Getting Started with Kafka Streams

Kafka Streams is a lightweight, yet powerful Java library for enriching, transforming, and processing real-time streams of data.

By the end of this topic, you will understand the following:

- Where Kafka Streams fits in the Kafka ecosystem
- Why Kafka Streams was built in the first place
- What kinds of features and operational characteristics are present in this library
- Who Kafka Streams is appropriate for
- How Kafka Streams compares to other stream processing solutions
- How to create and run a basic Kafka Streams application

The Kafka Ecosystem

Kafka Streams lives among a group of technologies that are collectively referred to as the *Kafka ecosystem*. Three APIs in the Kafka ecosystem, which are summarized in Table 2-1, are concerned with the *movement* of data to and from Kafka.

Table 2-1. APIs for moving data to and from Kafka

API	Topic interaction	Examples
Producer API	*Writing* messages to Kafka topics.	• Filebeat • rsyslog • Custom producers
Consumer API	*Reading* messages from Kafka topics.	• Logstash • kafkacat • Custom consumers
Connect API	*Connecting* external data stores, APIs, and filesystems to Kafka topics. Involves both *reading* from topics (sink connectors) and *writing* to topics (source connectors).	• JDBC source connector • Elasticsearch sink connector • Custom connectors

However, while moving data through Kafka is certainly important for creating data pipelines, some business problems require us to also *process* and *react* to data as it becomes available in Kafka. This is referred to as *stream processing*, and there are multiple ways of building stream processing applications with Kafka. Therefore, let's take a look at how stream processing applications were implemented before Kafka Streams was introduced, and how a dedicated stream processing library came to exist alongside the other APIs in the Kafka ecosystem.

Before Kafka Streams

Before Kafka Streams existed, there was a void in the Kafka ecosystem. The kind of void that made building stream processing applications more difficult than it needed to be. I'm talking about the lack of library support for processing data in Kafka topics.

During these early days of the Kafka ecosystem, there were two main options for building Kafka-based stream processing applications:

- Use the Consumer and Producer APIs directly
- Use another stream processing framework (e.g., Apache Spark Streaming, Apache Flink)

With the Consumer and Producer APIs, you can read from and write to the event stream directly using a number of programming languages (Python, Java, Go, C/C++, Node.js, etc.) and perform any kind of data processing logic you'd like, as long as you're willing to write a lot of code from scratch. These APIs are very basic and lack many of the primitives that would qualify them as a stream processing API, including:

- Local and fault-tolerant state
- A rich set of operators for transforming streams of data
- More advanced representations of streams
- Sophisticated handling of time

Therefore, if you want to do anything nontrivial, like aggregate records, join separate streams of data, group events into windowed time buckets, or run ad hoc queries against your stream, you will hit a wall of complexity pretty quickly. The Producer and Consumer APIs do not contain any abstractions to help you with these kinds of use cases, so you will be left to your own devices as soon as it's time to do somethingmore advanced with your event stream.

The second option, which involves adopting a full-blown streaming platform like Apache Spark or Apache Flink, introduces a lot of unneeded complexity. If we're optimizing for simplicity *and* power, then we need a solution that gives us the stream processing primitives without the overheadof a processing cluster. We also need better

integration with Kafka, especially when it comes to working with intermediate representations of data outside of the source and sink topics.

Enter Kafka Streams

In 2016, the Kafka ecosystem was forever changed when the first version of Kafka Streams (also called the *Streams API*) was released. With its inception, the landscape of stream processing applications that relied so heavily on hand-rolled features gave way to more advanced applications, which leveraged community-developed patterns and abstractions for transforming and processing real-time event streams.

Unlike the Producer, Consumer, and Connect APIs, Kafka Streams is dedicated to helping you *process* real-time data streams, not just *move* data to and from Kafka. It makes it easy to consume real-time streams of events as they move through our data pipeline, apply data transformation logic using a rich set of stream processing operators and primitives, and optionally write new representations of the data back to Kafka (i.e., if we want to make the transformed or enriched events available to downstream systems).

Figure 2-1 depicts the previously discussed APIs in the Kafka ecosystem, with Kafka Streams operating at the stream processing layer.

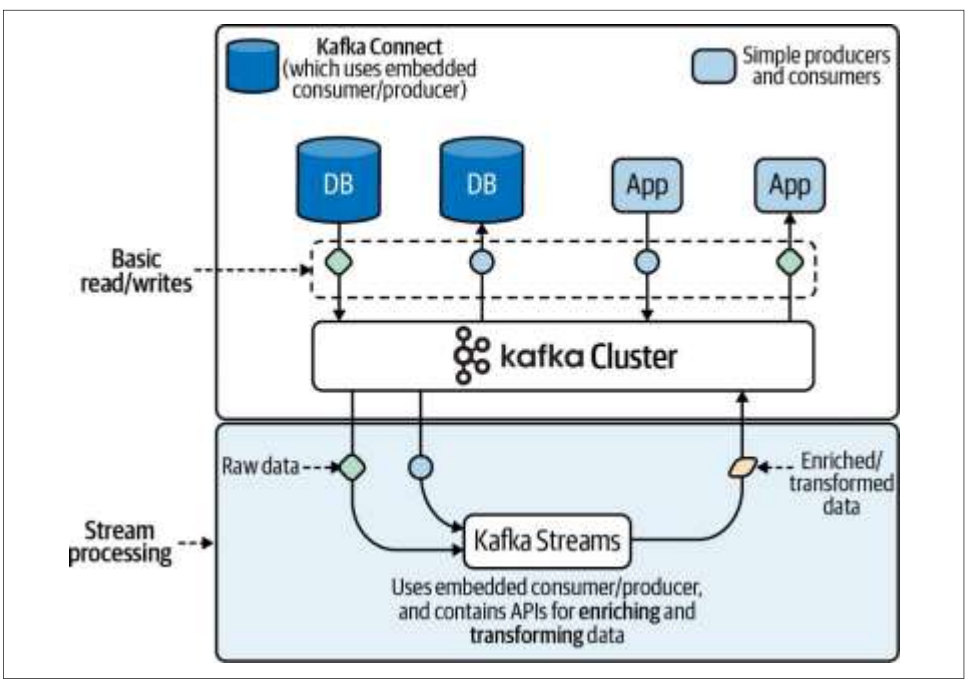

Figure 2-1. Kafka Streams is the "brain" of the Kafka ecosystem, consuming records from the event stream, processing the data, and optionally writing enriched or transformed records back to Kafka

As you can see from the diagram in Figure 2-1, Kafka Streams operates at an exciting layer of the Kafka ecosystem: the place where data from many sources converges. This is the layer where sophisticated data *enrichment*, *transformation*, and *processing*can happen. It's the same place where, in a pre–Kafka Streams world, we would have

tediously written our own stream processing abstractions (using the Consumer/ Producer API approach) or absorbed a complexity hit by using another framework.

Features at a Glance

Kafka Streams offers many features that make it an excellent choice for modern stream processing applications:

- A high-level DSL that looks and feels like Java's streaming API. The DSL provides a fluent and functional approach to processing data streams that is easy to learn and use.
- A low-level Processor API that gives developers fine-grained control when they need it.
- Convenient abstractions for modeling data as either streams or tables.
- The ability to join streams and tables, which is useful for data transformation and enrichment.
- Operators and utilities for building both stateless and stateful stream processing applications.
- Support for time-based operations, including windowing and periodic functions.
- Easy installation. It's just a library, so you can add Kafka Streams to any Java application.
- Scalability, reliability, maintainability.

Both the high-level DSL and low-level ProcessorAPI are not only easy to learn, but are also extremely powerful. Advanced stream processing tasks (such as joining live, moving streams of data) can be accomplished with very little code, which makes the development experience truly enjoyable.

