Processor API

Just a few chapters ago, we embarked on our journey to learn about Kafka Streams. We started with Kafka Streams' high-level DSL, which allows us to build stream processing applications using a functional and fluent interface. This involves composing and chaining stream processing functions using the library's built-in operators (e.g., `filter`, `flatMap`, `groupBy`, etc.) and abstractions (`KStream`, `KTable`, `GlobalKTable`).

In this chapter, we will explore a lower-level API that is available in Kafka Streams: the Processor API (sometimes called PAPI). The Processor API has fewer abstractions than the high-level DSL and uses an imperative style of programming. While the code is generally more verbose, it is also more powerful, giving us fine-grained control over the following: how data flows through our topologies, how stream processors relate to each other, how state is created and maintained, and even the timing of certain operations.

Some of the questions we will answer in this chapter include:

- When should you use the Processor API?
- How do you add source, sink, and stream processors using the Processor API?
- How can you schedule periodic functions?
- Is it possible to mix the Processor API with the higher-level DSL?
- What is the difference between processors and transformers?

As usual, we will demonstrate the fundamentals of the API through a tutorial, answering the preceding questions along the way. However, before we show you *how* to use the Processor API, let's first discuss *when* to use it.

When to Use the Processor API

Deciding which abstraction level to use for your stream processing application is important. In general, whenever you introduce complexity into a project, you should have a good reason for doing so. While the Processor API isn't unnecessarily complex, its low-level nature (compared to the DSL and ksqlDB) and fewer abstractions can lead to more code and, if you're not careful, more mistakes.

In general, you may want to utilize the Processor API if you need to take advantage of any of the following:

- Access to record metadata (topic, partition, offset information, record headers, and so on)
- Ability to schedule periodic functions
- More fine-grained control over when records get forwarded to downstream processors
- More granular access to state stores
- Ability to circumvent any limitations you come across in the DSL (we'll see an example of this later)

On the other hand, using the Processor API can come with some disadvantages, including:

- More verbose code, which can lead to higher maintenance costs and impair readability
- A higher barrier to entry for other project maintainers
- More footguns, including accidental reinvention of DSL features or abstractions, exotic problem-framing,[1] and performance traps

Fortunately, Kafka Streams allows us to mix both the DSL and Processor API in an application, so you don't need to go all in on either choice. You can use the DSL for simpler and more standard operations, and the Processor API for more complex or unique functions that require lower-level access to the processing context, state, or record metadata. We will discuss how to combine the DSL and Processor API at the end of this chapter, but initially, we will see how to implement an application using *only* the Processor API. So without further ado, let's take a look at the application we'll be building in this chapter.

Introducing Our Tutorial: IoT Digital Twin Service

In this tutorial, we will use the Processor API to build a *digital twin* service for an offshore wind farm. Digital twins (sometimes called device shadows) are popular in both IoT (Internet of Things) and IIoT (industrial IoT) use cases, in which the state of a physical object is mirrored in a digital copy. This is a great use case for Kafka Streams, which can easily ingest and process high-volume sensor data, capture the state of a physical object using state stores, and subsequently expose this state using interactive queries.

To give you a quick example of what a digital twin is (this will make our tutorial a little clearer), consider the following. We have a wind farm with 40 wind turbines. Whenever one of the turbines reports its current state (wind speed, temperature, power status, etc.), we save that information in a key-value store. An example of a *reported state* record value is shown here:

```
{
  "timestamp": "2020-11-23T09:02:00.000Z",
  "wind_speed_mph": 40,
  "temperature_fahrenheit": 60,
  "power": "ON"
}
```

Note that a device ID is communicated via the record key (e.g., the preceding value may correspond to a device with an ID of `abc123`). This will allow us to distinguish the reported/desired state events of one device from another.

Now, if we want to interact with a particular wind turbine, we don't do so directly. IoT devices can and do frequently go offline, so we can achieve higher availability and reduce errors if we instead only interact with the digital copy (twin) of a physical device.

For example, if we want to set the power state from `ON` to `OFF`, instead of sending that signal to the turbine directly, we would set the so-called *desired state* on the digital copy. The physical turbine would subsequently synchronize its state (i.e., disable power to the blades) whenever it comes online, and usually at set intervals, thereafter. Therefore, a digital twin record will include both a *reported* and *desired* state, and we will create and expose digital twin records like the following using Kafka Streams' Processor API:

```
{
  "desired": {
    "timestamp": "2020-11-23T09:02:01.000Z",
    "power": "OFF"
  },
  "reported": {
    "timestamp": "2020-11-23T09:00:01.000Z",
    "windSpeedMph": 68,
    "power": "ON"
  }
}
```

With this in mind, our application needs to ingest a stream of sensor data from a set of wind turbines, perform some minor processing on the data, and maintain the latest state of each wind turbine in a persistent key-value state store. We will then expose the data via Kafka Streams' interactive queries feature.

Though we will avoid much of the technical detail around interactive queries since this has already been covered in previous chapters, we will present some additional value statements of interactive queries that IoT use cases afford.

Figure 7-1 shows the topology we will be building in this chapter. Each step is detailed after the diagram.

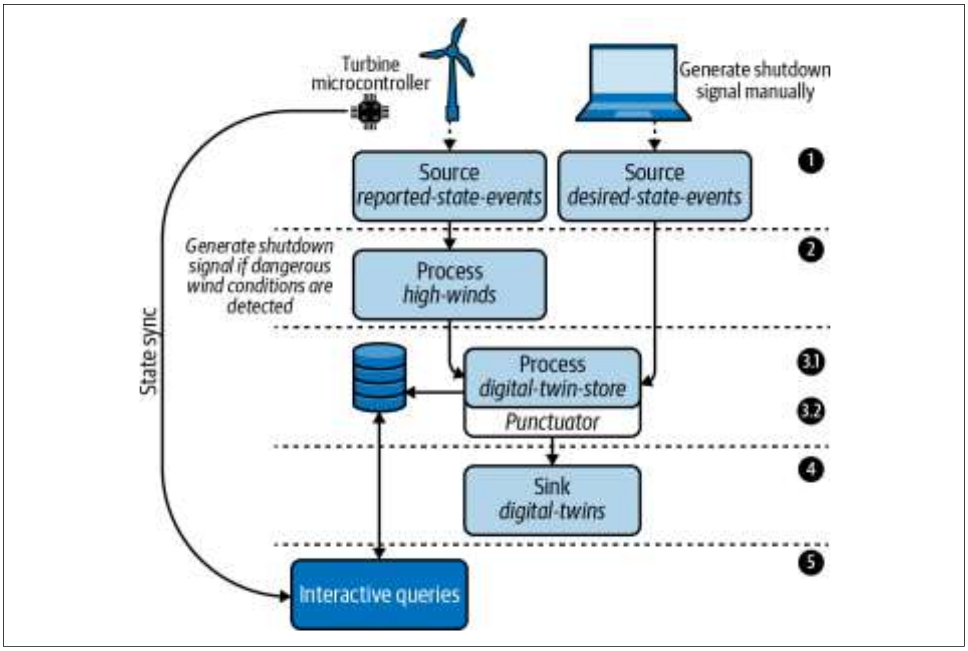
Turbine microcontroller
Generate shutdown signal manually
Source reported-state-events
Source desired-state-events
1
Generate shutdown signal if dangerous wind conditions are detected
Process high-winds
2
State sync
Process digital-twin-store
Punctuator
3.1
3.2
Sink digital-twins
4
Interactive queries
5

