# 6 Developing Kafka Streams

**This chapter covers**

- Introducing the Kafka Streams API
- Building our first Kafka Streams application
- Working with customer data; creating more complex applications
- Splitting, merging and branching streams oh my!

Simply stated, a Kafka Streams application is a graph of processing nodes that transforms event data as it streams through each node. Let's take a look at an illustration of what this means:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

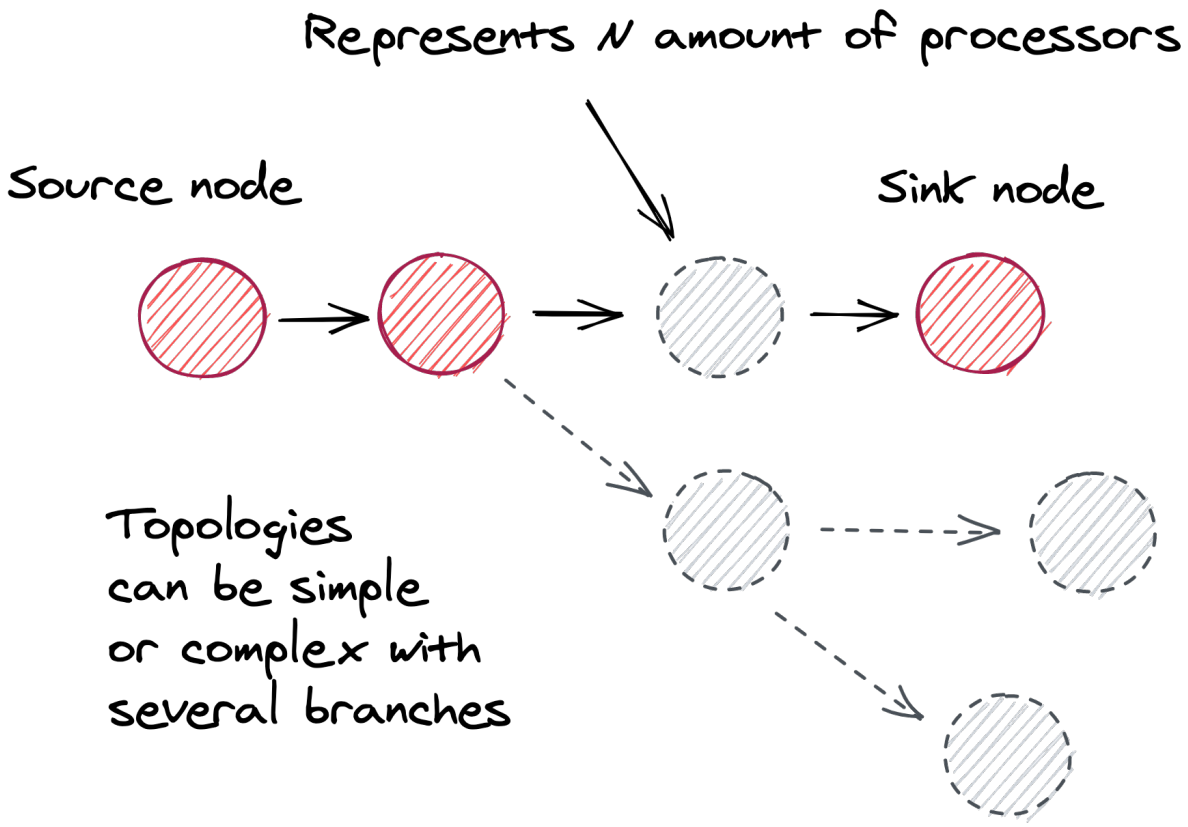


**Figure 6.1 Kafka Streams is a graph with a source node, any number of processing nodes and a sink node**

This illustration represents the generic structure of most Kafka Streams applications. There is a source node that consumes event records from a Kafka broker. Then there are any number of processing nodes, each performing a distinct task and finally a sink node used to write the transformed records back out to Kafka. In a previous chapter we discussed how to use the Kafka clients for producing and consuming records with Kafka. Much of what you learned in that chapter applies for Kafka Streams, because at it's heart, Kafka Streams is an abstraction over the producers and consumers, leaving you free to focus on your stream processing requirements.

| IMPORTANT | **While Kafka Streams is the native stream processing library for Apache Kafka ®, it does not run inside the cluster or brokers, but connects as a client application.** |
|---|---|

In this chapter, you'll learn how to build such a graph that makes up a stream processing application with Kafka Streams.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

## 6.1 The Streams DSL

The Kafka Streams DSL is the high-level API that enables you to build Kafka Streams applications quickly. This API is very well thought out, with 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 lambda expressions. Taking these factors into account, you can imagine the simplicity and ease with which you can build a Kafka Streams program.

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 a later chapter. With that introduction out of the way, let's dive into the requisite Hello World program for Kafka Streams.

## 6.2 Hello World for Kafka Streams

For the first Kafka Streams example, we'll build something fun that will get off the ground quickly so you can see how Kafka Streams works; a toy application that takes incoming messages and converts them to uppercase characters, effectively yelling at anyone who reads the message. We'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:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

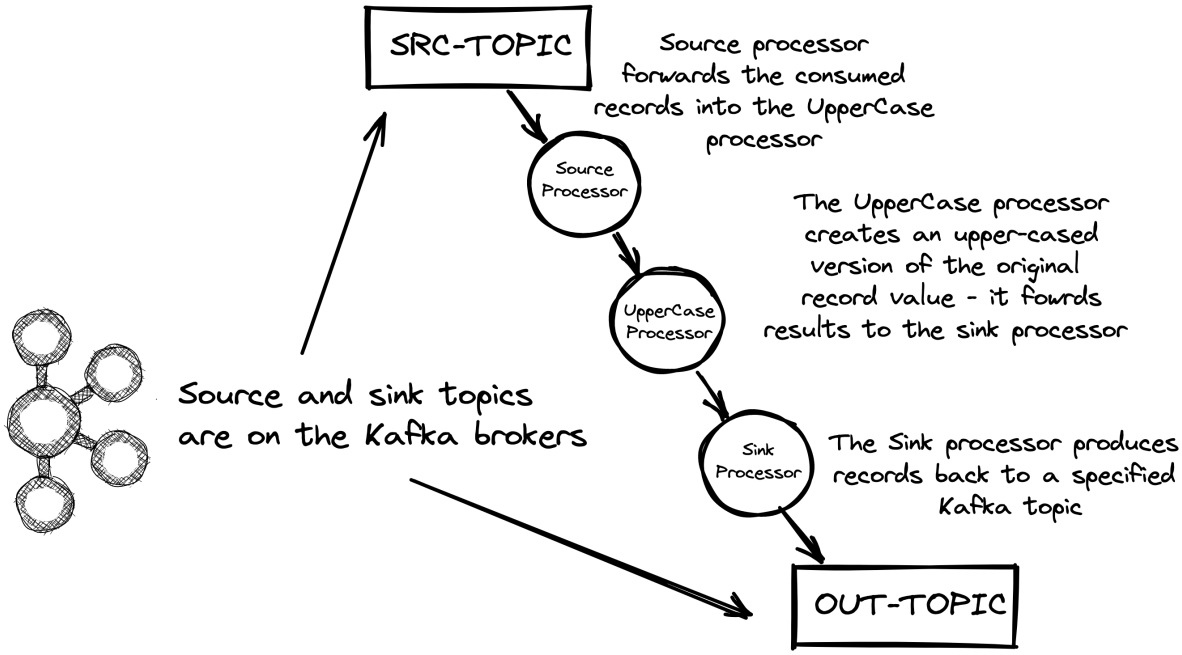


Figure 6.2 Topology of the Yelling App

As you can see, it's 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.