Now, the last bullet point pertains to the long-term stability of our stream processing applications. After all, many technologies are exciting to learn in the beginning, but what really counts is whether or not Kafka Streams will continue to be a good choice as our relationship gets more complicated through real-world, long-term usage of this library. Therefore, it makes sense to evaluate the long-term viability of Kafka

Operational Characteristics

Three important goals for data systems:

- Scalability
- Reliability
- Maintainability

These goals provide a useful framework for evaluating Kafka Streams, so in this section, we will define these terms and discover how Kafka Streams achieves each of them.

Scalability

Systems are considered *scalable* when they can cope and remain performant as load increases.

In Kafka Streams, the unit of work is a single topic-partition, and Kafka automatically distributes work to groups of cooperating consumers called consumer groups. This has two important implications:

- Since the unit of work in Kafka Streams is a single topic-partition, and since topics can be expanded by adding more partitions, the amount of work a Kafka Streams application can undertake can be scaled by increasing the number of partitions on the source topics.
- By leveraging consumer groups, the total amount of work being handled by a Kafka Streams application can be distributed across multiple, cooperating instances of your application.

A quick note about the second point. When you deploy a Kafka Streams application, you will almost always deploy multiple application instances, each handling a subset of the work (e.g., if your source topic has 32 partitions, then you have 32 units of work that can be distributed across all cooperating consumers). For example, you could deploy four application instances, each handling eight partitions (`4 * 8 = 32`), or you could just as easily deploy sixteen instances of your application, each handlingtwo partitions (`16 * 2 = 32`).

However, regardless of how many application instances you end up deploying, Kafka Streams' ability to cope with increased load by adding more partitions (units of work) and application instances (workers) makes Kafka Streams *scalable*.

On a similar note, Kafka Streams is also *elastic*, allowing you to seamlessly scale the number of application instances in or out, with a limit on the scale-out path being the number of tasks that are created for your topology.

Reliability

Reliability is an important feature of data systems, not only from an engineering perspective, butalso from our customers' perspective. Kafka Streamscomes with a few fault-tolerant features, but the most obvious one is: automatic failovers andpartition rebalancing via consumer groups.

If you deploy multiple instances of your Kafka Streams application and one goes offline due to some fault in the system (e.g., a hardware failure), then Kafka will automatically redistribute the load to the other healthy instances. When the failure is resolved, then Kafka will rebalance the work again. This ability to gracefully handle faults makes Kafka Streams *reliable*.

Maintainability

> It is well known that the majority of the cost of software is not in its initial development, but in its ongoing maintenance—fixing bugs, keeping its systems operational, investigating failures…
>
> —Martin Kleppmann

Since Kafka Streams is a Java library, troubleshooting and fixing bugs is relatively straightforward since we're working with standalone applications, and patterns for both troubleshooting and monitoring Java applications are well established and may already be in use at your organization (collecting and analyzing application logs, capturing application and JVM metrics, profiling and tracing, etc.).

Furthermore, since the Kafka Streams API is succinct and intuitive, code-level maintenance is less time-consuming than one would expect with more complicated libraries, and is very easy to understand for beginners and experts alike. If you build a Kafka Streams application and then don't touch it for months, you likely won't sufferthe usual project amnesia and require a lot of ramp-up time to understand the previous code you've written. For the same reasons, new project maintainers can typically get up to speed pretty quickly with a Kafka Streams application, which improves maintainability even further.

Comparison to Other Systems

We'll start by comparing Kafka Streams' deployment model with other popular systems.

Deployment Model

Kafka Streams takes a different approach to stream processing than technologies like Apache Flink and Apache Spark Streaming. The latter systems require you to set up a dedicated processing cluster for submitting and running your stream processing program. This can introduce a lot of complexity and overhead. Even experienced engineers at well-established companies have conceded that the overhead of a processing cluster is nontrivial.

On the other hand, Kafka Streams is implemented as a Java *library*, so getting started is much easier since you don't need to worry about a cluster manager; you simply need to add the Kafka Streams dependency to your Java application. Being able to build a stream processing application as a standalone program also means you have a lot of freedom in terms of how you monitor, package, and deploy your code. For example, at Mailchimp, our Kafka Streams applications are deployed using the same patterns and tooling we use for other internal Java applications. This ability to immediately integrate into your company's systems is a huge advantage for Kafka Streams.

Processing Model

Another key differentiator between Kafka Streams and systems like Apache Spark Streaming or Trident is that Kafka Streams implements *event-at-a-time processing*, so events are processed immediately, one at a time, as they come in. This is considered true streaming and provides lower latency than the alternative approach, which is called *micro-batching*. Micro-batching involves grouping records into small groups (or *batches*), which are buffered in memory and later emitted at some interval (e.g., every 500 milliseconds). Figure 2-2 depicts the difference between event-at-a-time and micro-batch processing.

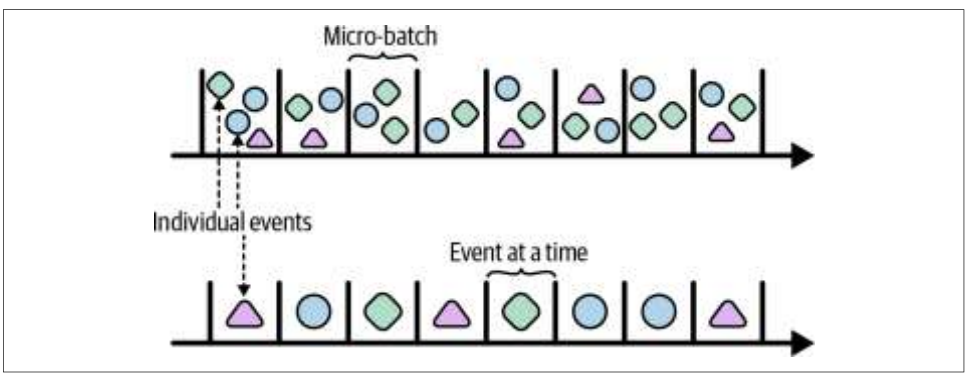

Figure 2-2. Micro-batching involves grouping records into small batches and emitting them to downstream processors at a fixed interval; event-at-a-time processing allows each event to be processed at soon as it comes in, instead of waiting for a batch to materialize

Frameworks that use micro-batching are often optimized for greater *throughput* at the cost of higher *latency*. In Kafka Streams, you can achieve extremely low latency while also maintaining high throughput by splitting data across many partitions.

Finally, let's take a look at Kafka Streams' data processing architecture, and see how its focus on streaming differs from other systems.

Kappa Architecture

Another important consideration when comparing Kafka Streams to other solutions is whether or not your use case requires support for both batch and stream processing. Kafka Streams focuses solely on streaming use cases (this is called a *Kappa architecture*), while frameworks like Apache Flink and Apache Spark support both batch and stream processing (this is called a *Lambda architecture*). However, architectures that support both batch and streaming use cases aren't without their drawbacks. Jay Kreps discussed some of the disadvantages of a hybrid system nearly two years before Kafka Streams was introduced into the Kafka ecosystem:

> The operational burden of running and debugging two systems is going to be very high. And any new abstraction can only provide the features supported by the intersection of the two systems.

These challenges didn't stop projects like Apache Beam, which defines a unified programming model for batch and stream processing, from gaining popularity in recent years. But Apache Beam isn't comparable to Kafka Streams in the same way that Apache Flink is. Instead, Apache Beam is an API layer that relies on an execution engine to do most of the work. For example, both Apache Flink and Apache Spark can be used as execution engines (often referred to as *runners*) in Apache Beam. So when you compare Kafka Streams to Apache Beam, you must also consider the execution engine that you plan on using in addition to the Beam API itself.