Figure 7-1. The topology that we will be implementing for our IoT digital twin service

❶ Our Kafka cluster contains two topics, and therefore we need to learn how to add source processors using the Processor API. Here is a description of these topics:

- Each wind turbine (edge node) is outfitted with a set of environmental sensors, and this data (e.g., wind speed), along with some metadata about the turbine itself (e.g., power state), is sent to the `reported-state-events` topic periodically.
- The `desired-state-events` topic is written to whenever a user or process wants to change the power state of a turbine (i.e., turn it off or on).

❷ Since the environmental sensor data is reported in the `reported-state-events` topic, we will add a stream processor that determines whether or not the reported wind speed for a given turbine exceeds safe operating levels,[6] and if it does, we will automatically generate a shutdown signal. This will teach you how to add a stateless stream processor using the Processor API.

❸ The third step is broken into two parts:

- First, both types of events (reported and desired) will be combined into a so-called digital twin record. These records will be processed and then written to a persistent key-value store called `digital-twin-store`. In this step, you will learn how to connect to and interact with state stores using the Processor API, and also how to access certain record metadata that isn't accessible via the DSL.
- The second part of this step involves scheduling a periodic function, called a *punctuator*, to clean out old digital twin records that haven't seen an update in more than seven days. This will introduce you to the Processor API's punctuation interface, and also demonstrate an alternative method for removing keys from state stores.[7]

❹ Each digital twin record will be written to an output topic called `digital-twins` for analytical purposes. In this step, you will learn how to add sink processors using the Processor API.

❺ We will expose the digital twin records via Kafka Streams' interactive queries feature. Every few seconds, the microcontroller on the wind turbine will attempt to synchronize its own state with the desired state exposed by Kafka Streams. For example, if we generate a shutdown signal in step 2 (which would set the desired power state to `OFF`), then the turbine would see this desired state when it queries our Kafka Streams app, and kill power to the blades.

Now that we understand what we'll be building (and what we'll learn at each step), let's set up our project.

Project Setup

The code for this chapter is located at *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git*.

If you would like to reference the code as we work our way through each topology step, clone the repo and change to the directory containing this chapter's tutorial. The following command will do the trick:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-07/digital-twin
```

You can build the project anytime by running the following command:

```
$ ./gradlew build --info
```

With our project set up, let's start implementing our digital twin application.

Data Models

As usual, before we start creating our topology, we’ll first define our data models. The example records and class definition shown in Table 7-1 correspond to the data in our input topics (see step 1 in our processor topology: Figure 7-1).

Note that the data coming through both of the input topics is formatted as JSON for simplicity, and both types of records are represented using a common class: `TurbineState`. We have omitted the accessor functions in the `TurbineState` class for brevity’s sake.

Table 7-1. Example records and data classes for each source topic

Kafka topic	Example record	Data class

`reported-state-events`	`{`
	`  "timestamp": "...",`
	`  "wind_speed_mph": 40,`
	`  "power": "ON"`
	`}`
		`public class TurbineState {`
		`  private String timestamp;`
		`  private Double windSpeedMph;`
		
		`  public enum Power { ON, OFF }`
		
		`  public enum Type { DESIRED, REPORTED }`
		
		`  private Power power;`
		`  private Type type;`
		`}`

`desired-state-events`	`{`
	`  "timestamp": "...",`
	`  "power": "OFF"`
	`}`
		Same as the data class for `reported-state`

As mentioned in the tutorial overview, we need to combine the reported and desired state records to create a digital twin record. Therefore, we also need a data class for the combined record. The following table shows the JSON structure of the combined digital twin record, as well as the corresponding data class:

Example record

```
{
  "desired": {
    "timestamp":  "2020-11-23T09:02:01.000Z",
    "power": "OFF"
  },
  "reported": {
    "timestamp":  "2020-11-23T09:00:01.000Z",
    "windSpeedMph": 68,
    "power": "ON"
```

Data class

```
public class DigitalTwin {
  private TurbineState desired;
  private TurbineState reported;

  // getters and setters omitted for
  // brevity
}
```

```
  }
}
```

As you can see from the example record, the desired state shows the turbine powered off, but the last reported state shows that the power is on. The turbine will eventually synchronize its state with the digital twin and power the blades off.

Now, at this point, you may be wondering how record serialization and deserialization work in the Processor API compared to the high-level DSL. In other words, how do we actually convert the raw record bytes in our Kafka topics into the data classes shown in Table 7-1? We talked about using Serdes classes in the DSL, which are wrapper classes that contain both a serializer and deserializer. Many DSL operators, like `stream`, `table`, `join`, etc., allow you to specify a Serdes instance, so they are commonplace in applications that use the DSL.

In the Processor API, the various API methods only require the underlying serializer or deserializer that a Serdes instance would typically contain. However, it is still often convenient to define a Serdes for your data classes, since 1) you can always extract the underlying serializer/deserializer to satisfy a Processor API method signature, and 2) Serdes are often useful for testing purposes.

With that said, for this tutorial, we will leverage the Serdes classes shown in Example 7-1.

Example 7-1. Serdes for our digital twin and turbine state records

```
public class JsonSerdes {

  public static Serde<DigitalTwin> DigitalTwin() { ❶
    JsonSerializer<DigitalTwin> serializer = new JsonSerializer<>();
    JsonDeserializer<DigitalTwin> deserializer =
      new JsonDeserializer<>(DigitalTwin.class);

    return Serdes.serdeFrom(serializer, deserializer);
  }

  public static Serde<TurbineState> TurbineState() { ❷
    JsonSerializer<TurbineState> serializer = new JsonSerializer<>();
    JsonDeserializer<TurbineState> deserializer =

      new JsonDeserializer<>(TurbineState.class);

    return Serdes.serdeFrom(serializer, deserializer); ❸
  }
}
```

❶ A method for retrieving a `DigitalTwin` Serdes.

❷ A method for retrieving a `TurbineState` Serdes.

❸ In previous tutorials, we implemented the `Serde` interface directly. This shows an alternative approach, which is to use the `Serdes.serdeFrom` method in Kafka Streams to construct a Serdes from a serializer and deserializer instance.

In the next section, we'll learn how to add source processors and deserialize input records using the Processor API.

Adding Source Processors

Now that we have defined our data classes, we are ready to tackle step 1 of our processor topology (see Figure 7-1). This involves adding two source processors, which will allow us to stream data from our input topics into our Kafka Streams application. Example 7-2 shows how we can accomplish this using the Processor API.

Example 7-2. The initial topology with both of our source processors added