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 pattern:

1. Define the configuration items.
2. Create `Serde` instances, either custom or predefined, used in deserialization/serializtion of records.
3. Build the processor topology.
4. Create and start the `Kafka Streams`.

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

### 6.2.1 Creating the topology for the Yelling App

The first step to creating any Kafka Streams application is to create a source node and that's exactly what you're going to do here. The source node is the root of the topology and fowards thge consumed records into application. Figure 6.3 highlights the source node in the graph.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

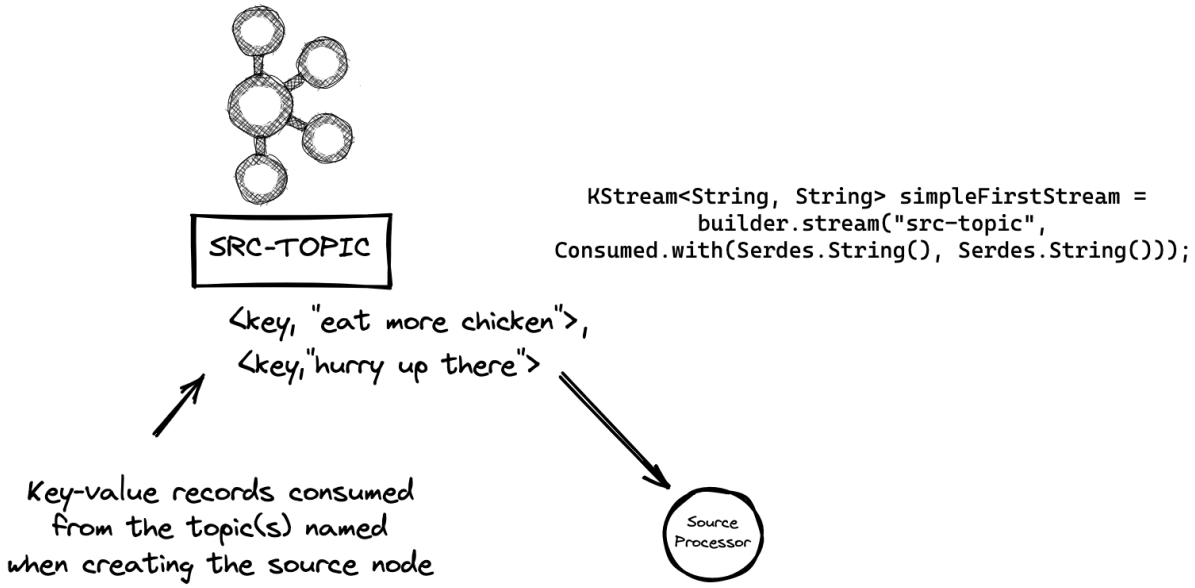


Figure 6.3 Creating the source node of the Yelling App

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

**Listing 6.1 Defining the source for the stream**

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

The `simpleFirstStream` instance is set to consume messages from the `src-topic` topic. In addition to specifying the topic name, you can add a `Consumed` object that Kafka Streams uses to configure optional parameters for a source node. In this example you've provided `Serde` instances, the first for the key and the second one for the value. A `Serde` is a wrapper object that contains a serializer and deserializer for a given type.

If you remember from our dicussion on consumer clients in a previous chapter, the broker stores and forwards records in byte array format. For Kafka Streams to perform any work, it needs to deserialize the bytes into concrete objects. Here both `Serde` objects are for strings, since that's the type of both the key and the value. Kafka Streams will use the `Serde` to deserialize the key and value, separately, into string objects. We'll explain Serdes in more detail soon. You can also use the `Consumed` class to configure a `TimestampExtractor`, the offset reset for the source node, and provide a name. We'll cover the `TimestampExtractor` and providing names in later sections and since we covered offset resets in a previous chapter, I won't cover them again here.

And that is how to create a `KStream` to read from a Kafka topic. But a single topic is not our only choice. Let's take a quick look at some other options. Let's say that there are several topics you'd like to like to "yell at". In that case you can subscribe to all of them at one time by using a `Collection<String>` to specify all the topic names as shown here:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**Listing 6.2 Creating the Yelling Application with multiple topics as the source**

```
KStream<String, String> simpleFirstStream =
  builder.stream(List.of("topicA", "topicB", "topicC"),
        Consumed.with(Serdes.String(), Serdes.String()))
```

Typically you'd use this approach when you want to apply the same processing to multiple topics at the same time. But what if you have long list of similarly named topics, do you have to write them all out? The answer is no! You can use a regular expresion to subscribe to any topic that matches the pattern:

**Listing 6.3 Using a regular expression to subscribe to topics in the Yelling Application**

```
KStream<String, String> simpleFirstStream =
   buider.source(Pattern.compile("topic[A-C]"),
      Consumed.with(Serdes.String(), Serdes.String()))
```

Using a regular expression for subscribing to topics is particulary handy when your organization uses a common naming pattern for topics related to their business function. You just have to know the naming pattern and you can subscribe to all of them concisely. Additionally as topics are created or deleted your subscription will automatically update to reflect the changes in the topics.

When subscribing to mulitple topics, there are a few caveats to keep in mind. The keys and values from all subscribed topics must be the same type, for example you can't combine topics where one topic contains `Integer` keys and another has `String` keys. Also, if they all aren't partitioned the same, it's up to you to repartition the data before performing any key based operation like aggregations. We'll cover repartitioning in the next chapter. Finally, there's no ordering guarantees of the incoming records.

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 6.4.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

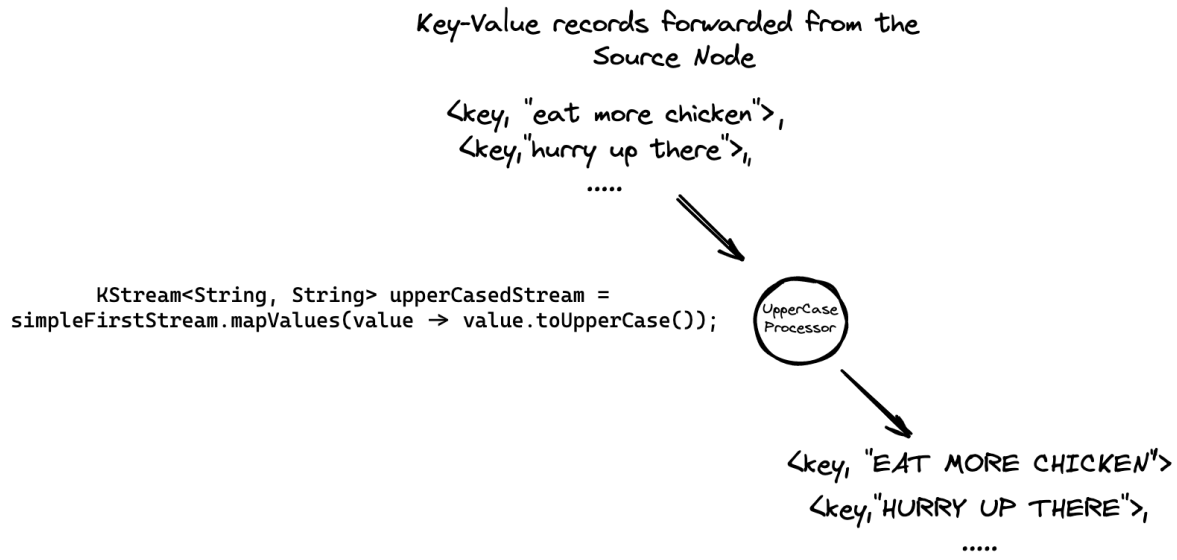


Figure 6.4 Adding the uppercase processor to the Yelling App

**Listing 6.4 Mapping incoming text to uppercase**

```
KStream<String, String> upperCasedStream =
  simpleFirstStream.mapValues(value -> value.toUpperCase());
```

In the introduction to this chapter I mentioned that a Kafka Streams application is a graph of processing nodes, a directed acyclic graph or DAG to be precise.