Furthermore, Apache Beam–driven pipelines lack some important features that are offered in Kafka Streams. Robert Yokota, who created an experimental Kafka Streams Beam Runner and who maintains several innovative projects in the Kafka ecosystem, puts it this way in his comparison of different streaming frameworks:

> One way to state the differences between the two systems is as follows:
>
> - Kafka Streams is a *stream-relational* processing platform.
> - Apache Beam is a *stream-only* processing platform.
>
> A stream-relational processing platform has the following capabilities which are typically missing in a stream-only processing platform:
>
> - Relations (or tables) are first-class citizens, i.e., each has an independent identity.
> - Relations can be transformed into other relations.
> - Relations can be queried in an ad-hoc manner.

Many of Kafka Streams' most powerful features (including the ability to query the state of a stream) are not available in Apache Beam or other more generalized frameworks. Furthermore, the Kappa architecture offers a simpler and more specialized approach for working with streams of data, which can improve the development experience and simplify the operation and maintenance of your software. So if your use case doesn't require batch processing, then hybrid systems will introduce unnecessary complexity.

Use Cases

Kafka Streams is optimized for processing unbounded datasets quickly and efficiently, and is therefore a great solution for problems in low-latency, time-critical domains. A few example use cases include:

- Financial data processing (Flipkart), purchase monitoring, fraud detection
- Algorithmic trading
- Stock market/crypto exchange monitoring
- Real-time inventory tracking and replenishment (Walmart)
- Event booking, seat selection (Ticketmaster)
- Email delivery tracking and monitoring (Mailchimp)
- Video game telemetry processing (Activision, the publisher of *Call of Duty*)
- Search indexing (Yelp)
- Geospatial tracking/calculations (e.g., distance comparison, arrival projections)
- Smart Home/IoT sensor processing (sometimes called AIOT, or the Artificial Intelligence of Things)
- Change data capture (Redhat)
- Sports broadcasting/real-time widgets (Gracenote)
- Real-time ad platforms (Pinterest)
- Predictive healthcare, vitals monitoring (Children's Healthcare of Atlanta)
- Chat infrastructure (Slack), chat bots, virtual assistants
- Machine learning pipelines (Twitter) and platforms (Kafka Graphs)

The list goes on and on, but the common characteristic across all of these examples is that they require (or at least benefit from) *real-time decision making* or data processing. The spectrum of these use cases, and others you will encounter in the wild, is really quite fascinating. On one end of the spectrum, you may be processing streams at the hobbyist level by analyzing sensor output from a Smart Home device. However, you could also use Kafka Streams in a healthcare setting to monitor and react to changes in a trauma victim's condition, as Children's Healthcare of Atlanta has done.

Kafka Streams is also a great choice for building microservices on top of real-time event streams. It not only simplifies typical stream processing operations (filtering, joining, windowing, and transforming data), it is also capable of exposing the state of a stream using a feature called *interactive queries*. The state of a stream could be an aggregation of some kind(e.g., the total number of views for each video in a streaming platform) or even the latest representation for a rapidly changing entity in your event

stream (e.g., the lateststock price for a given stock symbol).

Now that you have some idea of who is using Kafka Streams and what kinds of use cases it is well suited for, let's take a quick look at Kafka Streams' architecture before we start writing any code.

Processor Topologies

Kafka Streams leverages a programming paradigm called *dataflow programming* (DFP), which is a data-centric method of representing programs as a series of inputs, outputs, and processing stages. This leads to a very natural and intuitive way of creating stream processing programs and is one of the many reasons I think KafkaStreams is easy to pick up for beginners.

Instead of building a program as a sequence of steps, the stream processing logic in a Kafka Streams application is structured as a directed acyclic graph (DAG). Figure 2-3 shows an example DAG that depicts how data flows through a set of stream processors. The nodes (the rectangles in the diagram) represent a processing step, or processor, and the edges (the lines connecting the nodes in the diagram) represent input and output streams (where data flows from one processor to another).

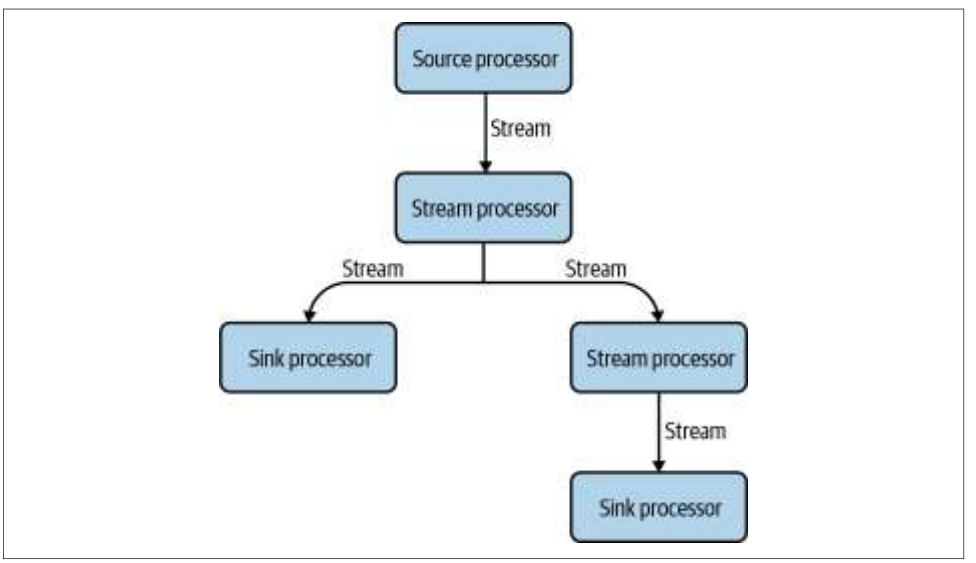

Figure 2-3. Kafka Streams borrows some of its design from dataflow programming, and structures stream processing programs as a graph of processors through which data flows

There are three basic kinds of processors in Kafka Streams:

Source processors
: Sources are where information flows into the Kafka Streams application. Data is read from a Kafka topic and sent to one or more *stream processors*.

Stream processors
: These processors are responsible for applying data processing/transformation logic on the input stream. In the high-level DSL, these processors are defined using a set of built-in *operators* that are exposed by the Kafka Streams library. Some example operators are `filter`, `map`, `flatMap`, and `join`.

Sink processors
: Sinks are where enriched, transformed, filtered, or otherwise processed records are written *back* to Kafka, either to be handled by another stream processing application or to be sent to a downstream data store via something like Kafka Connect. Like source processors, sink processors are connected to a Kafka topic.

A collection of processors forms a *processor topology*, which is often referred to as simply *the topology*.

Scenario

Say we are building a chatbot with Kafka Streams, and we have a topic named `slack-mentions` that contains every Slack message that mentions our bot, `@StreamsBot`. We will design our bot so that it expects each mention to be followed by a command, like `@StreamsBot restart myservice`.

We want to implement a basic processor topology that does some preprocessing/validation of these Slack messages. First, we need to consume each message in the source topic, determine if the command is valid, and if so, write the Slack message to a topic called `valid-mentions`. If the command is not valid (e.g., someone makes a spelling error when mentioning our bot, such as `@StreamsBot restart serverrr`), then we will write to a topic named `invalid-mentions`).

In this case, we would translate these requirements to the topology shown in Figure 2-4.

