false` depending on whether that object matches a given condition. You used predicates in the filter function to prevent records that didn't match a given predicate from being forwarded in the topology.
- The `KStream.branch` method uses predicates to split records into new streams when a record matches a given predicate. The processor assigns a record to a stream on the first match and drops unmatched records.
- You can modify an existing key or create a new one using the `KStream.selectKey` method.

In the next chapter, we'll start to look at state, the required properties for using state with a steaming application, and why you might need to add state at all. Then you'll add state to a `KStream` application, first by using stateful versions of `KStream` methods you've seen in this chapter (`KStream.mapValues()`). For a more advanced example, you'll perform joins between two different streams of purchases to help ZMart improve customer service.

SOFTWARE DEVELOPMENT

Kafka Streams IN ACTION

William P. Bejeck Jr.

Not all stream-based applications require a dedicated processing cluster. The lightweight Kafka Streams library provides exactly the power and simplicity you need for message handling in microservices and real-time event processing. With the Kafka Streams API, you filter and transform data streams with just Kafka and your application.

Kafka Streams in Action teaches you to implement stream processing within the Kafka platform. In this easy-to-follow book, you'll explore real-world examples to collect, transform, and aggregate data, work with multiple processors, and handle real-time events. You'll even dive into streaming SQL with KSQL! Practical to the very end, it finishes with testing and operational aspects, such as monitoring and debugging.

What's Inside

- Using the KStream API
- Filtering, transforming, and splitting data
- Working with the Processor API
- Integrating with external systems

Assumes some experience with distributed systems. No knowledge of Kafka or streaming applications required.

Bill Bejeck is a Kafka Streams contributor and Confluent engineer with over 15 years of software development experience.

Free eBook

See first page

"A great way to learn about Kafka Streams and how it is a key enabler of event-driven applications."

—From the Foreword by
Neha Narkhede
Cocreator of Apache Kafka

"A comprehensive guide to Kafka Streams—from introduction to production!"

—Bojan Djurkovic, Cvent

"Bridges the gap between message brokering and real-time streaming analytics."

—Jim Mantheiy Jr.
Next Century

"Valuable both as an introduction to streams as well as an ongoing reference."

—Robin Coe, TD Bank

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/kafka-streams-in-action

ISBN-13: 978-1-61729-447-1
ISBN-10: 1-61729-447-0
54499
9 781617 294471

MANNING

$44.99 / Can $59.99 [INCLUDING eBOOK]