You build the graph one processor at a time. With each method call, you establish a parent-child relationship between the nodes of the graph. The parent-child relationship in Kafka Streams establishes the direction for the flow of data, parent nodes forward records to their children. A parent node can have multiple children, but a child node will only have one parent.

So looking at the code example here, by executing `simpleFirstStream.mapValues` , you're creating a new processing node whose inputs are the records consumed in the source node. So the source node is the "parent" and it forwards records to its "child", the processing node returned from the `mapValues` operation.

**NOTE** **As you tell from the name `mapValues` only affects the value of the key-value pair, but the key of the original record is still forwared along.**

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 lambda expression, which is exactly what you've done here with `value value.toUpperCase()`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

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

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 6.5 shows where you are in the construction of the topology.

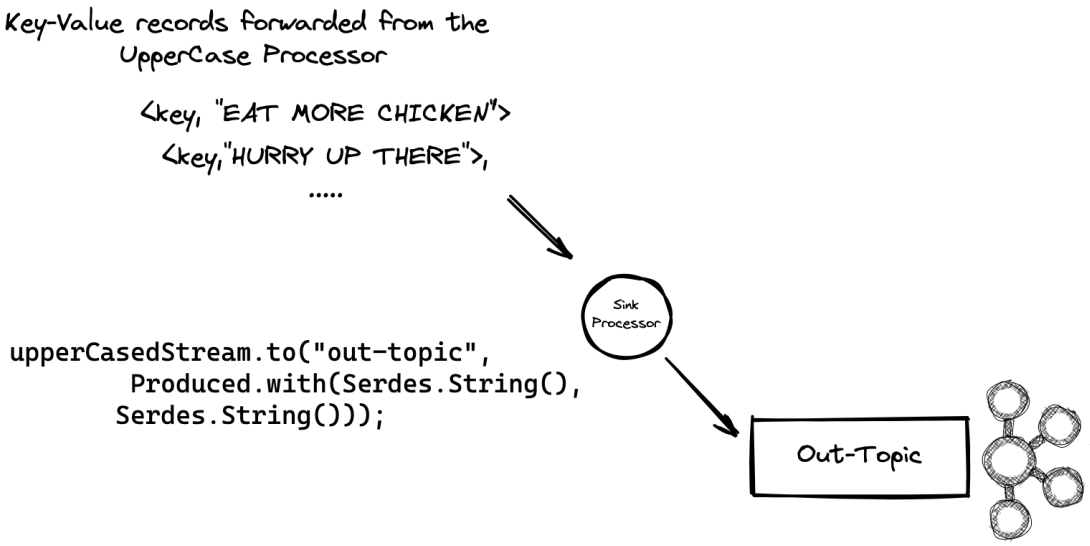


**Figure 6.5 Adding a processor for writing the Yelling App results**

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

**Listing 6.5 Creating a sink node**

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

The `KStream.to` method creates a processing node that writes the final transformed records to a Kafka topic. It is a child of the `upperCasedStream`, so it receives all of its inputs directly from the results of the `mapValues` operation.

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.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**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(Serdes.String(), Serdes.String()));

KStream<String, String> upperCasedStream =
simpleFirstStream.mapValues(value -> value.toUpperCase());
upperCasedStream.to("out-topic", Produced.with(Serdes.String(), Serdes.String()));
```

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. A fluent interface (martinfowler.com/bliki/FluentInterface.html) is an approach where you chain method calls together for more concise and readable code. To demonstrate this idea, here's another way you could construct the Yelling App topology:

```
builder.stream("src-topic", Consumed.with(Serdes.String(), Serdes.String()))
 .mapValues(value -> value.toUpperCase())
 .to("out-topic", Produced.with(Serdes.String(), Serdes.String()));
```

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.

### 6.2.2 Kafka Streams configuration

Although Kafka Streams is highly configurable, with several properties you can adjust for your specific needs, the uses the two required 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 are required because there's no practical way to provide default values for these configurations. Attempting to start a Kafka Streams program without these two properties defined will result in a `ConfigException` being thrown.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

The `StreamsConfig.APPLICATION_ID_CONFIG` property uniquely identifies your Kafka Streams application. Kafka Streams instances with the same application-id are considered one logical application. We'll discuss this concept later in Kafka Streams internals section. The application-id also serves as a prefix for the embedded client (`KafkaConsumer` and `KafkaProducer`) configurations. You can choose to provide custom configurations for the embedded clients by using one of the various prefix labels found in the `StreamsConfig` class. However, the default client configurations in Kafka Streams have been chosen to provide the best performance, so one should exercise caution when adjusting them.

The `StreamsConfig.BOOTSTRAP_SERVERS_CONFIG` property can be a single `hostname:port` pair or multiple `hostname:port` comma-separated pairs. The `BOOTSTRAP_SERVERS_CONFIG` is what Kafka Streams uses to establish a connection to the Kafka cluster. We'll cover several more configuration items as we explore more examples in the book.

### 6.2.3 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, Bytes, Long, Short, Integer, Double, Float, ByteBuffer, UUID, and Void.

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 upcoming examples, you'll use Serdes for working with Avro, Protobuf, and JSONSchema as well as 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_6/KafkaStreamsYellingApp.java (source code can be found on the book's website here: www.manning.com/books/kafka-streams-in-action-second-edition).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

### Listing 6.6 Hello World: the Yelling App

```
//Details left out for clarity
public class KafkaStreamsYellingApp extends BaseStreamsApplication {

private static final Logger LOG =
  LoggerFactory.getLogger(KafkaStreamsYellingApp.class);

@Override
public Topology topology(Properties streamProperties) {

  Serde<String> stringSerde = Serdes.String();   ❶
  StreamsBuilder builder = new StreamsBuilder();   ❷

  KStream<String, String> simpleFirstStream = builder.stream("src-topic",
            Consumed.with(stringSerde, stringSerde));   ❸
  KStream<String, String> upperCasedStream =
  simpleFirstStream.mapValues(value)-> value.toUpperCase());   ❹

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

  return builder.build(streamProperties);
}

public static void main(String[] args) throws Exception {
    Properties streamProperties = new Properties();
    streamProperties.put(StreamsConfig.APPLICATION_ID_CONFIG,
      "yelling_app_id");
    streamProperties.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
      "localhost:9092");
    KafkaStreamsYellingApp yellingApp = new KafkaStreamsYellingApp();
    Topology topology = yellingApp.topology(streamProperties);

    try(KafkaStreams kafkaStreams =
                    new KafkaStreams(topology, streamProperties)) {
        LOG.info("Hello World Yelling App Started");
        kafkaStreams.start();   ❻