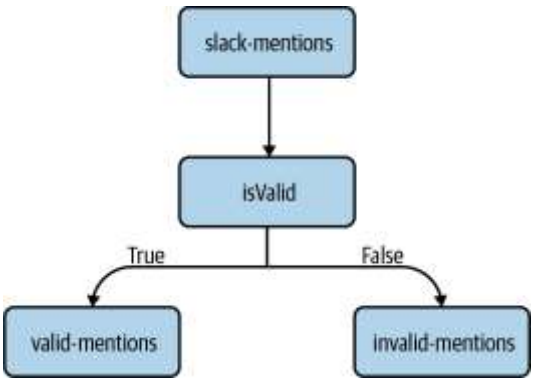

Figure 2-4. An example processor topology that contains a single source processor for reading Slack messages from Kafka (slack-mentions), a single stream processor that checks the validity of each message (isValid), and two sink processors that route the message to one of two output topics based on the previous check (valid-mentions, invalid-mentions)

Sub-Topologies

Kafka Streams also has the notion of sub-topologies. In the previous example, we designed a processor topology that consumes events from a single source topic(`slack-mentions`) and performs some preprocessing on a stream of raw chat messages. However, if our application needs to consume from multiple source topics, then Kafka Streams will (under most circumstances) divide our topology into smaller sub-topologies to parallelize the work even further. This division of work is possible since operations on one input stream can be executed independently of operations on another input stream.

For example, let's keep building our chatbot by adding two new stream processors: one that consumes from the `valid-mentions` topic and performs whatever command was issued to `StreamsBot` (e.g., `restart server`), and another processor that consumes from the `invalid-mentions` topic and posts an error response back to Slack.

As you can see in Figure 2-5, our topology now has three Kafka topics it reads from: `slack-mentions`, `valid-mentions`, and `invalid-mentions`. Each time we read from a new source topic, Kafka Streams divides the topology into smaller sections that it can execute independently. In this example, we end up with three sub-topologies for our chatbot application, each denoted by a star in the figure.

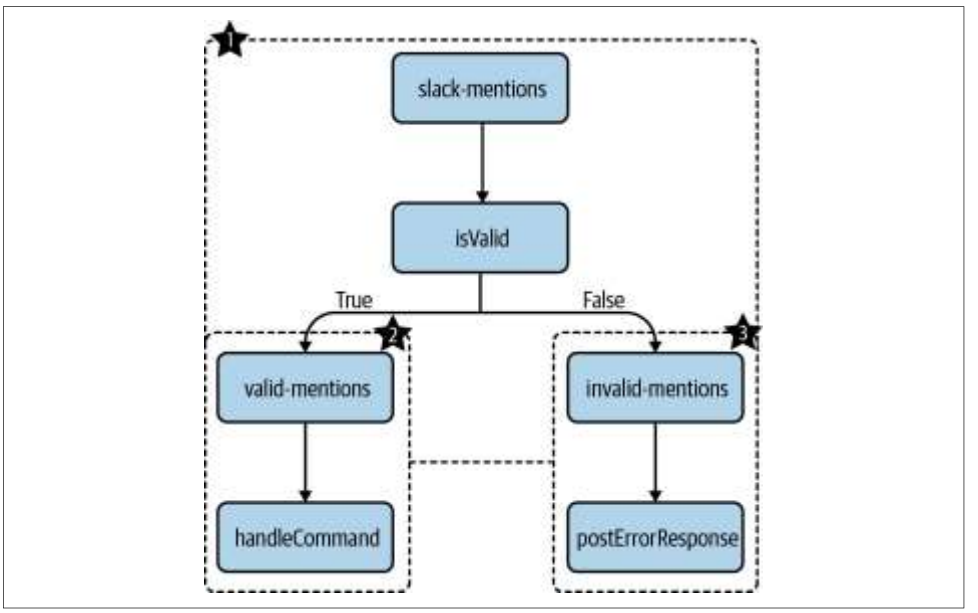

Figure 2-5. A processor topology, subdivided into sub-topologies (demarcated by dotted lines)

Notice that both the `valid-mentions` and `invalid-mentions` topics serve as a sink processor in the first sub-topology, but as a source processor in the second and third sub-topologies. When this occurs, there is no direct data exchange between sub-topologies. Records are produced to Kafka in the sink processor, and reread from Kafka by the source processors.

Now that we understand how to represent a stream processing program as a processor topology, let's take a look at how data actually flows through the interconnected processors in a Kafka Streams application.

Depth-First Processing

Kafka Streams uses a depth-first strategy when processing data. When a new record is received, it is routed through each stream processor in the topology before another record is processed. The flow of data through Kafka Streams looks something like what's shown in Figure 2-6.

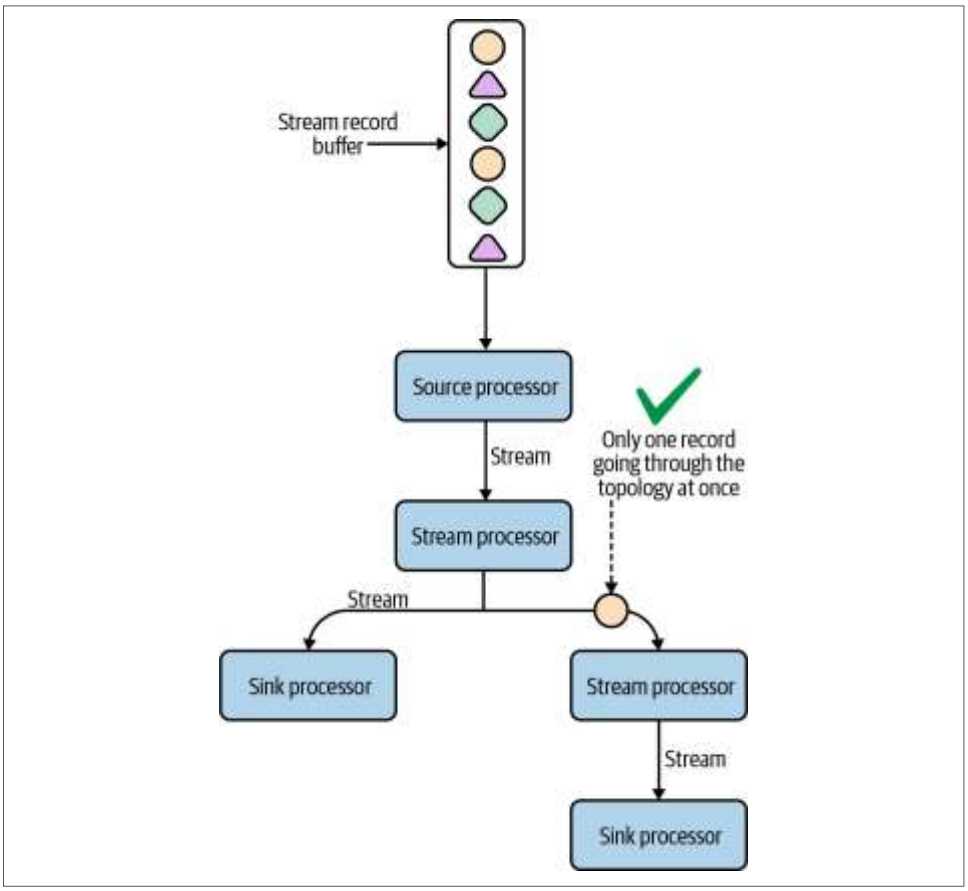

Figure 2-6. In depth-first processing, a single record moves through the entire topology before another record is processed

This depth-first strategy makes the dataflow much easier to reason about, but also means that slow stream processing operations can block other records from being processed in the same thread. Figure 2-7 demonstrates something you will never see happen in Kafka Streams: multiple records going through a topology at once.