```
Topology builder = new Topology(); ❶

builder.addSource(❷
    "Desired State Events", ❸
    Serdes.String().deserializer(), ❹
    JsonSerdes.TurbineState().deserializer(), ❺
    "desired-state-events"); ❻

builder.addSource(❼
    "Reported State Events",
    Serdes.String().deserializer(),
    JsonSerdes.TurbineState().deserializer(),
    "reported-state-events");
```

❶ Instantiate a `Topology` instance directly. This is what we will use to add and connect source, sink, and stream processors. Note: instantiating a `Topology` directly is different than how we work with the DSL, which requires us to instantiate a `StreamsBuilder` object, add our DSL operators (e.g., `map`, `flatMap`, `merge`, `branch`, etc.) to the `StreamsBuilder` instance, and ultimately build a `Topology` instance using the `StreamsBuilder#build` method.

❷ Use the `addSource` method to create a source processor. There are many overloaded versions of this method, including variations that support offset reset strategies, topic patterns, and more. So check out the Kafka Streams Javadocs or navigate to the Topology class using your IDE and choose the `addSource` variation that best fits your needs.

❸ The name of the source processor. Each processor must have a unique name, since under the hood, Kafka Streams stores these names in a topologically sortedmap (and therefore, each key must be unique). As we'll see shortly, names are important in the Processor API since they are used to connect child processors.

Again, this is very different from how the DSL makes connections, which doesn't need an explicit name to define the relationship between processors (by default, the DSL generates an internal name for you). It's advisable to use a descriptive name here to improve the readability of your code.

4. The key deserializer. Here, we use the built-in `String` deserializer since our keys are formatted as strings. This is another difference between the Processor API and the DSL. The latter requires us to pass in a Serdes (an object that contains both a record serializer and deserializer), while the Processor API only requires the underlying deserializer (which can be extracted directly from the Serdes, as we have done here).

5. The value deserializer. Here, we use a custom Serdes (found in the source code of this tutorial) to convert the record values into a `TurbineState` object. The additional notes about the key deserializer also apply here.

6. The name of the topic this source processor consumes from.

7. Add a second source processor for the `reported-state-events` topic. We won't go through each parameter again since the parameter types match the previous source processor.

One thing to note about the preceding example is that you will see no mention of a stream or table. These abstractions do not exist in the Processor API. However, conceptually speaking, both source processors we have added in the preceding code represent a stream. This is because the processors are not stateful (i.e., they are not connected to a state store) and therefore have no way of remembering the latest state/representation of a given key.

We will see a table-like representation of our stream when we get to step 3 of our processor topology, but this section completes step 1 of our topology. Now, let's see how to add a stream processor for generating shutdown signals when our wind turbine is reporting dangerous wind speeds.

Adding Stateless Stream Processors

The next step in our processor topology requires us to automatically generate a shutdown signal whenever the wind speed recorded by a given turbine exceeds safe operating levels (65 mph). In order to do this, we need to learn how to add a stream processor using the Processor API. The API method we can use for this purpose is called `addProcessor`, and an example of how to use this method is shown here:

```
builder.addProcessor(
    "High Winds Flatmap Processor", (1)
    HighWindsFlatmapProcessor::new, (2)
    "Reported State Events"); (3)
```

❶ The name of this stream processor.

❷ The second argument expects us to provide a `ProcessSupplier`, which is a functional interface that returns a `Processor` instance. `Processor` instances contain all of the data processing/transformation logic for a given stream processor. In the next section, we will define a class called `HighWindsFlatmapProcessor`, which will implement the `Processor` interface. Therefore, we can simply use a method reference for that class's constructor.

❸ Names of the parent processors. In this case, we only have one parent processor, which is the `Reported State Events` processor that we created in Example 7-2. Stream processors can be connected to one or more parent nodes.

Whenever you add a stream processor, you need to implement the `Processor` interface. This isn't a functional interface (unlike `ProcessSupplier`, which we just discussed), so you can't just pass in a lambda expression to the `addProcessor` method, like we often do with DSL operators. It's a little more involved and requires more code than we may be accustomed to, but we'll walk through how to do this in the nextsection.

Creating Stateless Processors

Whenever we use the `addProcessor` method in the Processor API, we need to implement the `Processor` interface, which will contain the logic for processing and transforming records in a stream. The interface has three methods, as shown here:

```
public interface Processor<K, V> { ❶

    void init(ProcessorContext context); ❷

    void process(K key, V value); ❸

    void close(); ❹
}
```

❶ Notice that the `Processor` interface specifies two generics: one for the key type (`K`) and one for the value type (`V`). We will see how to leverage these generics when we implement our own processor later in this section.

❷ The `init` method is called when the `Processor` is first instantiated. If your processor needs to perform any initialization tasks, you can specify the initialization logic in this method. The `ProcessorContext` that is passed to the `init` method is extremely useful, and contains many methods that we will explore in this chapter.

❸ The `process` method is called whenever this processor receives a new record. It contains the per-record data transformation/processing logic. In our example, this is where we will add the logic for detecting whether or not wind speeds exceed safe operating levels for our turbine.

4. The `close` method is invoked by Kafka Streams whenever it is finished with this operator (e.g., during shutdown). This method typically encapsulates any clean up logic you need for your processor and its local resources. However, you should not attempt to cleanup any Kafka Streams–managed resources, like state stores, in this method, since that is handled by the library itself.

With this interface in mind, let's implement a `Processor` that will generate a shutdown signal when wind speeds reach dangerous levels. The code in Example 7-3 shows what our high winds processor looks like.

Example 7-3. A Processor implementation that detects dangerous wind speeds

```
public class HighWindsFlatmapProcessor
    implements Processor<String, TurbineState, String, TurbineState> { (1)
  private ProcessorContext<String, TurbineState> context;

  @Override
  public void init(ProcessorContext<String, TurbineState> context) { (2)
    this.context = context; (3)
  }

  @Override
  public void process(Record<String, TurbineState> record) {
    TurbineState reported = record.value();
    context.forward(record); (4)

    if (reported.getWindSpeedMph() > 65 && reported.getPower() == Power.ON) { (5)
      TurbineState desired = TurbineState.clone(reported); (6)
      desired.setPower(Power.OFF);
      desired.setType(Type.DESIRED);

      Record<String, TurbineState> newRecord = (7)
        new Record<>(record.key(), desired, record.timestamp());
      context.forward(newRecord); (8)
    }
  }