        LOG.info("Shutting down the Yelling APP now");
    }
 }
}
```

❶ Creates the Serdes and store in a variable used to serialize/deserialize keys and values

❷ Creates the StreamsBuilder instance used to construct the processor topology

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

❹ A processor using a lambda (the first child node in the graph)

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

❻ 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 `Properties` instance for configurations.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

2. Create a `Serde` object.
3. Construct a processing topology.
4. Start the Kafka Streams program.

We'll now move on to a more complex example that will allow us to explore more of the Streams DSL API.

## 6.3 Masking credit card numbers and tracking purchase rewards in a retail sales setting

Imagine you work as a infrastructure engineer for the retail giant, ZMart. ZMart has adopted Kafka as its data processing backbone and is looking to capitalize on the ability to quickly process customer data, intended to help ZMart do business more efficiently.

At this point you're tasked to build a Kafka Streams application to work with purchase records as they come streaming in from transactions in ZMart stores.

Here are the requirements for the streaming program, which will also serve as a good description of what the program will do:

1. All Purchase objects need to have credit card numbers protected, in this case by masking the first 12 digits.
2. You need to extract the items purchased and the ZIP code to determine regional purchase patterns and inventory control. This data will be written out to a topic.
3. 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.

With these requirements at hand, let's get started building a streaming application that will satisfy ZMart's business requirements.

### 6.3.1 Building the source node and the masking processor

The first step in building the new application is to create the source node and first processor of the topology. You'll do this by chaining two calls to the `KStream` API together. The child processor of the source node will mask credit card numbers to protect customer privacy.

**Listing 6.7 Building the source node and first processor**

```
KStream<String, RetailPurchase> retailPurchaseKStream =
     streamsBuilder.stream("transactions",
     Consumed.with(stringSerde, retailPurchaseSerde))
    .mapValues(creditCardMapper);
```

You create the source node with a call to the `StreamBuilder.stream` method using a default `String` serde, a custom serde for `RetailPurchase` objects, and the name of the topic that's the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

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 code example, 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 `RetailPurchase` 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 (`RetailPurchase`), but with a masked credit card number.

When using the `KStream.mapValues` method, you don't have access to the key for the value computation. If you wanted to use the key to compute the new value, you could use the `ValueMapperWithKey<K, V, VR>` interface, with the expectation that the key remains the same. If you need to generate a new key along with the value, you'd use the `KStream.map` method that takes a `KeyValueMapper<K, V, KeyValue<K1, V1>>` interface.

**IMPORTANT** **Keep in mind that Kafka Streams functions are expected to operate without side effects, meaning the functions don't modify the original key and or value, but return new objects when making modifications.**

### 6.3.2 Adding the patterns processor

Now you'll build the second processor, responsible for extracting geographical data from the purchase, which ZMart can use to determine purchase patterns and inventory control in regions of the country. There's also an additional wrinkle with building this part of the topology. The ZMart business analysts have determined they want to see individual records for each item in a purchase and they want to consider purchases made regionally together.

The `RetailPurchase` data model object contains all the items in a customer purchase so you'll need to emit a new record for each one in the transaction. Additionally, you'll need to add the zip-code in the transaction as the key. Finally you'll add a sink node responsible for writing the pattern data to a Kafka topic.

In pattrens processor example you can see the `retailPurchaseKStream` processor using a `flatMap` operator. The `KStream.flatMap` method takes a `ValueMapper` or a `KeyValueMapper` that accepts a single record and returns an `Iterable` (any Java `Collection`) of new records, possibly of a different type. The `flapMap` processor "flattens" the `Iterable` into one or more records forwarded to the topology. Let's take a look at an illustrating how this works:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

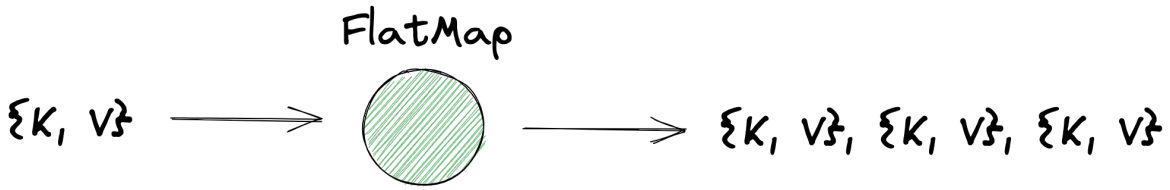


**Figure 6.6 FlatMap emits zero or more records from a single input records by flattening a collection returned from a KeyValueMapper or ValueMapper**

The process of a flatMap is a common operation from functional programming where one input results creating a collection of items (the map portion of the function) but instead of returning the collection, it "flattens" the collection or grouping into a sequence of records.

In our case here with Kafka Streams, a retail purchase of five items results in five individual `KeyValue` objects with the keys corresponding to the zip-code and the values a `PurchasedItem` object.

Here's the code listing for the `KeyValueMapper`:

**Listing 6.8 KeyValueMapper returning a collection of PurchasedItem objects**

```
KeyValueMapper<String, RetailPurchase,
 Iterable<KeyValue<String, PurchasedItem>>> retailTransactionToPurchases =
    (key, value) -> {
      String zipcode = value.getZipCode();   ❶
      return value.getPurchasedItemsList().stream()   ❷
                .map(purchasedItem ->
                    KeyValue.pair(zipcode, purchasedItem))
                .collect(Collectors.toList());
}
```

❶ Extracting the zipcode on the purchase for the new key

❷ Using the Java stream API to create a list of KeyValue pairs

The `KeyValueMapper` here takes an individual transaction object and returns a list of KeyValue objects. The key is the zipcode where the transaction took place and the value is an item included in the purchase. Now let's put our new `KeyValueMapper` into this section of the topology we're creating:

**Listing 6.9 Patterns processor and a sink node that writes to Kafka**

```
KStream<String, Pattern> patternKStream = retailPurchaseKStream
                .flatMap(retailTransactionToPurchases)   ❶
                .mapValues(patternObjectMapper);   ❷

patternKStream.print(Printed.<String,Pattern>toSysOut()
              .withLabel("patterns"));   ❸
patternKStream.to("patterns",
        Produced.with(stringSerde,purchasePatternSerde));   ❹
```

❶ Using flatMap to create new object for each time in a transaction

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

❷ Mapping each purchase to a pattern object

❸ Printing records to the console

❹ Producing each record from the purchase to a Kafka topic called "patterns"

In this code example you declare a variable to hold the reference of the new `KStream` instance and you'll see why in an upcoming section. The purchase-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`. I've also snuck in a `KStream#print` processor that prints the key-values of the stream to the console, we'll talk more about viewing stream records in an upcoming section.

The `KStream.to` method is a mirror image of the `KStream.source` method. Instead of setting a source for the topology to read from, it 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 overloads which accept an object allowing for dynamic topic selection and we'll discuss that soon.

### 6.3.3 Building the rewards processor

The third processor in the topology is the customer rewards accumulator node shown in figure 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.

**Listing 6.10 Third processor and a terminal node that writes to Kafka**

```
 KStream<String, RewardAccumulatorProto.RewardAccumulator> rewardsKStream =
       retailPurchaseKStream.mapValues(rewardObjectMapper);
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 retail purchase object 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.