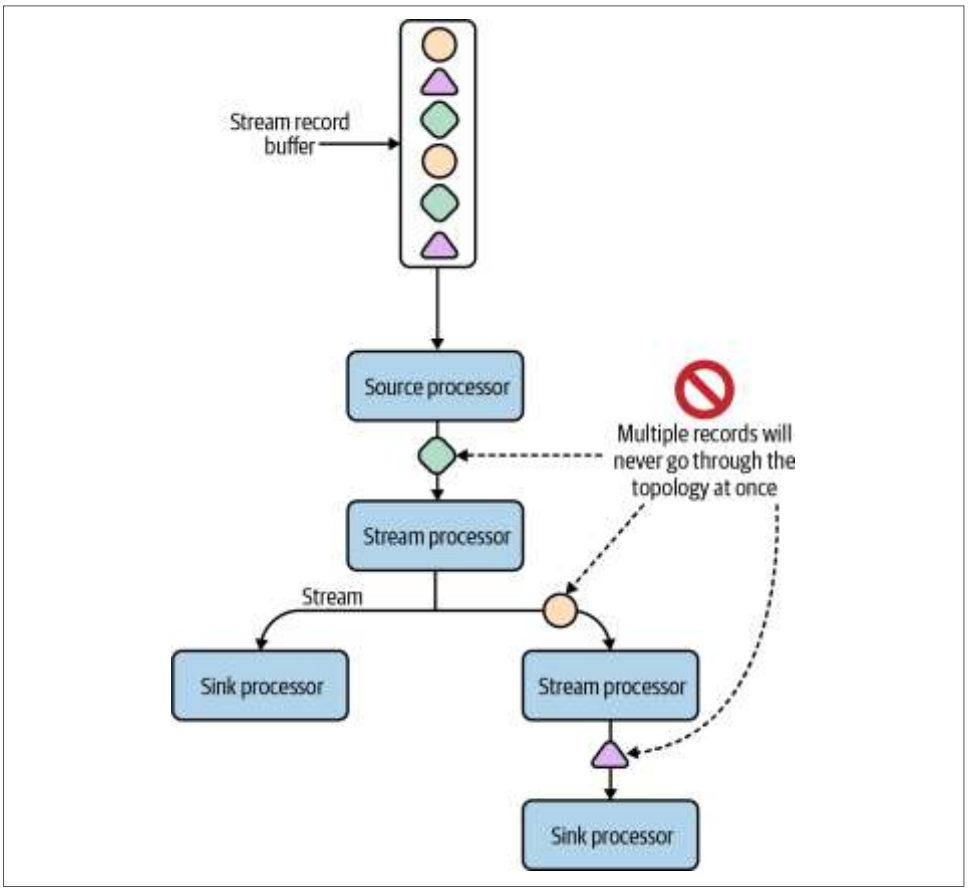

Figure 2-7. Multiple records will never go through the topology at the same time

When multiple sub-topologies are in play, the single-event rule does not apply to the entire topology, but to each sub-topology.

Now that we know how to design processor topologies and how data flows through them, let's take a look at the advantages of this data-centric approach to building stream processing applications.

Benefits of Dataflow Programming

There are several advantages of using Kafka Streams and the dataflow programming model for building stream processing applications. First, representing the program as a directed graph makes it easy to reason about. You don't need to follow a bunch of conditionals and control logic to figure out how data is flowing through your application. Simply find the source and sink processors to determine where data enters and exits your program, and look at the stream processors in between to discover how the data is being processed, transformed, and enriched along the way.

Furthermore, expressing our stream processing program as a directed graph allows us to standardize the way we frame real-time data processing problems and, subsequently, the way we build our streaming solutions. A Kafka Streams application that I write will have some level of familiarity to anyone who has worked with Kafka Streams before in their own projects—not only due to the reusable abstractions available in the library itself, but also thanks to a common problem-solving approach: defining the flow of data using operators (nodes) and streams (edges). Again, this makes Kafka Streams applications easy to reason about and maintain.

Directed graphs are also an intuitive way of visualizing the flow of data for nontechnical stakeholders. There is often a disconnect about how a program works between engineering teams and nonengineering teams. Sometimes, this leads to nontechnical teams treating the software as a closed box. This can have dangerous side effects, especially in the age of data privacy laws and GDPR compliance, which requires close coordination between engineers, legal teams, and other stakeholders. Thus, being able to simply communicate how data is being processed in your application allows people who are focused on another aspect of a business problem to understand or even contribute to the design of your application.

Finally, the processor topology, which contains the source, sink, and stream processors, acts as a *template* that can be instantiated and parallelized very easily across multiple threads and application instances. Therefore, defining the dataflow in this way allows us to realize performance and scalability benefits since we can easily replicate our stream processing program when data volume demands it.

Now, to understand how this process of replicating topologies works, we first need to understand the relationship between tasks, stream threads, and partitions.

Tasks and Stream Threads

When we define a topology in Kafka Streams, we are not actually executing the program. Instead, we are building a template for how data should flow through our application. This template (our topology) can be instantiated multiple times in a single application instance, and parallelized across many *tasks* and *stream threads* (which

we'll refer to as simply threads going forward.) There is a close relationship between the number of tasks/threads and the amount of work your stream processing application can handle, so understanding the content in this section is especially important for achieving good performance with Kafka Streams.

Let's start by looking at tasks:

> A task is the smallest unit of work that can be performed in parallel in a Kafka Streams application…
>
> Slightly simplified, the maximum parallelism at which your application may run is bounded by the maximum number of stream tasks, which itself is determined by the maximum number of partitions of the input topic(s) the application is reading from.
>
> —Andy Bryant

Translating this quote into a formula, we can calculate the number of tasks that can be created for a given Kafka Streams sub-topology with the following math:

```
max(source_topic_1_partitions, ... source_topic_n_partitions)
```

For example, if your topology reads from one source topic that contains `16` partitions, then Kafka Streams will create `16` tasks, each of which will instantiate its own copy of the underlying processor topology. Once Kafka Streams has created all of the tasks, it will assign the source partitions to be read from to each task.

As you can see, tasks are just logical units that are used to instantiate and run a processor topology. *Threads*, on the other hand, are what actually execute the task. In Kafka Streams, the stream threads are designed to be isolated and thread-safe. Furthermore, unlike tasks, there isn't any formula that Kafka Streams applies to figure out how many threads your application should run. Instead, you are responsible for specifying the thread count using a configuration property named `num.stream.threads`. The upper bound for the number of threads you can utilize corresponds to the task count, and there are different strategies for deciding on the number of stream threads you should run with.

Now, let's improve our understanding of these concepts by visualizing how tasks and threads are created using two separate configs, each specifying a different number of threads. In each example, our Kafka Streams application is reading from a source topic that contains four partitions (denoted by `p1` - `p4` in Figure 2-8).

First, let's configure our application to run with two threads (`num.stream.threads = 2`). Since our source topic has four partitions, four tasks will be created and distributed across each thread. We end up with the task/thread layout depicted inFigure 2-8.

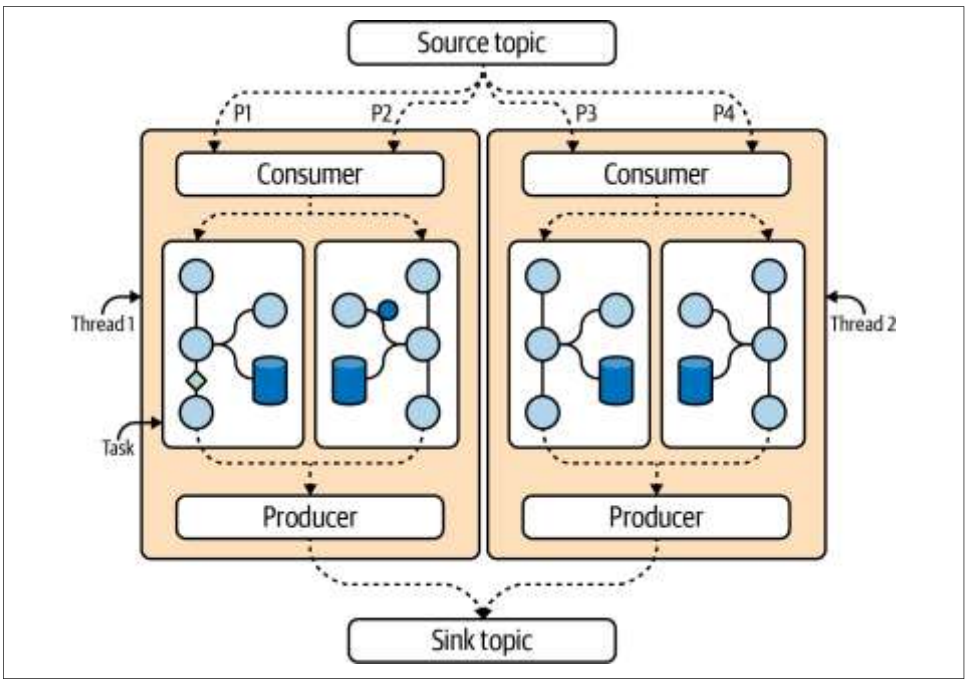

Figure 2-8. Four Kafka Streams tasks running in two threads

Running more than one task per thread is perfectly fine, but sometimes it is often desirable to run with a higher thread count to take full advantage of the available CPU resources. Increasing the number of threads doesn't change the number of tasks, but it does change the distribution of tasks among threads. For example, if we reconfigure the same Kafka Streams application to run with four threads instead of two (`num.stream.threads = 4`), we end up with the task/thread layout depicted in Figure 2-9.

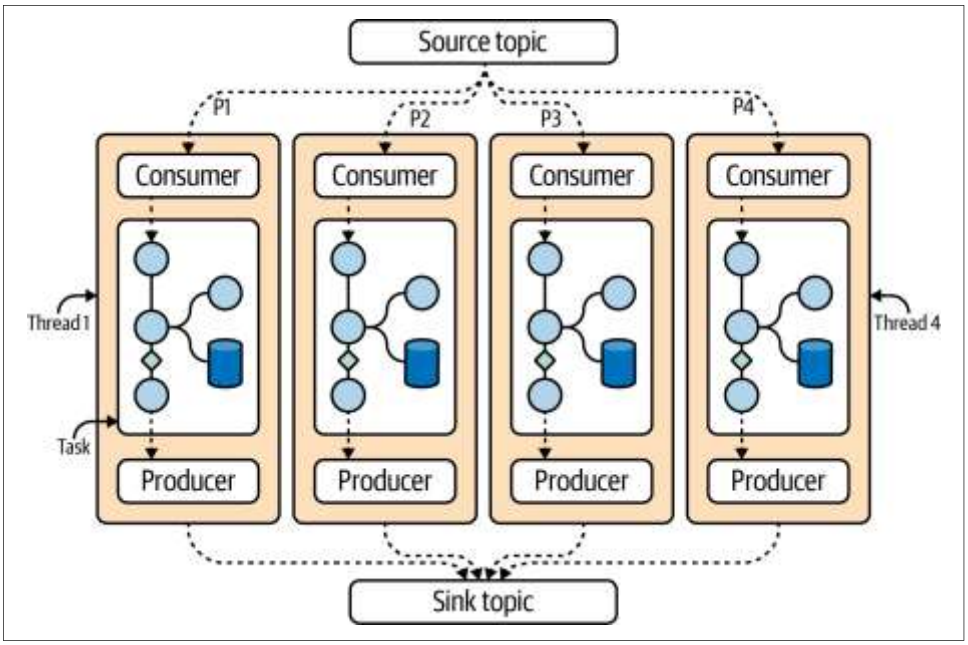

Figure 2-9. Four Kafka Streams tasks running in four threads

Now that we've learned about Kafka Streams' architecture, let's take a look at the APIs that Kafka Streams exposes for creating stream processing applications.

High-Level DSL Versus Low-Level Processor API

> Different solutions present themselves at different layers of abstraction.
>
> —James Clear

A common notion in the software engineering field is that abstraction usually comes at a cost: the more you abstract the details away, the more the software feels like "magic," and the more control you give up. As you get started with Kafka Streams, you may wonder what kind of control you will be giving up by choosing to implement a stream processing application using a high-level library instead of designing your solution using the lower-level Consumer/Producer APIs directly.

Luckily for us, Kafka Streams allows developers to choose the abstraction level that works best for them, depending on the project and also the experience and preference of the developer.

The two APIs you can choose from are:

- The high-level DSL
- The low-level Processor API

The relative abstraction level for both the high-level DSL and low-level Processor API is shown in Figure 2-10.

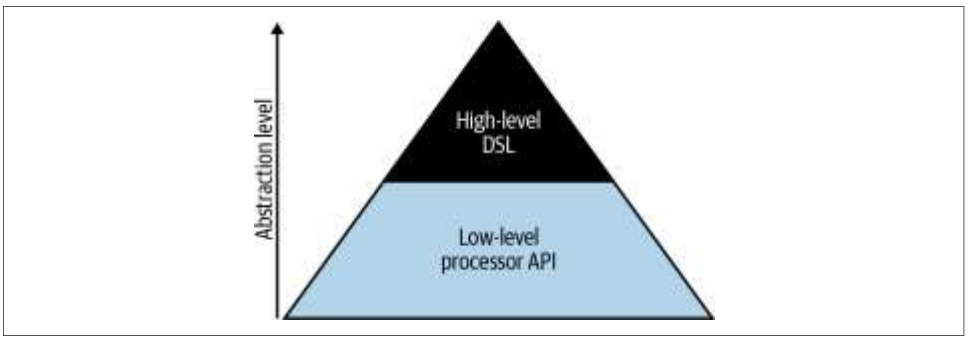

Figure 2-10. Abstraction levels of Kafka Streams APIs

The high-level DSL is built on top of the Processor API, but the interface each exposes is slightly different. If you would like to build your stream processing application using a functional style of programming, and would also like to leverage some higher-level abstractions for working with your data (streams and tables), then the DSL is for you.

On the other hand, if you need lower-level access to your data (e.g., access to record metadata), the ability to schedule periodic functions, more granular access to your application state, or more fine-grained control over the timing of certain operations, then the Processor API is a better choice.

Now, the best way to see the difference between these two abstraction levels is with a code example. Let's move on to our first Kafka Streams tutorial: Hello Streams.

Introducing Our Tutorial: Hello, Streams

In this section, we will get our first hands-on experience with Kafka Streams. This is a variation of the "Hello, world" tutorial that has become the standard when learning new programming languages and libraries. There are two implementations of this tutorial: the first uses the high-level DSL, while the second uses the low-levelProcessor API. Both programs are functionally equivalent, and will print a simple

greeting whenever they receive a message from the `users` topic in Kafka (e.g., upon receiving the message `Mitch`, each application will print `Hello, Mitch`).

Before we get started, let's take a look at how to set up the project.

Project Setup

All of the tutorials in this book will require a running Kafka cluster, and the source code for each chapter will include a *docker-compose.yml* file that will allow you to run a development cluster using Docker. Since Kafka Streams applications are meant to run outside of a Kafka cluster (e.g., on different machines than the brokers), it's best to view the Kafka cluster as a separate infrastructure piece that is required but distinct from your Kafka Streams application.

To start running the Kafka cluster, clone the repository and change to the directory containing this chapter's tutorial. The following commands will do the trick:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-02/hello-streams
```

Then, start the Kafka cluster by running:

```
docker-compose up
```

The broker will be listening on port 29092. Furthermore, the preceding command will start a container that will precreate the `users` topic needed for this tutorial. Now,with our Kafka cluster running, we can start building our Kafka Streams application.

Creating a New Project

In this book, we will use a build tool called Gradle to compile and run our Kafka Streams applications. Other build tools (e.g., Maven) are also supported, but we have chosen Gradle due to the improved readability of its build files.

In addition to being able to compile and run your code, Gradle can also be used to quickly bootstrap new Kafka Streams applications that you build outside of this book. This can be accomplished by creating a directory for your project to live in and then by running the `gradle init` command from within that directory. An example of this workflow is as follows:

```
$ mkdir my-project && cd my-project

$ gradle init \
 --type java-application \
 --dsl groovy \
 --test-framework junit-jupiter \
 --project-name my-project \
 --package com.example
```

The source code for this book already contains the initialized project structure for each tutorial, so it's not necessary to run `gradle init` unless you are starting a new project

for yourself. We simply mention it here with the assumption that you will be writing your own Kafka Streams applications at some point, and want a quick way tobootstrap your next project.

Here is the basic project structure for a Kafka Streams application:

```
.
├── build.gradle ❶
└── src
    ├── main
    │   ├── java ❷
    │   └── resources ❸
    └── test
        └── java ❹
```

❶ This is the project's build file. It will specify all of the dependencies (including the Kafka Streams library) needed by our application.

❷ We will save our source code and topology definitions in *src/main/java*.

❸ *src/main/resources* is typically used for storing configuration files.

❹ Our unit and topology tests will live in *src/test/java*.

Now that we've learned how to bootstrap new Kafka Streams projects and have had an initial look at the project structure, let's take a look at how to add Kafka Streams to our project.

Adding the Kafka Streams Dependency

To start working with Kafka Streams, we simply need to add the Kafka Streams library as a dependency in our build file. (In Gradle projects, our build file is called *build.gradle*.) An example build file is shown here:

```
plugins {
    id 'java'
    id 'application'
}

repositories {
```

```
    jcenter()
}

dependencies {
    implementation 'org.apache.kafka:kafka-streams:2.7.0' ❶
}

task runDSL(type: JavaExec) { ❷
    main = 'com.example.DslExample'
    classpath sourceSets.main.runtimeClasspath
}

task runProcessorAPI(type: JavaExec) { ❸
    main = 'com.example.ProcessorApiExample'
    classpath sourceSets.main.runtimeClasspath
}
```

❶ Add the Kafka Streams dependency to our project.

❷ This tutorial is unique among others in this book since we will be creating two different versions of our topology. This line adds a Gradle task to execute the DSL version of our application.

❸ Similarly, this line adds a Gradle task to execute the Processor API version of our application.

Now, to build our project (which will actually pull the dependency from the remote repository into our project), we can run the following command:

```
./gradlew build
```

That's it! Kafka Streams is installed and ready to use. Now, let's continue with the tutorial.

DSL

The DSL example is exceptionally simple. We first need to use a Kafka Streams class called `StreamsBuilder` to build our processor topology:

```
StreamsBuilder builder = new StreamsBuilder();
```

Next, we need to add a source processor in order to read data from a Kafka topic (in this case, our topic will be called `users`). There are a few different methods we could use here depending onhow we decide to model our data, but for now, let's model our data as a stream. The followingline adds the source processor:

```
KStream<Void, String> stream = builder.stream("users");❶
```

❶ The generics in `KStream<Void, String>` refer to the key and value types. In this case, the key is empty (`Void`) andthe value is a `String` type.

Now, it's time to add a stream processor. Since we're just printing a simple greeting for each message, we can use the `foreach` operator with a simple lambda like so:

```
stream.foreach(
    (key, value) -> {
        System.out.println("(DSL) Hello, " + value);
    });
```

Finally, it's time to build our topology and start running our stream processing application:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

The full code, including some boilerplate needed to run the program, is shown in Example 2-1.

Example 2-1. Hello, world—DSL example

```
class DslExample {

  public static void main(String[] args) {
    StreamsBuilder builder = new StreamsBuilder(); ❶

    KStream<Void, String> stream = builder.stream("users"); ❷

    stream.foreach( ❸
        (key, value) -> {
          System.out.println("(DSL) Hello, " + value);
        });

    // omitted for brevity
    Properties config = ...; ❹

    KafkaStreams streams = new KafkaStreams(builder.build(), config); ❺
    streams.start();

    // close Kafka Streams when the JVM shuts down (e.g., SIGTERM)
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❻
  }
}
```

❶ The builder is used to construct the topology.

❷ Add a source processor that reads from the `users` topic.

❸ Use the DSL's `foreach` operator to print a simple message. The DSL includes many operators.

❹ We have omitted the Kafka Streams configuration for brevity, but will discuss this in upcoming chapters. Among other things, this configuration allows us to specify which Kafka cluster our application should read from and what consumer group this application belongs to.

❺ Build the topology and start streaming.

❻ Close Kafka Streams when the JVM shuts down.

To run the application, simply execute the following command:

```
./gradlew runDSL --info
```

Now your Kafka Streams application is running and listening for incoming data. We can produce some data to our Kafka cluster using the `kafka-console-producer` console script. To do this, run the following commands:

```
docker-compose exec kafka bash ❶

kafka-console-producer \ ❷
    --bootstrap-server localhost:9092 \
    --topic users
```

❶ The console scripts are available in the `kafka` container, which is running the broker in our development cluster. You can also download these scripts as part of the official Kafka distribution.

❷ Start a local producer that will write data to the `users` topic.

Once you are in the producer prompt, create one or more records by typing the name of the user, followed by the Enter key. When you are finished, press Control-C on your keyboard to exit the prompt:

```
>angie
>guy
>kate
>mark
```

Your Kafka Streams application should emit the following greetings:

```
(DSL) Hello, angie
(DSL) Hello, guy
(DSL) Hello, kate
(DSL) Hello, mark
```

We have now verified that our application is working as expected. We will explore some more interesting use cases over the next several chapters, but this process of defining a topology and running our application is a foundation we can build upon. Next, let's look at how to create the same Kafka Streams topology with the lower-level Processor API.

Processor API

The Processor API lacks some of the abstractions available in the high-level DSL, and its syntax is more of a direct reminder that we're building processor topologies, with methods like `Topology.addSource`, `Topology.addProcessor`, and `Topology.addSink` (the latter of which is not used in this example). The first step in using the processor topology is to instantiate a new `Topology` instance, like so:

```
Topology topology = new Topology();
```

Next, we will create a source processor to read data from the `users` topic, and a stream processor to print a simple greeting. The stream processor references a class called `SayHelloProcessor` that we'll implement shortly:

```
topology.addSource("UserSource", "users"); ❶
topology.addProcessor("SayHello", SayHelloProcessor::new, "UserSource"); ❷
```

❶ The first argument for the `addSource` method is an arbitrary name for thisstream processor. In this case, we simply call this processor `UserSource`. We willrefer to this name in the next line when we want to connect a child processor, which in turn defines how data should flow through our topology. The second argument is the topic name that this source processor should read from (in this case, `users`).

❷ This line creates a new downstream processor called `SayHello` whose processing logic is defined in the `SayHelloProcessor` class (we will create this in the next section). In the Processor API, we can connect one processor to another by specifying the name of the parent processor. In this case, we specify the `UserSource` processor as the parent of the `SayHello` processor, which means data will flow from the `UserSource` to `SayHello`.

As we saw before, in the DSL tutorial, we now need to build the topology and call `streams.start()` to run it:

```
KafkaStreams streams = new KafkaStreams(topology, config);
streams.start();
```

Before running the code, we need to implement the `SayHelloProcessor` class. Whenever you build a custom stream processor using the Processor API, you need to implement the `Processor` interface. The interface specifies methods for initializing the stream processor (`init`), applying the stream processing logic to a single record (`process`), and a cleanup function (`close`). The initialization and cleanup function aren't needed in this example.

The following is a simple implementation of `SayHelloProcessor` that we will use for this example.

```
public class SayHelloProcessor implements Processor<Void, String, Void, Void> { ❶
  @Override
  public void init(ProcessorContext<Void, Void> context) {} ❷

  @Override
  public void process(Record<Void, String> record) { ❸
    System.out.println("(Processor API) Hello, " + record.value());
  }

  @Override
  public void close() {} ❹
}
```

❶ The first two generics in the `Processor` interface (in this example, `Processor<Void, String, ..., ...>`) refer to the *input* key and value types. Since our keys are null and our values are usernames (i.e., text strings), `Void` and `String` are the appropriate choices. The last two generics (`Processor<..., ..., Void, Void>`) refer to the *output* key and value types. Inthis example, our `SayHelloProcessor` simply prints a greeting. Since we aren't forwarding any output keys or values downstream, `Void` is the appropriate type for the final two generics.

❷ No special initialization is needed in this example, so the method body is empty. The generics in the `ProcessorContext` interface (`ProcessorContext<Void, Void>`) refer to the output key and value types (again, as we're not forwarding any messages downstream in this example, both are `Void`).

❸ The processing logic lives in the aptly named `process` method in the `Processor` interface. Here, we print a simple greeting. Note that the generics in the `Record` interface refer to the key and value type of the *input* records.

❹ No special cleanup needed in this example.

We can now run the code using the same command we used in the DSL example:

```
./gradlew runProcessorAPI --info
```

You should see the following output to indicate your Kafka Streams application is working as expected:

```
(Processor API) Hello, angie
(Processor API) Hello, guy
(Processor API) Hello, kate
(Processor API) Hello, mark
```

Now, despite the Processor API's power, using the DSL is often preferable because, among other benefits, it includes two very powerful abstractions: streams and tables.

Streams and Tables

If you look closely at Example 2-1, you will notice that we used a DSL operator called `stream` to read a Kafka topic into a *stream*. The relevant line of code is:

```
KStream<Void, String> stream = builder.stream("users");
```

However, kafka streams also supports an additional way to view our data: as a *table*.

Designing a processor topology involves specifying a set of source and sink processors, which correspond to the topics your application will read from and write to. However, instead of working with Kafka topics *directly*, the Kafka Streams DSL allows you to work with different *representations* of a topic, each of which are suitable for different use cases. There are twoways to model the data in your Kafka topics: as a *stream* (also called a *record stream*)or a *table* (also known as a *changelog stream*). The easiest way to compare these twodata models is through an example.

Say we have a topic containing ssh logs, where each record is keyed by a user ID as shown in Table 2-2.

Table 2-2. Keyed records in a single topic-partition

Key	Value	Offset
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

Before consuming this data, we need to decide which abstraction to use: a stream or a table. When making this decision, we need to consider whether or not we want to track

only the latest state/representation of a given key, or the entire history of messages. Let's compare the two options side by side:

Streams

These can be thought of as inserts in database parlance. Each distinct record remains in this view of the log. The stream representation of our topic can be seen in Table 2-3.

Table 2-3. Stream view of ssh logs

Key	Value	Offset
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

Tables

Tables can be thought of as updates to a database. In this view of the logs, only the current state (either the latest record for a given key or some kind of aggregation) for each key is retained. Tables are usually built from *compacted topics* (i.e., topics that are configured with a `cleanup.policy` of `compact`, which tells Kafka that you only want to keep the latest representation of each key). The table representation of our topic can be seen in Table 2-4.

Table 2-4. Table view of ssh logs

Key	Value	Offset
mitch	`{ "action": "logout" }`	1
elyse	`{ "action": "login" }`	2
isabelle	`{ "action": "login" }`	3

Tables, by nature, are *stateful*, and are often used for performing aggregations in Kafka Streams. In Table 2-4, we didn't really perform a mathematical aggregation, we just kept the latest ssh event for each user ID. However, tables also support mathmatical aggregations. For example, instead of tracking the latest record for each key, we could have just as easily calculated a rolling `count`. In this case, we would have ended up with a slightly different table, where the values contain the result of our `count` aggregation. You can see a count-aggregated table in Table 2-5.

Table 2-5. Aggregated table view of ssh logs

Key	Value	Offset
mitch	2	1
elyse	1	2
isabelle	1	3

Careful readers may have noticed a discrepancy between the design of Kafka's storage layer (a distributed, append-only log) and a table. Records that are written to Kafka are immutable, so how is it possible to model data as updates, using a *table* representation of a Kafka topic?

The answer is simple: the table is materialized on the Kafka Streams side using a key-value store which, by default, is implemented using RocksDB. By consuming an ordered stream of events and keeping only the latest record for each key in the client-side key-value store (more commonly called a *state store* in Kafka Streams terminology), we end up with a table or map-like representation of the data. In other words, the table isn't something we *consume* from Kafka, but something we *build* on the client side.

You can actually write a few lines of Java code to implement this basic idea. In the following code snippet, the `List` represents a stream since it contains an ordered collection of records, and the table is constructed by iterating through the list (`stream.forEach`) and only retaining the latest record for a given key using a `Map`. The following Java code demonstrates this basic idea:

```
import java.util.Map.Entry;

var stream = List.of(
    Map.entry("a", 1),
    Map.entry("b", 1),
    Map.entry("a", 2));

var table = new HashMap<>();

stream.forEach((record) -> table.put(record.getKey(), record.getValue()));
```

If you were to print the stream and table after running this code, you would see the following output:

```
stream ==> [a=1, b=1, a=2]

table ==> {a=2, b=1}
```

Of course, the Kafka Streams implementation of this is more sophisticated, and can leverage fault-tolerant data structures as opposed to an in-memory `Map`. But this ability to construct a table representation of an unbounded stream is only one side of a more complex relationship between streams and tables.

Stream/Table Duality

The *duality* of tables and streams comes from the fact that tables can be represented as streams, and streams can be used to reconstruct tables. We saw the latter transformation of a stream into a table in the previous section, when discussing the discrepancy between Kafka's append-only, immutable log and the notion of a mutable table structure that accepts updates to its data.

This ability to reconstruct tables from streams isn't unique to Kafka Streams, and is in fact pretty common in various types of storage. For example, MySQL's replication process relies on the same notion of taking a stream of events (i.e., row changes) to reconstruct a source table on a downstream replica. Similarly, Redis has the notion of an append-only file (AOF) that captures every command that is written to the in-memory key-value store. If a Redis server goes offline, then the stream of commands in the AOF can be replayed to reconstruct the dataset.

What about the other side of the coin (representing a table as a stream)? When viewing a table, you are viewing a single point-in-time representation of a stream. As we saw earlier, tables can be updated when a new record arrives. By changing our view of the table to a stream, we can simply process the update as an insert, and append the new record to the end of the log instead of updating the key. Again, the intuition behind this can be seen using a few lines of Java code:

```
var stream = table.entrySet().stream().collect(Collectors.toList());

stream.add(Map.entry("a", 3));
```

This time, if you print the contents of the stream, you'll see we're no longer using update semantics, but instead insert semantics:

```
stream ==> [a=2, b=1, a=3]
```

So far, we've been working with the standard libraries in Java to build intuition around streams and tables. However, when working with streams and tables in KafkaStreams, you'll use a set of more specialized abstractions. We'll take a look at these abstractions next.

KStream, KTable, GlobalKTable

One of the benefits of using the high-level DSL over the lower-level Processor API in

Kafka Streams is that the former includes a set of abstractions that make working with streams and tables extremely easy.

The following list includes a high-level overview of each:

`KStream`
: A `KStream` is an abstraction of a partitioned *record stream*, in which data is represented using insert semantics (i.e., each event is considered to be *independent* of other events).

`KTable`
: A `KTable` is an abstraction of a partitioned table (i.e., *changelog stream*), in which data is represented using update semantics (the latest representation of a given key is tracked by the application). Since `KTable`s are partitioned, each Kafka Streams task contains only a subset of the full table.

`GlobalKTable`
: This is similar to a `KTable`, except each `GlobalKTable` contains a complete (i.e., unpartitioned) copy of the underlying data.

Kafka Streams applications can make use of multiple stream/table abstractions, or just one. It's entirely dependent on your use case.