  @Override
  public void close() {
    // nothing to do (9)
  }
}
```

1. Recall that the `Processor` interface is parameterized with four generics. The first two generics (in this case, `Processor<String, TurbineState, ..., ...>`) refer to the *input* key and value types. The last two generics (in this case, `Processor<..., ..., String, TurbineState>`) refer to the *output* key and value types.

2. The generics in the `ProcessorContext` interface refer to the output key and value

types (in this case, `ProcessorContext<String, TurbineState>`).

❸ It is typical to save the processor context as an instance property, as we do here, so that we can access it later (e.g., from the `process` and/or `close` methods).

❹ Whenever you want to send a record to downstream processors, you can call the `forward` method on the `ProcessorContext` instance (we have saved this in the `context` property). This method accepts the record that you would like to forward. In our processor implementation, we always want to forward the reported state records, which is why we call `context.forward` using an unmodified record in this line.

❺ Check to see if our turbine meets the conditions for sending a shutdown signal. In this case, we check that wind speeds exceed a safe threshold (65 mph), and that our turbine is currently powered on.

❻ If the previous conditions are met, generate a new record containing a desired power state of `OFF`. Since we have already sent the original reported state record downstream, and we are now generating a desired state record, this is effectively a type of `flatMap` operation (our processor has created two output records from one input record).

❼ Create an output record that includes the desired state that we saved to the state store. The record key and timestamp are inherited from the input record.

❽ Call the `context.forward` method in order to send the new record (the shut- down signal) to downstream processors.

❾ In this processor, there's no special logic that we need to execute when our processor is closed.

As you can see, implementing a `Processor` is pretty straightforward. One interesting thing to call out is that when you build processors like this, you usually don't need to concern yourself with *where* the output records will be forwarded to from the `Processor` implementation itself (you can define the dataflow by setting the parent name of a downstream processor). The exception to this is if you use a variation of `ProcessorContext#forward` that accepts a list of downstream processor names, which tells Kafka Streams which *child processors* to forward the output to. An exam-ple of this is shown here:

```
context.forward(newRecord, "some-child-node");
```

Whether or not you use this variation of the `forward` method depends on if you want to broadcast the output to all of the downstream processors or to a specific downstream processor. For example, the DSL's branch method uses the above variation since it only needs to broadcast its output to a subset of the available downstream processors.

This completes step 2 of our processor topology (see Figure 7-1). Next, we need to implement a stateful stream processor that creates and saves digital twin records in a key-value store.

Creating Stateful Processors

Stateful operations in Kafka Streams require so-called state stores for maintaining some memory of previously seen data. In order to create our digital twin records, we need to combine desired state and recorded state events into a single record. Since these records will arrive at different times for a given wind turbine, we have a stateful requirement of remembering the last recorded and desired state record for each turbine.

So far in this book, we have mostly focused on using state stores in the DSL. Furthermore, the DSL gives us a couple of different options for using state stores. We can use a default internal state store by simply using a stateful operator without specifying a state store, like so:

```
grouped.aggregate(initializer, adder);
```

Or we can use the `Stores` factory class to create a *store supplier*, and materialize the state store using the `Materialized` class in conjunction with a stateful operator, as shown in the following code:

```
KeyValueBytesStoreSupplier storeSupplier =
    Stores.persistentTimestampedKeyValueStore("my-store");

grouped.aggregate(
    initializer,
    adder,
    Materialized.<String, String>as(storeSupplier));
```

Using state stores in the Processor API is a little different. Unlike the DSL, the Processor API won't create an internal state store for you. Therefore, you must always create *and* connect state stores to the appropriate stream processors yourself when you need to perform stateful operations. Furthermore, while we can still use the `Stores` factory class, we will use a different set of methods that are available in thisclass for creating state stores. Instead of using one of the methods that returns a *store supplier*, we will use the methods for creating *store builders*.

For example, to store the digital twin records, we need a simple key-value store. The factory method for retrieving a key-value store *builder* is called `keyValueStoreBuilder`, and the following code demonstrates how we can use this method for creating our digital twin store:

```
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("digital-twin-store"),
        Serdes.String(), ❶
        JsonSerdes.DigitalTwin()); ❷
```

❶ We'll use the built-in String Serdes for serializing/deserializing keys.

❷ Use the Serdes we defined in Example 7-1 for serializing/deserializing values.

Once you've created a store builder for your stateful processor, it's time to implement the `Processor` interface. We simply need to use the `addProcessor` method in the Processor API, as shown in the following code:

```
builder.addProcessor(
  "Digital Twin Processor", ❶
  DigitalTwinProcessor::new, ❷
  "High Winds Flatmap Processor", "Desired State Events"); ❸
```

❶ The name of this stream processor.

❷ A `ProcessSupplier`, which is a method that can be used to retrieve a `Processor` instance. We will implement the `DigitalTwinProcessor` referenced in this line shortly.

❸ Names of the parent processors. By specifying multiple parents, we are effectively performing what would be a `merge` operation in the DSL.

Before we implement the `DigitalTwinProcessor`, let's go ahead and add our new state store to the topology. We can do this using the `Topology#addStateStore`method, and a demonstration of its usage is shown in Example 7-4.

Example 7-4. Example usage of the addStateStore *method*

```
builder.addStateStore(
  storeBuilder, ❶
  "Digital Twin Processor" ❷
);
```

❶ A store builder that can be used to obtain the state store.

❷ We can *optionally* pass in the name of the processors that should have access to this store. In this case, the `DigitalTwinProcessor` that we created in the previous code block should have access. We could also have passed in more processor names here if we had multiple processors with shared state. Finally, if we omit this optional argument, we could instead use the `Topology#connectProcessorAndState` to connect a state store to a processor *after* the store was added to the topology (instead of at the same time, which is what we're doing here).

The last step is to implement our new stateful processor: `DigitalTwinProcessor`. Like stateless stream processors, we will need to implement the `Processor` interface. However, this time, our implementation will be slightly more involved since this processor needs to interact with a state store. The code in Example 7-5, and the anno-tations that follow it, will describe how to implement a stateful processor.

Example 7-5. A stateful processor for creating digital twin records

```
public class DigitalTwinProcessor
    implements Processor<String, TurbineState, String, DigitalTwin> { ❶
  private ProcessorContext<String, DigitalTwin> context;
  private KeyValueStore<String, DigitalTwin> kvStore;

  @Override
  public void init(ProcessorContext<String, DigitalTwin> context) { ❷
    this.context = context; ❸
    this.kvStore = (KeyValueStore) context.getStateStore("digital-twin-store"); ❹
  }

  @Override
  public void process(Record<String, TurbineState> record) {
    String key = record.key(); ❺
    TurbineState value = record.value();
    DigitalTwin digitalTwin = kvStore.get(key); ❻
    if (digitalTwin == null) { ❼
      digitalTwin = new DigitalTwin();
    }

    if (value.getType() == Type.DESIRED) { ❽
      digitalTwin.setDesired(value);
    } else if (value.getType() == Type.REPORTED) {
      digitalTwin.setReported(value);
    }

    kvStore.put(key, digitalTwin); ❾

    Record<String, DigitalTwin> newRecord =
      new Record<>(record.key(), digitalTwin, record.timestamp()); ❿
    context.forward(newRecord); ⓫
  }