Now that you've built the application piece by piece, let's look at the entire application (src/main/java/bbejeck/chapter_6/ZMartKafkaStreamsApp.java).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**Listing 6.11 ZMart customer purchase `KStream` program**

```
public class ZMartKafkaStreamsApp {

// Details left out for clarity


@Override
public Topology topology(Properties streamProperties) {

StreamsBuilder streamsBuilder = new StreamsBuilder();

KStream<String, RetailPurchaseProto.RetailPurchase> retailPurchaseKStream =
        streamsBuilder.stream("transactions",
            Consumed.with(stringSerde, retailPurchaseSerde))
        .mapValues(creditCardMapper);    ❶

KStream<String, PatternProto.Pattern> patternKStream =
   retailPurchaseKStream
        .flatMap(retailTransactionToPurchases)
        .mapValues(patternObjectMapper);    ❷

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

KStream<String, RewardAccumulatorProto.RewardAccumulator> rewardsKStream =
      retailPurchaseKStream.mapValues(rewardObjectMapper);    ❸

rewardsKStream.to("rewards",
      Produced.with(stringSerde,rewardAccumulatorSerde));
retailPurchaseKStream.to("purchases",
      Produced.with(stringSerde,retailPurchaseSerde));

return streamsBuilder.build(streamProperties);

}
```

❶ Builds the source and first processor

❷ Builds the PurchasePattern processor

❸ Builds the RewardAccumulator processor

**NOTE** **I've left out some details in the listing 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 `StreamsBuilder` instance.
- Build one or more `Serde` instances.
- Construct the processing topology.
- Assemble all the components and start the Kafka Streams program.

You'll also notice that I haven't shown the logic responsible for creating the various mappings from the original transaction object to new types and that is by design. First of all, the code for a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

`KeyValueMapper` or `ValueMapper` is going to be distinct for each use case, so the particular implementations don't matter too much.

But more to the point, if you look over the entire Kafka Streams application you can quickly get a sense of what each part is accomplishing, and for the most part any details of working directly with Kafka are abstracted away. And to me that is the strength of Kafka Streams; with the DSL you get specify ***what*** operations you need to perform on the event stream and Kafka Streams handles the details. Now it's true that no one framework can solve every problem and sometimes you need a more hands-on lower level approach and you'll learn about that in a upcoming chapter when we cover the Processor API.

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.

### 6.3.4 Using Serdes to encpsulate serializers and deserializers in Kafka Streams

As you learned in previous chapters, Kafka brokers work with records in byte array format. It's the responsibility of the client to serialize when producing records and deserialize when consuming. It's no different with Kafka Streams as it uses embedded consumers and producers. There is one small difference when configuring a Kafka Streams application for serialization vs. raw producer or consumer clients. Instead of providing a specific deserializer or serializer, you configure Kafka Streams with a `Serde`, which contains both the serializer and deserializer for a specific type.

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 of the arbitrary object types. We'll look at what's involved in building a `Serde` for the `RetailPurchase` class. We won't cover the other `Serde` instances, because they follow the same pattern, just with different types.

> **NOTE** **I'm including this discussion on Serdes creation for completeness, but in the source code there is a class `SerdeUtil` which provides a `protobufSerde` method which you'll see in the examples and encapsulates the steps described in this section.**

Building a `Serde` requires implementations of the `Deserializer<T>` and `Serializer<T>` interfaces. We covered creating your own serializer and deserializer instances towards the end of

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

chapter 3 on Schema Registry, so I won't go over those details again here. For reference you can see the full code for the `ProtoSerializer` and `ProtoDeserializer` in the `bbejeck.serializers` package in the source for the book.

Now, to create a `Serde<T>` object, you'll use the `Serdes.serdeFrom` factory method taking steps like the following:

```
Deserializer<RetailPurchaseProto.RetailPurchase> purchaseDeserializer =
      new ProtoDeserializer<>();   ❶
Serializer<RetailPurchaseProto.RetailPurchase> purchaseSerializer =
      new ProtoDeserializer<>();   ❷
Map<String, Class<RetailPurchaseProto.RetailPurchase>> configs
            = new HashMap<>();
   configs.put(false, RetailPurchaseProto.RetailPurchase.class);
         deserializer.configure(configs,isKey);    ❸
Serde<RetailPurchaseProto.RetailPurchase> purchaseSerde =
     Serdes.serdeFrom(purchaseSerializer,purchaseDeserializer);   ❹
```

❶ Creates the Deserializer for the RetailPurchaseProto.RetailPurchase class

❷ Creates the Serializer for the RetailPurchaseProto.RetailPurchase class

❸ Configurations for the deserializer

❹ Creates the Protobuf Serde for RetailPurchaseProto.RetailPurchase 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. Here you need to create a custom Serde for the Protobuf objects because the streams example does not use Schema Registry, but using it with Kafka Streams is a perfectly valid use case. Let's take a quick pause to go over how you configure your Kafka Streams application when using it with Schema Registry.

### 6.3.5 Kafka Streams and Schema Registry

In chapter four I discussed the reasons why you'd want to use Schema Registry with a Kafka based application. I'll briefly describe those reasons here. The domain objects in your application represent an implicit contract between the different users of your application. For example imagine one team of developers change a field type from a `java.util.Date` to a `long` and start producing those changes to Kafka, the downstream consumers applications will break due to the unexpected field type change.

So by using a schema and using Schema Resitry to store it, you make it much easier to enforce this contract by enabling better coordination and compatibility checks. Additionally, there are Schema Registry project provides Schema Registry "aware" (de)serializers and Serdes, alleviating the developer from writing the serialization code.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**IMPORTANT** **Schema Registry provides both a `JSONSerde` and a `JSONSchemaSerde`, but they are not interchangeable! The `JSONSerde` is for Java objects that use JSON for describing the object. The `JSONSchemaSerde` is for objects that use `JSONSchema` as the formal definition of the object.**

So how would the `ZMartKafkaStreamsApp` change to work with Schema Registry? All that is required is to use Schema Registry aware Serde instances. The steps for creating a Schema Regeistry aware Serde are simple:

1. Create an instance of one the provided Serde instances
2. Configure it with the URL for a Schema Registry server.

Here are the concrete steps you'll take:

```
KafkaProtobufSerdePurchase> protobufSerde =
 new KafkaProtobufSerde<>(Purchase.class);   ❶
String url = "https://...";                                  ❷
Map<String, Object> configMap = new HashMap<>();
configMap.put(
  AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
  url);                                                      ❸
protobufSerde.configure(configMap, false);     ❹
```

❶ Instantiating the KafkaProtobufSerde providing the class type as a constructor parameter

❷ The URL for the location of a Schema Registry instance

❸ Putting the URL in a HashMap

❹ Calling the KafkaProtobufSerde#configure method

So, with just few lines of code, you've created a Schema Registry aware Serde that you can use in your Kafka Streams application.

**IMPORTANT** **Since Kafka Streams contains consumer and producer clients, the same rules for schema evolution and compatibility apply**

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.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

## 6.4 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 instead of using an external tool, it would be more convenient to have your Kafka Streams application print or log from anywhere you want inside the topology. This visual type of feedback directly from the application is very efficient during the development process. You enable this output by using the `KStream.peek()` or the `KStream.print()` method.

The `KStream.peek()` allows you to perform a stateless action (via the `ForeachAction` interface) on each record flowing throw the `KStream` instance. It's important to note that this operation is not expected to alter the incoming key and value. Instead the `peek` operator is an opportunity to print, log, or collect information at arbitrary points in the topology. Let's take a another look at Yelling application, but now add a way to view the records before and after the application starts "yelling":

**Listing 6.12 Printing records flowing through the Yelling application found in bbejeck/chapter_6/KafkaStreamsYellingAppWithPeek**

```
// Details left out for clarity

ForeachAction<String, String> sysout =
  (key, value) ->
   System.out.println("key " + key
   + " value " + value);


builder.stream("src-topic",
  Consumed.with(stringSerde, stringSerde))
  .peek(sysout)        ❶
  .mapValues(value -> value.toUpperCase())
  .peek(sysout)        ❷
  .to( "out-topic",
  Produced.with(stringSerde, stringSerde));
```

❶ Printing records to the console as they enter the application

❷ Printing the yelling events

Here we've strategically placed these `peek` operations that will print records to the console, both pre and post the `mapValues` call.

The `KStream.print()` method is purpose built for printing records. Some of the previous code snippets contained examples of using it, but we'll show it again here.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**Listing 6.13 Printing records using KStream.print found in bbejeck/chapter_6/KafkaStreamsYellingApp**

```
// Details left out for clarity
 ...
 KStream<...> upperCasedStream = simpleFirstStream.mapValues(...);
 upperCasedStream.print(Printed.toSysOut());   (1)
 upperCasedStream.to(...);
```

(1) Printing the upper cased lettters this is an example of a terminal method in Kafka Streams

In this case, you're printing the upper-cased words immediately after transformation. So what's the difference between the two approaches? You should notice with the `KStream.print()` operation, you didn't chain the method calls together like you did using `KStream.peek()` and this is becuase `print` is a terminal method.

Terminal methods in Kafka Streams have a return signature of `void`, hence you can't chain another method call afterward, as it terminates the stream. The terminal methods in `KStream` interface are `print`, `foreach`, `process`, and `to`. Asside from the `print` method we just discussed, you'll use `to` when you write results back to Kafka. The `foreach` method is useful for performing an operation on each record when you don't need to write the results back to Kafka, such as calling a microservice. The `process` method allows for integrating the DSL with Processor API which we'll discuss in an upcoming chapter.

While either printing method is a valid approach, my preference is to use the `peek` method because it makes it easy to slip a print statement into an existing stream. But this is a personal preference so ultimately it's up to you to decide which approach to use.

So far we've covered some of the basic things we can do with a Kafka Streams application, but we've only scratched the surface. Let's continue exploring what we can do with an event stream.

## 6.5 Choosing which events to process

So far you've seen how to apply operations to events flowing through the Kafka Streams application. But you are processing every event in the stream and in the same manner. What if there are events you don't want to handle? Or what about events with a given attribute that require you to handle them differently?

Fortunately, there are methods available to provide you the flexibility to meet those needs. The `KStream#filter` method drops records from the stream not matching a given predicate. The `KStream#split` allows you split the original stream into branches for different processing based on provided predicate(s) to reroute records. To make these new methods more concrete let's update the requirements to the original ZMart application:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

- The ZMart updated their rewards program and now only provides points for purchases over $10. With this change it would be ideal to simply drop any non-qualifying purchases from the rewards stream.
- ZMart has expanded and has bought an electronics chain and a popular coffee house chain. All purchases from these new stores will flow into the streaming application you've set up, but you'll need to separate those purchases out for different treatment while still processing everything else in the application the same.

**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_6/ZMartKafkaStreamsFilteringBranchingApp.java.

### 6.5.1 Filtering purchases

The first update is to remove non-qualifying purchases from the rewards stream. To accomplish this, you'll insert a `KStream.filter()` before the `KStream.mapValues` method. The `filter` takes a `Predicate` interface as a parameter, and it 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.

**NOTE** There is also `KStream.filterNot`, which performs filtering, but in reverse. Only records that don't match the given predicate are processed further in the topology.

By making these changes, the processor topology graph changes as shown in figure 6.12.

**Listing 6.14 Adding a filter to drop purchases not meeting rewards criteria**

```
KStream<String, RewardAccumulatorProto.RewardAccumulator> rewardsKStream =
   retailPurchaseKStream   ❶
   .mapValues(rewardObjectMapper)   ❷
   .filter((key,potentialReward) ->
           potentialReward.getPurchaseTotal() > 10.00);   ❸
```

❶ The original rewards stream

❷ Mapping the purchase into a `RewardAccumulator` object

❸ The `KStream.filter` method, which takes a `Predicate<K,V>` instance as a parameter

You have now successfully updated the rewards stream to drop purchases that don't qualify for reward points.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

### 6.5.2 Splitting/branching the stream

There are new events flowing into the purchase stream and you need to process them differently. You'll still want to mask any credit card information, but after that the purchases from the acquired coffee and electronics chain need to get pulled out and sent to different topics. Additionally, you need to continue to process the original events in the same manner.

What you need to do is split the original stream into 3 sub-streams or branches; 2 for handling the new events and 1 to continue processing the original events in the topology you've already built. This splitting of streams sounds tricky, but Kafka Streams provides an elegant way to do this as we'll see now. Here's an illustration demonstrating the conceptual idea of what splitting a stream involves:

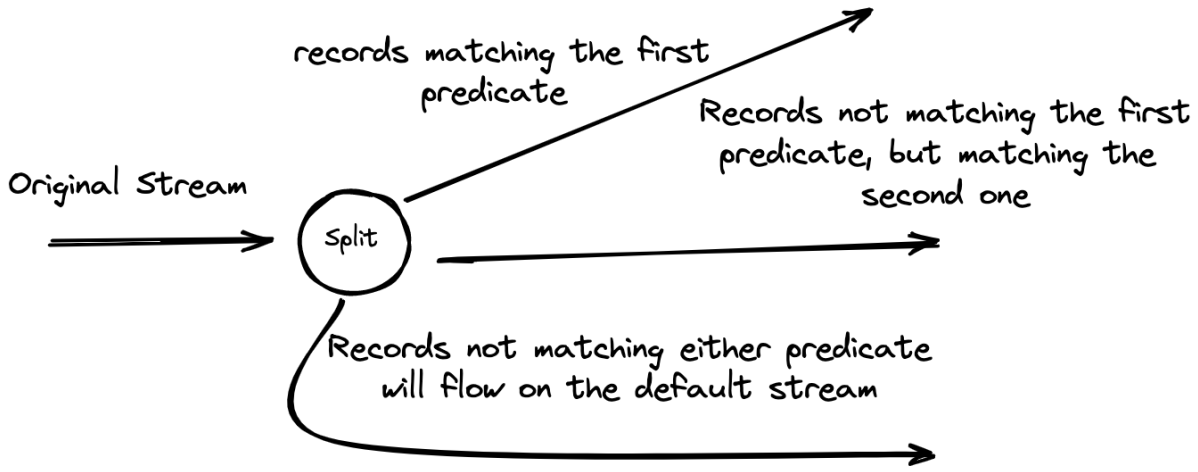


Figure 6.7 Creating branches for the two specific purchase types

The general steps you'll take to split a stream into branches are the following:

1. Use the `KStream.split()` method which returns a `BranchedKStream` object
2. Call `BranchedKStream.branch()` with a with a pair of `Predicate` and `Branched` objects as parameters. The `Predicate` contains a condition when tested against a record returns either true or false. The `Branched` object contains the logic for processing a record. Each execution of this method creates a new branch in the stream.
3. You complete the branching with a call to either `BranchedKStream.defaultBranch()` or `BranchedKStream.noDefaultBranch()`. If you define a default branch any records not matching all the predicates are routed there. With the `noDefaultBranch` option, non-matching records get dropped. When calling either of the branching termination methods a `Map<String, KStream<K, V>` is returned. The `Map` ***may*** contain `KStream` objects for new branch, depending on how you've built the `Branched` objects. We'll cover more options for branching soon.

The `Predicate` acts as a logical gate for it's companion `Branched` object. If the condition returns `true`, then the "gate" opens and the record flows into the processor logic for that branch.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**IMPORTANT** **When splitting a `KStream` you can't change the types of the keys or values, as each branch has the same types as the parent or original branch.**

In our case here, you'll want to filter out the two purchase types into their own branch. Then create a default branch consisting of everything else. This default branch is really the original purchase stream so it will handle all of the records that don't match either predicate.