  @Override
  public void close() {
    // nothing to do
  }
}
```

❶ The first two generics in the `Processor` interface (in this case, `Processor<String, TurbineState, ..., ...>`) refer to the *input* key and value types, and the last two generics (`Processor<..., ..., String, DigitalTwin>`) refer to the *output* key and value types.

❷ The generics in the `ProcessorContext` interface (`ProcessorContext<String, DigitalTwin>`) refer to the output key and value types

❸ We'll save the `ProcessorContext` (referenced by the `context` property) so that we can access it later on.

❹ The `getStateStore` method on the `ProcessorContext` allows us to retrieve the state store we previously attached to our stream processor. We will interact with this state store directly whenever a record is processed, so we will save it to an instance property named `kvStore`.

❺ This line and the one that follows it show how to extract the key and value of the input record

❻ Use our key-value store to perform a point lookup for the current record's key. If we have seen this key before, then this will return the previously saved digital twin record.

❼ If the point lookup didn't return any results, then we will create a new digital twin record.

❽ In this code block, we set the appropriate value in the digital twin record depending on the current record's type (reported state or desired state).

❾ Store the digital twin record in the state store directly, using the key-value store's `put` method.

❿ Create an output record that includes the digital twin instance that we saved to the state store. The record key and timestamp are inherited from the inputrecord.

⓫ Forward the output record to downstream processors.

We've now implemented the first part of step 3 in our processor topology. The next step (step 3.2 in Figure 7-1) will introduce you to a very important feature in the Processor API that has no DSL equivalent. Let's take a look at how to schedule periodic functions in the DSL.

Periodic Functions with Punctuate

Depending on your use case, you may need to perform some periodic task in your Kafka Streams application. This is one area where the Processor API really shines, since it allows you to easily schedule a task using the `ProcessorContext#schedule` method. In this tutorial, we will present another method for cleaning out state stores that leverages this task scheduling capability. Here, we will remove all digital twin records that haven't seen any stateupdates in the last seven days. We will assume these turbines are no longer active or are under long-term maintenance, and therefore we'll delete these records from our key-value store.

In Chapter 5, we showed that when it comes to stream processing, *time* is a complex subject. When we think about *when* a periodic function will execute in Kafka Streams, we are reminded of this complexity. There are two *punctuation types* (i.e., timing strategies) that you can select from, as shown in Table 7-2.

Table 7-2. The types of punctuations that are available in Kafka Streams

Punctuation type	Enum	Description
Stream time	`PunctuationType.STREAM_TIME`	Stream time is the highest timestamp observed for a particular topic-partition. It is initially unknown and can only increase or stay the same. It advances only when new data is seen, so if you use this punctuation type, then your function will not execute unless data arrives on a continuous basis.
Wall clock time	`PunctuationType.WALL_CLOCK_TIME`	The local system time, which is advanced during each iteration of the consumer poll method. The upper bound for how often this gets updated is defined by the `StreamsConfig#POLL_MS_CONFIG` configuration, which is the maximum amount of time (in milliseconds) the underlying poll method will block as it waits for new data. This means periodic functions will continue to execute regardless of whether or not new messages arrive.

Since we don't want our periodic function to be tied to new data arriving (in fact, the very presence of this TTL ("time to live") function is based on the assumption that data may stop arriving), then we will use wall clock time as our punctuation type. Now that we've decided which abstraction to use, the rest of the work is simply a matter of scheduling and implementing our TTL function.

The following code shows our implementation:

```
public class DigitalTwinProcessor
    implements Processor<String, TurbineState, String, DigitalTwin> {

  private Cancellable punctuator; ❶

  // other omitted for brevity

  @Override
  public void init(ProcessorContext<String, DigitalTwin> context) {

    punctuator =  this.context.schedule(
        Duration.ofMinutes(5),
        PunctuationType.WALL_CLOCK_TIME, this::enforceTtl); ❷

    // ...
  }

  @Override
  public void close() {
    punctuator.cancel(); ❸
  }

  public void enforceTtl(Long timestamp) {
    try (KeyValueIterator<String, DigitalTwin> iter = kvStore.all()) { ❹

      while (iter.hasNext()) {
        KeyValue<String, DigitalTwin> entry = iter.next();
        TurbineState lastReportedState = entry.value.getReported(); ❺
        if (lastReportedState == null) {
          continue;
        }

        Instant lastUpdated = Instant.parse(lastReportedState.getTimestamp());
        long daysSinceLastUpdate =
          Duration.between(lastUpdated, Instant.now()).toDays(); ❻
        if (daysSinceLastUpdate >= 7) {
          kvStore.delete(entry.key); ❼
        }
      }
    }
  }

  // ...
}
```

❶ When we schedule the punctuator function, it will return a `Cancellable` object that we can use to stop the scheduled function later on. We'll use an object variable named `punctuator` to keep track of this object.

❷ Schedule our periodic function to execute every five minutes based on wall clock time, and save the returned `Cancellable` under the `punctuator` property (see the preceding callout).

❸ Cancel the punctuator when our processor is closed (e.g., during a clean shutdown of our Kafka Streams application).

❹ During each invocation of our function, retrieve each value in our state store. Note that we use a try-with-resources statement to ensure the iterator is closed properly, which will prevent resource leaks.

❺ Extract the last reported state of the current record (which corresponds to a physical wind turbine).

❻ Determine how long it's been (in days) since this turbine last reported its state.

❼ Delete the record from the state store if it's stale (hasn't been updated in at least seven days).

> The `process` function and any punctuations we schedule will be executed in the same thread (i.e., there's no background thread for punctuations) so you don't need to worry about concurrency issues.

As you can see, scheduling periodic functions is extremely easy. Next, let's look at another area where the Processor API shines: accessing record metadata.

Accessing Record Metadata

When we use the DSL, we typically only have access to a record's key and value. However, there's a lot of additional information associated with a given record that is not exposed by the DSL, but that we can access using the Processor API. Some of the more prominent examples of record metadata that you might want to access are shown in Table 7-3. Note that the `context` variable in the following table refers to aninstance of `ProcessorContext`, which is made available in the `init` method, as wefirst showed in Example 7-3.

Table 7-3. Methods for accessing additional record metadata

Metadata	Example
Record headers	`context.headers()`
Offset	`context.offset()`
Partition	`context.partition()`
Timestamp	`context.timestamp()`
Topic	`context.topic()`

The methods shown in Table 7-3 pull metadata about the current record, and can be used within the `process()` function. However, there isn't a current record when the `init()` or `close()` functions are invoked *or* when a punctuation is executing, so there's no metadata to extract.

So what could you do with this metadata? One use case is to decorate the record values with additional context before they are written to some downstream system. You can also decorate application logs with this information to help with debugging purposes. For example, if you encounter a malformed record, you could log an error containing the partition and offset of the record in question, and use that as a basis for further troubleshooting.

Record headers are also interesting, because they can be used to inject additional metadata (for example, tracing context that can be used for distributed tracing). Some examples of how to interact with record headers are shown here:

```
Headers headers = context.headers();
headers.add("hello", "world".getBytes(StandardCharsets.UTF_8)); ❶
headers.remove("goodbye"); ❷
headers.toArray(); ❸
```

❶ Add a header named `hello`. This header will be propagated to downstream processors.

❷ Remove a header named `goodbye`.

❸ Get an array of all of the available headers. You could iterate over this and do something with each.

Finally, if you'd like to trace the origin of a record, the `topic()` method could be useful for this purpose. In this tutorial, we don't need to do this, or really access any metadata at all, but you should now have a good understanding of how to access additional metadata in case you encounter a use case that requires it in the future.

We’re ready to move to the next step of our processor topology and learn how to add a sink processor using the Processor API.

Adding Sink Processors

Tackling step 4 of our processor topology (see Figure 7-1). involves adding a sink processor that will write all digital twin records to an output topic called `digital-twins`. This is very simple with the Processor API, so this section will be short. We simply need to use the `addSink` method and specify a few additional parameters, which are detailed in the following code:

```
builder.addSink(
  "Digital Twin Sink", ❶
  "digital-twins", ❷
  Serdes.String().serializer(), ❸
  JsonSerdes.DigitalTwin().serializer(), ❹
  "Digital Twin Processor"); ❺
```

❶ The name of this sink node.

❷ The name of the output topic.

❸ The key serializer.

❹ The value serializer.

❺ The name of one or more parent nodes to connect to this sink.

That’s all there is to adding a sink processor. Of course, as with most methods in Kafka Streams, there are some additional variations of this method you may want to utilize. For example, one variation allows you to specify a custom `StreamParti tioner` to map the output record to a partition number. Another variation allows you to exclude the key and value serializers and, instead, use the default serializers that arederived from the `DEFAULT_KEY_SERDE_CLASS_CONFIG` property. But no matter which overloaded method you use, adding a sink processor is a pretty simple operation.

Let’s move on to the final step of exposing digital twin records to external services (including the wind turbines themselves, which will synchronize their state to the digital twin records in our state store).

Interactive Queries

We’ve completed steps 1–4 of our topology (see Figure 7-1). The fifth step simply involves exposing the digital twin records using Kafka Streams’ interactive queries feature.

so we won't go into too much detail or show the entire implementation. However, Example 7-6 shows a very simple REST service that uses interactive queries to pull the latest digital twin record. Note that in this example, remote queries aren't shown, but you can view this example's source code for a more complete example.

The important thing to note is that from an interactive query perspective, using the Processor API is exactly the same as using the DSL.

Example 7-6. An example REST service to expose digital twin records

```
class RestService {
  private final HostInfo hostInfo;
  private final KafkaStreams streams;

  RestService(HostInfo hostInfo, KafkaStreams streams) {
    this.hostInfo = hostInfo;
    this.streams = streams;
  }

  ReadOnlyKeyValueStore<String, DigitalTwin> getStore() {
    return streams.store(
        StoreQueryParameters.fromNameAndType(
            "digital-twin-store", QueryableStoreTypes.keyValueStore()));
  }

  void start() {
    Javalin app = Javalin.create().start(hostInfo.port());
    app.get("/devices/:id", this::getDevice);
  }

  void getDevice(Context ctx) {
    String deviceId = ctx.pathParam("id");
    DigitalTwin latestState = getStore().get(deviceId);
    ctx.json(latestState);
  }
}
```

This completes step 5 of our processor topology (see Figure 7-1). Let's put the various pieces that we've constructed together.

Putting It All Together

The following code block shows what our full processor topology looks like at this point:

```
    Topology builder = new Topology();

    builder.addSource( ❶
        "Desired State Events",
        Serdes.String().deserializer(),
        JsonSerdes.TurbineState().deserializer(),
        "desired-state-events");

    builder.addSource( ❷
        "Reported State Events",
        Serdes.String().deserializer(),
        JsonSerdes.TurbineState().deserializer(),
        "reported-state-events");

    builder.addProcessor( ❸
        "High Winds Flatmap Processor",
        HighWindsFlatmapProcessor::new,
        "Reported State Events");

    builder.addProcessor( ❹
        "Digital Twin Processor",
        DigitalTwinProcessor::new,
        "High Winds Flatmap Processor",
        "Desired State Events");

    StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
        Stores.keyValueStoreBuilder( ❺
            Stores.persistentKeyValueStore("digital-twin-store"),
            Serdes.String(),
            JsonSerdes.DigitalTwin());

    builder.addStateStore(storeBuilder, "Digital Twin Processor"); ❻

    builder.addSink( ❼
        "Digital Twin Sink",
        "digital-twins",
        Serdes.String().serializer(),
        JsonSerdes.DigitalTwin().serializer(),
        "Digital Twin Processor");
```

❶ Create a *source processor* named `Desired State Events` that consumes data from the `desired-state-events` topic. This is the equivalent of a *stream* in the DSL.

❷ Create a *source processor* named `Reported State Events` that consumes data from the `reported-state-events` topic. This is also the equivalent of a *stream* in the DSL.

❸ Add a *stream processor* named `High Winds Flatmap Processor` that generates a shutdown signal if high winds are detected. This processor receives events from the `Reported State Events` processor. This would be a `flatMap` operation in the DSL since there is a 1:N relationship between the number of input and output records for this stream processor. Example 7-3 shows the implementation of this processor.

4. Add a *stream processor* named `Digital Twin Processor` that creates digital twin records using data emitted from both the `High Winds Flatmap Processor` and `Desired State Events`. This would be a *merge* operation in the DSL since multiple sources are involved. Furthermore, since this is a stateful processor, this would be the equivalent of an aggregated *table* in the DSL. Example 7-5 shows the implementation of this processor.

5. Use the `Stores` factory class to create a *store builder*, which can be used by Kafka Streams to build persistent key-value stores that are accessible from the `Digital Twin Processor` node.

6. Add the state store to the topology and connect it to the `Digital Twin Processor` node.

7. Create a *sink processor* named `Digital Twin Sink` that writes all digital twin records that get emitted from the `Digital Twin Processor` node to an output topic named `digital-twins`.

We can now run our application, write some test data to our Kafka cluster, and query our digital twin service. Running our application is no different than what we've seen in previous chapters, as you can see from the following code block:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev-consumer"); ❶
// ...

KafkaStreams streams = new KafkaStreams(builder, props); ❷
streams.start(); ❸

Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❹

RestService service = new RestService(hostInfo, streams); ❺
service.start();
```

1. Configure the Kafka Streams application. This works the same way as we saw when building applications with the DSL. Most of the configs are omitted for brevity's sake.

2. Instantiate a new `KafkaStreams` instance that can be used to execute our topology.

3. Start the Kafka Streams application.

❹ Add a shutdown hook to gracefully stop the Kafka Streams application when a global shutdown signal is received.

❺ Instantiate and (on the following line) start the REST service, which we implemented in Example 7-6.

Our application now reads from multiple source topics, but we will only produce test data to the `reported-state-events` topic in this book (see the source code for a more complete example). To test that our application generates a shutdown signal, wewill include one record that contains a wind speed that exceeds our safe operating threshold of 65 mph. The following code shows the test data we will produce, with record keys and values separated by `|` and timestamps omitted for brevity:

```
1|{"timestamp": "...", "wind_speed_mph": 40, "power": "ON", "type": "REPORTED"}
1|{"timestamp": "...", "wind_speed_mph": 42, "power": "ON", "type": "REPORTED"}
1|{"timestamp": "...", "wind_speed_mph": 44, "power": "ON", "type": "REPORTED"}
1|{"timestamp": "...", "wind_speed_mph": 68, "power": "ON", "type": "REPORTED"} ❶
```

❶ This sensor data shows a wind speed of 68 mph. When our application sees this record, it should generate a shutdown signal by creating a new `TurbineState` record, with a desired power state of OFF.

If we produce this test data into the `reported-state-events` topic and then query our digital twin service, we will see that our Kafka Streams application not only processed the reported states of our windmill, but also produced a desired state record where the power is set to `OFF`. The following code block shows an example request and response to our REST service:

```
$ curl localhost:7000/devices/1 | jq '.'

{
  "desired": {
    "timestamp": "2020-11-23T09:02:01.000Z",
    "windSpeedMph": 68,
    "power": "OFF",
    "type": "DESIRED"
  },
  "reported": {
    "timestamp": "2020-11-23T09:02:01.000Z",
    "windSpeedMph": 68,
    "power": "ON",
    "type": "REPORTED"
  }
```

Now, our wind turbines can query our REST service and synchronize their own state with the desired state that was either captured (via the `desired-state-events` topic) or enforced (using the high winds processor to send a shutdown signal) by Kafka Streams.

Combining the Processor API with the DSL

We’ve verified that our application works. However, if you look closely at our code, you’ll see that only one of the topology steps requires the low-level access that the Processor API offers. The step I’m referring to is the `Digital Twin Processor` step (see step 3 in Figure 7-1), which leverages an important feature of the Processor API: the ability to schedule periodic functions.

Since Kafka Streams allows us to combine the Processor API and the DSL, we can easily refactor our application to *only* use the Processor API for the `Digital Twin Processor` step, and to use the DSL for everything else. The biggest benefit of per- forming this kind of refactoring is that other stream processing steps can be simpli- fied. In this tutorial, the `High Winds Flatmap Processor` offers the biggest opportunity of simplification, but in larger applications, this kind of refactoringreduces complexity on an even greater scale.

The first two steps in our processor topology (registering the source processors and generating a shutdown signal using a `flatMap`-like operation) can be refactored using operators we’ve already discussed in this book. Specifically, we can make the following changes:

Processor API	DSL
`Topology builder = new Topology();`	`StreamsBuilder builder = new StreamsBuilder();`

`builder.addSource(`
`  "Desired State Events",`
`  Serdes.String().deserializer(),`
`  JsonSerdes.TurbineState().deserializer(),`
`  "desired-state-events");`
	`KStream<String, TurbineState> desiredStateEvents =`
	`  builder.stream("desired-state-events",`
	`    Consumed.with(`
	`      Serdes.String(),`
	`      JsonSerdes.TurbineState()));`

`builder.addSource(`
`  "Reported State Events",`
`  Serdes.String().deserializer(),`
`  JsonSerdes.TurbineState().deserializer(),`
`  "reported-state-events");`
	`KStream<String, TurbineState> highWinds =`
	`  builder.stream("reported-state-events",`
	`    Consumed.with(`
	`      Serdes.String(),`
	`      JsonSerdes.TurbineState()))`
	`  .flatMapValues((key, reported) -> ... )`
	`  .merge(desiredStateEvents);`

`builder.addProcessor(`
`  "High Winds Flatmap Processor",`
`  HighWindsFlatmapProcessor::new,`
`  "Reported State Events");`	

As you can see, these changes are pretty straightforward. However, step 3 in our top- ology actually does require the Processor API, so how do we go about combining the DSL and Processor API in this step? The answer lies in a special set of DSL operators, which we will explore next.

Processors and Transformers

The DSL includes a special set of operators that allow us to use the Processor API whenever we need lower-level access to state stores, record metadata, and processor context (which can be used for scheduling periodic functions, among other things). These special operators are broken into two categories: *processors* and *transformers*. The following outlines the distinction between these two groups:

- A *processor* is a *terminal operation* (meaning it returns void and downstream operators cannot be chained), and the computational logic must be implemented using the `Processor` interface. Processors should be used whenever you need to leverage the Processor API from the DSL, but don't need to chain any downstream operators. There is currently only one variation of this type of operator, as shown in the following table:

DSL operator	Interface to implement	Description
`process`	`Processor`	Apply a `Processor` to each record at a time

- *Transformers* are a more diverse set of operators and can return one or more records (depending on which variation you use), and are therefore more optimal if you need to chain a downstream operator. The variations of the transform operator are shown in Table 7-4.

Table 7-4. Various transform operators that are available in Kafka Streams

DSL operator	Interface to implement	Description	Input/output ratio
`transform`	`Transformer`	Apply a `Transformer` to each record, generating one or more output records. Single records can be returned from the `Transformer#transform` method, and multiple values can be emitted using `ProcessorContext#forward`.[a] The transformer has access to the record key, value, metadata, processor context (which can be used for scheduling periodic functions), and connected state stores.	1:N

DSL operator	Interface to implement	Description	Input/ output ratio
transform Values	ValueTransformer	Similar to transform, but *does not* have access to the record key and *cannot* forward multiple records using ProcessorCon text#forward (if you try to forward multiple records, you will get a StreamsEx ception). Since state store operations are key-based, this operator is not ideal if you needto perform lookups against a state store. Furthermore, output records will have the same key as the input records, and downstream auto-repartitioning will not be triggered since the key cannot be modified (which is advantageous since it can help avoid network trips).	1:1
transform Values	ValueTransformerWithKey	Similar to transform, but the record key is *read-only* and *should not be modified*. Also, you cannot forward multiple records using ProcessorContext#forward (if you try to forward multiple records, you will get a StreamsException).	1:1
flatTrans form	Transformer (with an iterable return value)	Similar to transform, but instead of relying on ProcessorContext#forward to return multiple records, you can simply return a collection of values. For this reason, it's recommended to use flatTransform over transform if you need to emit multiple records, since this method is type-safe while the latter is not (since it relies on Processor Context#forward).	1:N
flatTrans formValues	ValueTransformer (with an iterable return value)	Apply a Transformer to each record, returning one or more output records directly from the ValueTransformer#trans form method.	1:N
flatTrans formValues	ValueTransformerWithKey (with an iterable return value)	A stateful variation of flatTransformVal ues in which a read-only key is passed to the transform method, which can be used for state lookups. One or more output records are returned directly from the ValueTransfor merWithKey#transform method.	1:N

[a] Though 1:N transformations are technically supported, transform is better for 1:1 or 1:0 transformations in which a single record is returned directly, since the ProcessorContext#forward approach is not type-safe. Therefore, if you need to forward multiple records from your transform, flatTransform is recommended instead, since it is type-safe.

No matter which variation you choose, if your operator is stateful, you will need to connect the state store to your topology builder before adding your new operator.

Since we are refactoring our stateful `Digital Twin Processor` step, let's go ahead and do that:

```
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
  Stores.keyValueStoreBuilder(
      Stores.persistentKeyValueStore("digital-twin-store"),
      Serdes.String(),
      JsonSerdes.DigitalTwin());

builder.addStateStore(storeBuilder); ❶
```

❶ In Example 7-4, we discussed an optional second parameter to the `Topology#addStateStore` method, which specifies the processor names that should be connected with the state store. Here, we omit the second parameter, so this state store is *dangling* (though we will connect it in the next code block).

Now, we need to make a decision. Do we use a processor or transformer for refactoring the `Digital Twin Processor` step? Looking at the definitions in the preceding tables, you may be tempted to use the `process` operator since we already implemented the `Processor` interface in the pure Processor API version of our app (see Example 7-5). If we were to take this approach (which is problematic for reasons we'll discuss shortly), we'd end up with the following implementation:

```
highWinds.process(
  DigitalTwinProcessor::new, ❶
  "digital-twin-store"); ❷
```

❶ A `ProcessSupplier`, which is used to retrieve an instance of our `DigitalTwinProcessor`.

❷ The name of the state store that our processor will interact with.

Unfortunately, this isn't ideal because we need to connect a sink processor to this node, and the `process` operator is a terminal operation. Instead, one of the trans- former operators would work better here since it allows us to easily connect a sink processor (as we'll see shortly). Now, looking at Table 7-4, let's find an operator thatmeets our requirements:

- Each input record will always produce one output record (1:1 mapping)
- We need read-only access to the record key since we're performing point lookups in our state store, but do not need to modify the key in any way

The operator that best fits these requirements is `transformValues` (the variation that uses a `ValueTransformerWithKey`). We've already implemented the computational logic for this step using a `Processor` (see Example 7-5), so we just need to implement the `ValueTransformerWithKey` interface and copy the logic from the `process` method in Example 7-5 to the `transform` method, shown in the following. Most of the code has been omitted because it's the same as the processor implementation. The changes are highlighted in the annotations following this example:

```
public class DigitalTwinValueTransformerWithKey
    implements ValueTransformerWithKey<String, TurbineState, DigitalTwin> { ❶

  @Override
  public void init(ProcessorContext context) {
    // ...
  }

  @Override
  public DigitalTwin transform(String key, TurbineState value) {
    // ...
    return digitalTwin; ❷
  }

  @Override
  public void close() {
    // ...
  }

  public void enforceTtl(Long timestamp) {
    // ...
  }
}
```

❶ Implement the `ValueTransformerWithKey` interface. `String` refers to the key type, `TurbineState` refers to the value type of the input record, and `DigitalTwin` refers to the value type of the output record.

❷ Instead of using `context.forward` to send records to downstream processors, we can return the record directly from the `transform` method. As you can see, this is already feeling much more DSL-like.

With our transformer implementation in place, we can add the following line to our application:

```
highWinds
  .transformValues(DigitalTwinValueTransformerWithKey::new, "digital-twin-store")
  .to("digital-twins", Produced.with(Serdes.String(), JsonSerdes.DigitalTwin()));
```

Putting It All Together: Refactor

Now that we've discussed the individual steps in our DSL refactor, let's take a look at the two implementations of our application side by side, as seen in Table 7-5.

Table 7-5. Two different implementations of our digital twin topology

Processor API only	DSL + Processor API
`Topology builder = new Topology();`	`StreamsBuilder builder = new StreamsBuilder();`
`builder.addSource( "Desired State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState().deserializer(), "desired-state-events");`	`KStream<String, TurbineState> desiredStateEvents = builder.stream("desired-state-events", Consumed.with( Serdes.String(), JsonSerdes.TurbineState()));`
`builder.addSource( "Reported State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState().deserializer(), "reported-state-events");`	`KStream<String, TurbineState> highWinds = builder.stream("reported-state-events", Consumed.with( Serdes.String(), JsonSerdes.TurbineState())) .flatMapValues((key, reported) -> ... ) .merge(desiredStateEvents);`
`builder.addProcessor( "High Winds Flatmap Processor", HighWindsFlatmapProcessor::new, "Reported State Events");`	
`builder.addProcessor( "Digital Twin Processor", DigitalTwinProcessor::new, "High Winds Flatmap Processor", "Desired State Events");`	`// empty space to align next topology step`
`StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder = Stores.keyValueStoreBuilder( Stores.persistentKeyValueStore( "digital-twin-store"), Serdes.String(), JsonSerdes.DigitalTwin());`	`StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder = Stores.keyValueStoreBuilder( Stores.persistentKeyValueStore( "digital-twin-store"), Serdes.String(), JsonSerdes.DigitalTwin());`
`builder.addStateStore(storeBuilder, "Digital Twin Processor");`	`builder.addStateStore(storeBuilder);`
`builder.addSink( "Digital Twin Sink", "digital-twins", Serdes.String().serializer(), JsonSerdes.DigitalTwin().serializer(), "Digital Twin Processor");`	`highWinds .transformValues( DigitalTwinValueTransformerWithKey::new, "digital-twin-store") .to("digital-twins", Produced.with( Serdes.String(), JsonSerdes.DigitalTwin()));`

Either implementation is perfectly fine. But going back to something I mentioned earlier, you don't want to introduce additional complexity unless you have a good reason for doing so.

The benefits of the hybrid DSL + Processor API implementation are:

- It’s easier to construct a mental map of your dataflow by chaining operators instead of having to define the relationship between processors using node names and parent names.
- The DSL has lambda support for most operators, which can be beneficial for succinct transformations (the Processor API requires you to implement the `Processor` interface, even for simple operations, which can be tedious).
- Although we didn’t need to rekey any records in this tutorial, the method for doing this in the Processor API is much more tedious. You not only need to implement the `Processor` interface for a simple rekey operation, but you also have to handle the rewrite to an intermediate repartition topic (this involves adding an additional sink and source processor explicitly, which can lead to unnecessarily complex code).
- The DSL operators give us a standard vocabulary for defining what happens at a given stream processing step. For example, we can infer that a `flatMap` operator may produce a different number of records than the input, without knowing any- thing else about the computational logic. On the other hand, the Processor API makes it easy to disguise the nature of a given `Processor` implementation, whichhurts code readability and can have a negative impact on maintenance.
- The DSL also gives us a common vocabulary for different types of streams. Theseinclude pure record streams, local aggregated streams (which we usually refer to as tables), and global aggregated streams (which we refer to as global tables).

Therefore, I usually recommend leveraging the DSL’s special set of operators for using the Processor API whenever you need lower-level access, as opposed to implementingan application purely in the Processor API.