Now that we've reviewed the concept let's take a look at the code you'll implement:

**Listing 6.15 Splitting the stream found in bbejeck/chapter_6/ZMartKafkaStreamsFilteringBranchingApp**

```
//Several details left out for clarity

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

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

purchaseKStream.split())    ❷
.branch(isCoffee,
 Branched.withConsumer(coffeeStream -> coffeeStream.to("coffee-topic")))   ❸
.branch(isElectronics,
  Branched.withConsumer(electronicStream ->
  electronicStream.to("electronics"))    ❹
.defaultBranch(Branched.withConsumer(retailStream ->
                retailStream.to("purchases"));   ❺
```

❶ Create the predicates for determining branches

❷ Splitting the stream

❸ Writing the coffee purchases out to a topic

❹ Writing the electronic purchases out to a topic

❺ The default branch where non-matching records go

Here in this example you've split the purchase stream into two new streams, one each for the coffee and electronic purchases. Branching provides an elegant way to process records differently within the same stream. While in this initial example each one is a single processor writing records to a topic, these branched streams can be as complex as you need to make them.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

> **NOTE** This example sends records to several different topics. Although you can configure Kafka to automatically create topics it's not a good idea to rely on this mechanism. If you use auto-creation, 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 number of partitions, the replication factor and create them before running your Kafka Streams application.

In this branching example, you've split out discrete `KStream` objects, which stand alone and don't interact with anything else in the application and that is perfectly an acceptable approach. But now let's consider a situation where you have an event stream you want to tease out into separate components, but you need to combine the new streams with existing ones in the application.

Consider you have IoT sensors and early on you combined two related sensor readings into one topic, but as time went on newer sensors started to send results to distinct topics. The older sensors are fine as is and it would be cost prohibited to go back and make the necessary changes to fit the new infrastructure. So you'll need an application that will split the legacy stream into two streams ***and*** combine or merge them with the newer streams consisting of a single reading type. Another factor is that any proximity readings are reported in feet, but the new ones are in meters, so in addition to extracting the proximity reading into a separate stream, you need to convert the reading values into meters.

Now let's walk through an example of how you'll do splitting and merging starting with the splitting

**Listing 6.16 Splitting the stream in a way you have access to new streams**

```
//Details left out for clarity

KStream<String, SensorProto.Sensor> legacySensorStream =
    builder.stream("combined-sensors", sensorConsumed);


 Map<String, KStream<String, SensorProto.Sensor>> sensorMap =
        legacySensorStream.split(Named.as("sensor-"))   ❶
        .branch(isTemperatureSensor, Branched.as("temperature"))   ❷
        .branch(isProximitySensor,
            Branched.withFunction(
                ps -> ps.mapValues(feetToMetersMapper), "proximity"))   ❸
        .noDefaultBranch();   ❹
```

❶ Splitting the stream and providing the base name for the map keys

❷ Creating the temperature reading branch and naming the key

❸ Creating the proximity sensor branch with a ValueMapper function

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

❹ Specifying no default branch, because we know all records fall into only two categories

What's happening overall is each branch call places an entry into a `Map` where the key is the concatenation of name passed into the `KStream.split()` method and the string provided in the `Branched` parameter and the value is a `KStream` instance resulting from each `branch` call.

In the first branching example, the split and subsequent branching calls also returns a `Map`, but in that case it would have been empty. The reason is that when you pass in a `Branched.withConsumer` (a `java.util.Consumer` interface) it's a void method, it returns nothing, hence no entry is placed in the map. But the `Branched.withFunction` (a `java.util.Function` interface) accepts a `KStream<K, V>` object as a parameter and ***returns a KStream<K, V>*** instance so it goes into the map as an entry. At annotation three, the function takes the branched `KStream` object and executes a `MapValues` to convert the proximity sensor reading values from feet to meters, since the sensor records in the updated stream are in meters.

I'd like to point out some subtlety here, the `branch` call at annotation two does not provide a function, but it still ends up in the resulting `Map`, how is that so? When you only provide a `Branched` parameter with name, it's treated the same if you had used a `java.util.Function` that simply returns the provided `KStream` object, also known as an **identity function**. So what's the determining factor to use either `Branched.withConsumer` or `Branched.withFunction`? I can answer that question best by going over the next block of code in our example:

**Listing 6.17 Splitting the stream and gaining access to the newly created streams**

```
KStream<String, SensorProto.Sensor> temperatureSensorStream =    ❶
      builder.stream("temperature-sensors", sensorConsumed);

KStream<String, SensorProto.Sensor> proximitySensorStream =    ❷
      builder.stream("proximity-sensors", sensorConsumed);

temperatureSensorStream.merge(sensorMap.get("sensor-temperature"))
      .to("temp-reading", Produced.with(stringSerde, sensorSerde));   ❸

proximitySensorStream.merge(sensorMap.get("sensor-proximity"))
     .to("proximity-reading", Produced.with(stringSerde, sensorSerde));   ❹
```

❶ The stream with the new temperature IoT sensors

❷ The stream with the updated proximity IoT sensors

❸ Merging the legacy temperature readings with the new ones

❹ Merging the updated to meters proximity stream with the new proximity stream

To refresh your memory, the requirements for splitting the stream were to extract the different IoT sensor results by type and place them in the same stream as the new updated IoT results and

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

convert any proximity readings into meters. You accomplish this task by extracting the `KStream` from the map with the corresponding keys created in the branching code in the previous code block.

To accomplish putting the branched legacy stream with the new one, you use a DSL operator `KStream.merge` which is the functional analog of `KStream.split` it merges different `KStream` objects into one. With `KStream.merge` there is no ordering guarantees between records of the different streams, but the relative order of each stream remains. In other words the order of processing between the legacy stream and the updated one is not guaranteed to be in any order but the order inside in each stream is preserved.

So now it should be clear why you use `Branched.withConsumer` or `Branched.withFunction` in the latter case you need to get a handle on the branched `KStream` so you can integrate into the outer application in some manner, while with the former you don't need access to the branched stream.

That wraps up our discussion on branching and merging, so let's move on to cover naming topology nodes in the DSL.

### 6.5.3 Naming topology nodes

When you build a topology in the DSL, Kafka Streams creates a graph of processor nodes, giving each one a unique name. Kafka Streams generates these node names by taking the name the function of the processor and appending globally incremented number. To view this description of the topology, you'll need to get the `TopologyDescription` object. Then you can view it by printing it to the console.

**Listing 6.18 Getting a description of the topology and printing it out**

```
TopologyDescription topologyDescription =
  streamsBuilder.build().describe();
System.out.println(topologyDescription.toString());
```

Running the code above yields this output on the console:

**Listing 6.19 Full topology description of the KafkaStreamsYellingApplication**

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [src-topic])    ❶
      --> KSTREAM-MAPVALUES-0000000001    ❷
    Processor: KSTREAM-MAPVALUES-0000000001 (stores: [])    ❸
      --> KSTREAM-SINK-0000000002
      <-- KSTREAM-SOURCE-0000000000    ❹
    Sink: KSTREAM-SINK-0000000002 (topic: out-topic)    ❺
      <-- KSTREAM-MAPVALUES-0000000001
```

❶ The source node name

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

❷ The processor that the source node sends records to

❸ The name of the map values processor

❹ The processor that provided input the the map values processor

❺ The name of the sink node

From looking at the names, you can see the first node ends in a zero, with the second node `KSTREAM-MAPVALUES` ending in a one etc. The `Sub-topology` listing indicates a portion of the topology that is a distinct source node and every processor downstream of the source node is a member of the given `Sub-topology`. If you were to define a second stream with a new source, then that would show up as `Sub-topology: 1`. We'll see more about sub-topologies a bit later in the book when we cover repartitioning.

The arrows `->` pointing to the right show the flow of records in the topology. The arrows pointing left `<-` indicate the lineage of the record flow, where the current processor received records one. Note that a processor could forward records to more than one node and a single node could get input from multiple nodes.

Looking at this topology description, it's easy to get sense of the structure of the Kafka Streams application. However, once you start building more complex applications, the generic names with the numbers become hard to follow. For this reason, Kafka Streams provides a way to name the processing nodes in the DSL.

Almost all the methods in the Streams DSL have an overload that takes a `Named` object where you can specify the name used for the node in the topology. Being able to provide the name is important as you can make it relate to the processing nodes *role* in your application, not just what the processor *does*. Configuration objects like `Consumed` and `Produced` have a `withName` method for giving a name to the operator. Let's revisit the `KafkaStreamsYellingApplication` but this time we'll add a name for each processor:

**Listing 6.20 Updated KafkaStreamsYellingApplication with names**

```
builder.stream("src-topic",
               Consumed.with(stringSerde, stringSerde)
                       .withName("Application Input"))   ❶
       .mapValues((key, value) -> value.toUpperCase(),
                  Named.as("Convert to Yelling"))   ❷
       .to("out-topic",
            Produced.with(stringSerde, stringSerde)
                    .withName("Application Output"))   ❸
```

❶ Naming the source node

❷ Giving a name to the mapValues processor

❸ Naming the sink node

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

And the description from the updated topology with names will now look like this:

**Listing 6.21 Full topology description with provided names**

```
Topologies:
   Sub-topology: 0
    Source: Application-Input (topics: [src-topic])
      --> Convert-to-Yelling
    Processor: Convert-to-Yelling (stores: [])
      --> Application-Output
      <-- Application-Input
    Sink: Application-Output (topic: out-topic)
      <-- Convert-to-Yelling
```

Now you can view the topology description and get a sense of the role for each processor in the overall application, instead of just what the processor itself does. Naming the processor nodes becomes critical for your application when there is state involved, but we'll get to that in a later chapter.

Next we'll take a look at how you can use dynamic routing for your Kafka Streams application.

### 6.5.4 Dynamic routing of messages

Say you need to differentiate which department of the store the purchase comes from—housewares, say, or shoes. You can use dynamic routing to accomplish this task on a per-record basis. The `KStream.to()` method has an overload that takes a `TopicNameExtractor` which will dynamically determine the correct Kafka topic name to use. Note that the topics need to exist ahead of time, by default Kafka Streams will not create extracted topic names automatically.

So, if we go back to the branching example each object has a `department` field, so instead of creating a branch we will process these events with everything else and use the `TopicNameExtractor` to determine the topic where we route the events to.

The `TopicNameExtractor` has one method `extract` which you implement to provide the logic for determining the topic name. What you've going to do here is check if the department of the purchase matches one of the special conditions for routing the purchase events to a different topic. If it does match, then return the name of the department for the topic name (knowing they've been created ahead of time). Otherwise return the name of topic where the rest of the purchase events are sent to.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**Listing 6.22 Implementing the extract method to determine the topic name based on purchase department**

```
@Override
public String extract(String key,
                      Purchase value,
                      RecordContext recordContext) {
    String department = value.getDepartment();
    if (department.equals("coffee")
            || department.equals("electronics")) {   ❶
        return department;
    } else {
        return "purchases";                          ❷
    }
}
```

❶ Checking if the department matches one of special cases

❷ The default case for the topic name

**NOTE** **The `TopicNameExtractor` interface only has one method to implement, I've chosen to use a concrete class because you can then write a test for it.**

Although the code example here is using the value to determine the topic to use, it could very well use the key or a combination of the key and the value. But the third parameter to the `TopicNameExtractor#extract` method is a `RecordContext` object. Simply stated the `RecordContext` is the context associated with a record in Kafka Streams.

The context contains metadata about the record- the original timestamp of the record, the original offset from Kafka, the topic and partition it was received, and the `Headers`. We discussed headers in the chapter on Kafka clients, so I won't go into the details again here. One of the primary use cases for headers is routing information, and Kafka Streams exposes them via the `ProcessorContext`. Here's one possible example for retrieving the topic name via a `Header`

In this example you'll extract the `Headers` from the record context. You first need to check that the `Headers` are not null, then you proceed to drill down to get the specific routing information. From there you return the name of topic to use based on the value stored in the `Header`. Since `Headers` are optional and may not exist or contain the specific "routing" `Header` you've defined a default value in the `TopicNameExtractor` and return it in the case where the output topic name isn't found.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

**Listing 6.23 Using information in a Header for dynamically determining the topic name to send a record to**

```
public String extract(String key,
                      PurchaseProto.Purchase value,
                      RecordContext recordContext) {

       Headers headers = recordContext.headers();   ❶
      if (headers != null) {
       Iterator<Header> routingHeaderIterator =
         headers.headers("routing").iterator();

       if (routingHeaderIterator.hasNext()) {
            Header routing = routingHeaderIterator.next();   ❷

            return new String(routing.value(),
                              StandardCharsets.UTF_8);   ❸
       }
     }
      return defaultTopicName;   ❹
   }
```

❶ Retrieving the headers from the `RecordContext`

❷ Extracting the specific routing `Header`

❸ Returning the name of the topic to use from the `Header` value

❹ If no routing information found, return a default topic name

Now you've learned about using the Kafka Streams DSL API.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

## 6.6 Summary

- Kafka Streams is a graph of processing nodes called a topology. Each node in the topology is responsible for performing some operation on the key-value records flowing through it. A Kafka Streams applciation is minimally composed of a source node that consumes records from a topic and sink node that produces results back to a Kafka topic. You configure a Kafka Streams application minimally with the application-id and the bootstrap servers configuration. Multiple Kafka Streams applications with the same application-id are logically considered one application.
- 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.
- To selectively process records you can use the `KStream.filter` operation where records that don't match a predicate get dropped. 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. There is also the `KStream.filterNot` method that does the opposite - it only forwards key-value pairs that ***don't match*** the predicate.
- 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. Branching is an elegant way of splitting a stream up into multiple streams where each stream can operate independantly. To perform the opposite action there is `KStream.merge` which you can use to merge 2 `KStream` objects into one stream.
- You can modify an existing key or create a new one using the `KStream.selectKey` method.
- For viewing records in the topology you can use either `KStream.print` or `KStream.peek` (by providing a `ForeachAction` that does the acutal printing). `KStream.print` is a terminal operation meaning that you can't chain methods after calling it. `KStream.peek` returns a `KStream` instance and this makes it easier to embed before and after `KStream` methods.
- You can view the generated graph of a Kafka Streams application by using the `Topology.describe` method. All graph nodes in Kafka Streams have auto-generated names by default which can make the graph hard to understand when the application grows in complexity. You can avoid this situation by providing names to each `KStream` method so when you print the graph, you have names describing the role of each node.
- You can route records to different topics by passing a `TopicNameExtractor` as a parameter to the `KStream.to` method. The `TopicNameExtractor` can inspect the key, value, or headers to determine the corect topic name to use for producing records back to Kafka. The topics must be created ahead of time.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion