
**Apache Kafka**

- Kafka streams is a client library for processing and analyzing data stored in Kafka. Being a lightweight and straightforward library, it is the most convincing feature of Kafka Streams.
- We can easily embed Kafka streams library in our Java application and execute it on a single machine or take advantage of our preferred packaging and deployment frameworks like docker.
- When we want to scale our system, we can run multiple instances of the same application on multiple machines. Yet, we do not need to get into the complexities of load balancing.
- Kafka Streams transparently handles the load balancing of multiple instances of the same application by leveraging Kafka's parallelism model.
- The core functionality of Kafka Streams library is available in two flavors.
  1. Streams DSL
  2. Processor API

1. **Streams DSL**

✓ Streams DSL is a high-level API that offers common data transformation functions. Kafka streams DSL is a very well thought API set that allows us to handle most of the stream processing needs.

✓ DSL is the recommended way to create Kafka Streams application, and it should cover most of our needs and most of the use cases.

2. **Processor API**

✓ Processor APIs are the low-level API, and they provides us with more flexibility than the DSL. Using Processor API requires little extra manual work and code on the application developer side.

✓ However, they allow us to handle those one-off problems that may not be possible to solve with higher level abstraction.

Let's start with a simple Kafka Stream application. As our first example, we want to create a simple Kafka streams application to do the following things.

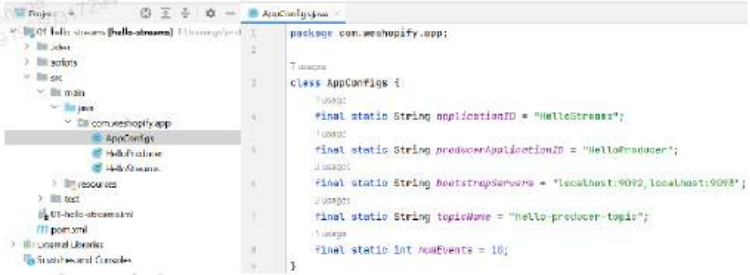
1. Connect to the Kafka cluster and start reading data from a given topic.


**Apache Kafka**

2. We do not want to get into the processing that data yet, but we just want to print the messages on the console.

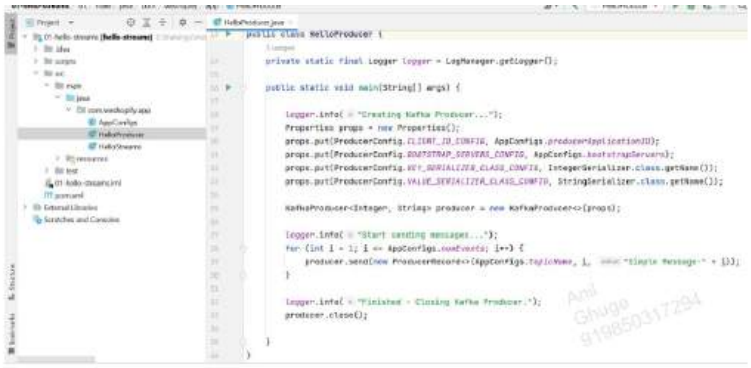
**Implementation**

**Step-1:** Implement a AppConfigs.java class as shown below:



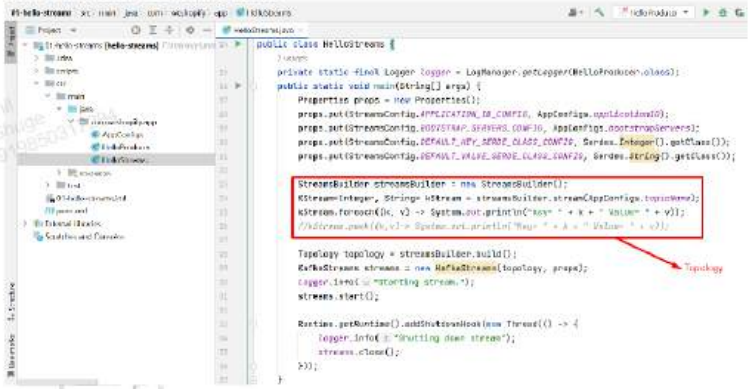
```
package com.weshopify.app;

class AppConfigs {
    final static String applicationID = "HelloStreams";
    final static String producerApplicationID = "HelloProducer";
    final static String bootstrapServers = "localhost:9092,localhost:9093";
    final static String topicName = "hello-producer-topic";
    final static int numEvents = 10;
}
```

Step-2: Implement a Producer as shown below:



**Apache Kafka**

**Step-3: Implement a Streaming class.**



At a high level, creating a Kafka Streams application is a four-step process.

Step-1: Line-17-to-21:

```
17          Properties props = new Properties();
18          props.put(StreamsConfig.APPLICATION_ID_CONFIG, AppConfigs.applicationID);
19          props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, AppConfigs.bootstrapServers);
20          props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.Integer().getClass());
21          props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

- The first step is to create a Java Properties object and put the necessary configuration in the Properties object. Kafka streams API is highly configurable, and we use Java Properties object to specify those configurations. In the above example, we are setting four most basic settings.
  - ✓ APPLICATION_ID_CONFIG
  - ✓ BOOTSTRAP_SERVERS_CONFIG
  - ✓ DEFAULT_KEY_SERDE_CLASS_CONFIG
  - ✓ DEFAULT_VALUE_SERDE_CLASS_CONFIG

Every Kafka Streams application must be identified by a unique application id. If we are executing multiple instances of our Kafka Streams application, all instances of the same application must have the same application id.

The application id is used by Kafka Streams application in many ways to isolate resources for the same application and share workload among the instances of the same application.

The next configuration is the bootstrap server which we already know, A bootstrap server is a comma-separated list of host/port pairs that streams application would use to establishing the initial connection to the Kafka cluster.

The next two configurations are the Key and the Value Serdes. Serdes is a Factory class that we will be using for creating a Serde (a combination of serializer and a matching deserializer).

In this example, we are setting an Integer Serde for the key and a String Serde for the value. we might be wondering, why do we need a Serde?

While creating producers, we required a serializer and consumers needed a deserializer. But why do we need Serde (a combination of both) for the Streams API?

This example is a data consuming stream application. However, ***a typical Kafka Streams application internally creates a combination of consumer and producer.*** Hence, we need a serializer as well as a deserializer for the internal producer/consumer to work correctly. Therefore, we set a default Serdes for the key and a value.

**Step-2: Line-23-to-28:**

```
23      StreamsBuilder streamsBuilder = new StreamsBuilder();
24      KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
25      kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
26      //kStream.peek((k,v)-> System.out.println("Key= " + k + " Value= " + v));
27
28      Topology topology = streamsBuilder.build();
```

Next comes the central part of our Kafka Streams application. In this part, we define our streams computational logic that we want to execute.

For our example, the computation logic is as straightforward as following steps.

1. Open a stream to a source topic: define a Kafka stream for a Kafka topic that can be used to read all the messages.
2. Process the stream: for each message, print the Key and the Value.


Apache Kafka

3. Create a Topology: bundle your business logic into a stream topology.

**1. Opening a Stream**

We want to use Kafka Streams DSL for defining the above computational logic. Most of the DSL APIs are available through StreamsBuilder() class.

So, the first step is to create a StreamBuilder object. After creating a builder, we can open a Kafka Stream using the stream() method on the StreamBuilder. The stream() method takes a Kafka topic name and returns a KStream object.

Once we get a KStream object, we have successfully completed the first part of our computational logic I.e. Opening a stream to a source topic.

```
23      StreamsBuilder streamsBuilder = new StreamsBuilder();
24      KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
```

**2. Process the stream**

KStream class provides a bunch of methods for us to build our computational logic. We just want to navigate each message (key/value pair) and apply a println. So, the example is making use of the foreach() method on KStream to implement the second part of our computational logic I.e. Process the stream.

```
StreamsBuilder streamsBuilder = new StreamsBuilder();
KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
```

Instead of using foreach() method, we can also use the peek() method to work with each record in the stream.

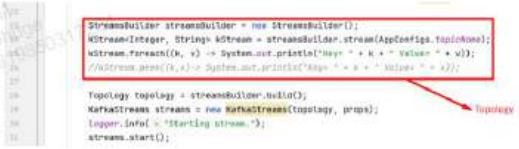
```
StreamsBuilder streamsBuilder = new StreamsBuilder();
KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
//kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
kStream.peek((k,v)-> System.out.println("Key= " + k + " Value= " + v));
```

The foreach() and the peek() method takes a lambda expression on key/value pair, and the logic to work with each pair is implemented in the lambda body. That's all. That's what we wanted to do as a computational logic.


**Apache Kafka**

**3. Create a Topology**

The Kafka Streams computational logic is known as a Topology and is represented by the Topology class. So, whatever we defined as computational logic, we can get all that bundled into a Topology object by making a call to the build() method on the StreamsBuilder object.




That's all. That's want we wanted to achieve in the Step-2.

However, we implemented the computational logic by creating some intermediate objects such as KStream, and Topology.

A typical application may not be creating and holding these intermediate objects. Instead, we prefer to define all these operations as a chain of methods to avoid unnecessary code clutter as shown in the below code implemented as a chain of methods.

```
StreamsBuilder builder = new StreamsBuilder();
builder.stream(topicName)
        .foreach((k, v) -> System.out.println("Key = " + k + " Value
= " + v));
KafkaStreams streams = new KafkaStreams(builder.build(),
props);
streams.start();
```

**Step 3 (Line 12 – 14):**

Once we have the Properties and the Topology, we are ready to instantiate the KafkaStreams and start() it. That's all we do in a typical Kafka Streams application.

```
29      KafkaStreams streams = new KafkaStreams(topology, props);
30      logger.info( s: "Starting stream.");
31      streams.start();
```

We can summarize a Kafka Streams application in just three steps.


**Apache Kafka**

Configure using Properties Create Topology using StreamBuilder Create and start KafkaStreams using Properties and Topology.

- We have seen a simple Kafka Streams application. We must have realized that the core of our Kafka streams application is the Topology of our application. A typical Kafka Streams application define its computational logic through one or more processor typologies. The next important point to notice

is that our Kafka Streams application does not Kafka streams is a client library for processing and analyzing data stored in Kafka. Being a lightweight and straightforward library, it is the most convincing feature of Kafka Streams.

- We can easily embed Kafka streams library in our Java application and execute it on a single machine or take advantage of our preferred packaging and deployment frameworks like docker.
- When we want to scale our system, we can run multiple instances of the same application on multiple machines. Yet, we do not need to get into the complexities of load balancing.
- Kafka Streams transparently handles the load balancing of multiple instances of the same application by leveraging Kafka's parallelism model.
- The core functionality of Kafka Streams library is available in two flavors.
  3. Streams DSL
  4. Processor API



3. **Streams DSL**

- ✓ Streams DSL is a high-level API that offers common data transformation functions. Kafka streams DSL is a very well thought API set that allows us to handle most of the stream processing needs.
- ✓ DSL is the recommended way to create Kafka Streams application, and it should cover most of our needs and most of the use cases.

4. **Processor API**

- ✓ Processor APIs are the low-level API, and they provides us with more flexibility than the DSL. Using Processor API requires little extra manual work and code on the application developer side.


**Apache Kafka**

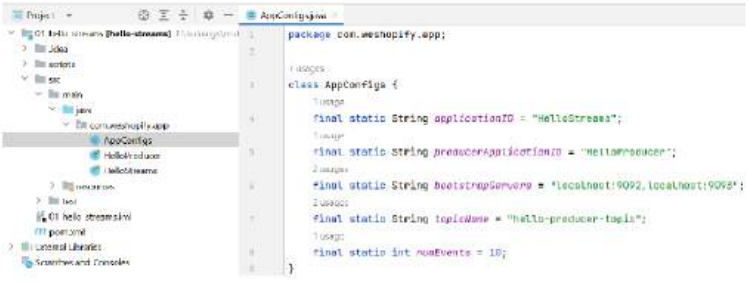
- ✓ However, they allow us to handle those one-off problems that may not be possible to solve with higher level abstraction.

Let's start with a simple Kafka Stream application. As our first example, we want to create a simple Kafka streams application to do the following things.

3. Connect to the Kafka cluster and start reading data from a given topic.
4. We do not want to get into the processing that data yet, but we just want to print the messages on the console.

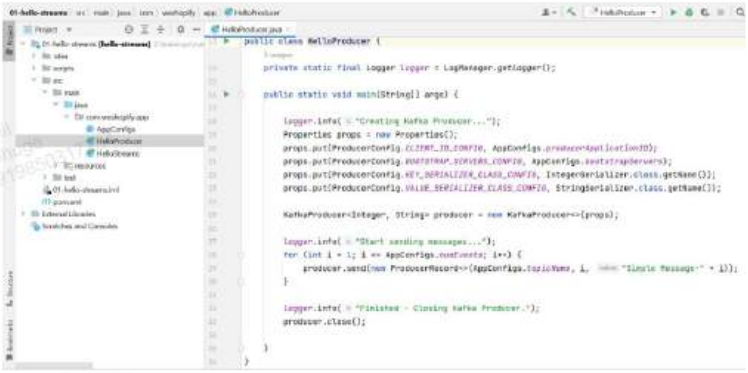
**Implementation**

Step-1: Implement a AppConfigs.java class as shown below:

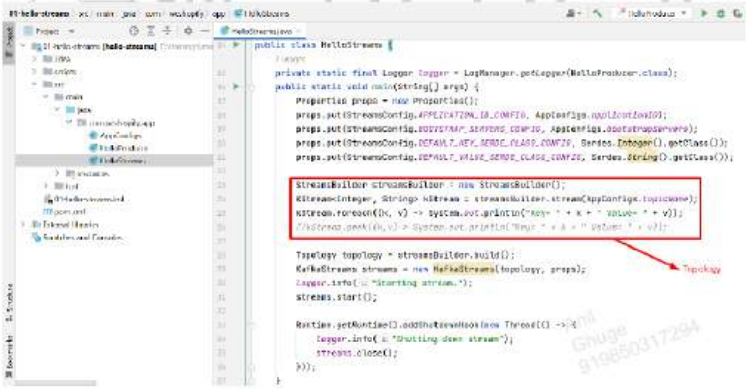



Step-2: Implement a Producer as shown below:


**Apache Kafka**



**Step-3: Implement a Streaming class.**



At a high level, creating a Kafka Streams application is a four-step process.


**Apache Kafka**

**Step-1: Line-17-to-21:**

```
17      Properties props = new Properties();
18      props.put(StreamsConfig.APPLICATION_ID_CONFIG, AppConfigs.applicationID);
19      props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, AppConfigs.bootstrapServers);
20      props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.Integer().getClass());
21      props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

- The first step is to create a Java Properties object and put the necessary configuration in the Properties object. Kafka streams API is highly configurable, and we use Java Properties object to specify those configurations. In the above example, we are setting four most basic settings.
  - ✓ APPLICATION_ID_CONFIG
  - ✓ BOOTSTRAP_SERVERS_CONFIG
  - ✓ DEFAULT_KEY_SERDE_CLASS_CONFIG

- ✓ DEFAULT_VALUE_SERDE_CLASS_CONFIG

Every Kafka Streams application must be identified by a unique application id. If we are executing multiple instances of our Kafka Streams application, all instances of the same application must have the same application id.

The application id is used by Kafka Streams application in many ways to isolate resources for the same application and share workload among the instances of the same application.

The next configuration is the bootstrap server which we already know, A bootstrap server is a comma-separated list of host/port pairs that streams application would use to establishing the initial connection to the Kafka cluster.

The next two configurations are the Key and the Value Serdes. Serdes is a Factory class that we will be using for creating a Serde (a combination of serializer and a matching deserializer).

In this example, we are setting an Integer Serde for the key and a String Serde for the value. we might be wondering, why do we need a Serde?

While creating producers, we required a serializer and consumers needed a deserializer. But why do we need Serde (a combination of both) for the Streams API?

This example is a data consuming stream application. However, ***a typical Kafka Streams application internally creates a combination of consumer and producer.*** Hence, we need a serializer as well as a deserializer for the internal producer/consumer to work correctly. Therefore, we set a default Serdes for the key and a value.


**Apache Kafka**

**Step-2: Line-23-to-28:**

```
23          StreamsBuilder streamsBuilder = new StreamsBuilder();
24          KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
25          kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
26          //kStream.peek((k,v)-> System.out.println("Key= " + k + " Value= " + v));
27
28          Topology topology = streamsBuilder.build();
```

Next comes the central part of our Kafka Streams application. In this part, we define our streams computational logic that we want to execute.

For our example, the computation logic is as straightforward as following steps.

4. Open a stream to a source topic: define a Kafka stream for a Kafka topic that can be used to read all the messages.
5. Process the stream: for each message, print the Key and the Value.
6. Create a Topology: bundle your business logic into a stream topology.

**4. Opening a Stream**

We want to use Kafka Streams DSL for defining the above computational logic. Most of the DSL APIs are available through StreamsBuilder() class.

So, the first step is to create a StreamBuilder object. After creating a builder, we can open a Kafka Stream using the stream() method on the StreamBuilder. The stream() method takes a Kafka topic name and returns a KStream object.

Once we get a KStream object, we have successfully completed the first part of our computational logic I.e. Opening a stream to a source topic.

```
23          StreamsBuilder streamsBuilder = new StreamsBuilder();
24          KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
```

**5. Process the stream**

KStream class provides a bunch of methods for us to build our computational logic. We just want to navigate each message (key/value pair) and apply a println. So, the example is making use of the foreach() method on KStream to implement the second part of our computational logic I.e. Process the stream.


**Apache Kafka**

```
StreamsBuilder streamsBuilder = new StreamsBuilder();
KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
```

Instead of using foreach() method, we can also use the peek() method to work with each record in the stream.

```
23        StreamsBuilder streamsBuilder = new StreamsBuilder();
24        KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
25        //kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
26        kStream.peek((k,v)-> System.out.println("Key= " + k + " Value= " + v));
```

The foreach() and the peek() method takes a lambda expression on key/value pair, and the logic to work with each pair is implemented in the lambda body. That's all. That's what we wanted to do as a computational logic.

**6. Create a Topology**

The Kafka Streams computational logic is known as a Topology and is represented by the Topology class. So, whatever we defined as computational logic, we can get all that bundled into a Topology object by making a call to the build() method on the StreamsBuilder object.

```
StreamsBuilder streamsBuilder = new StreamsBuilder();
KStream<Integer, String> kStream = streamsBuilder.stream(AppConfigs.topicName);
kStream.foreach((k, v) -> System.out.println("Key= " + k + " Value= " + v));
//kStream.peek((k,v)-> System.out.println("Key= " + k + " Value= " + v));

Topology topology = streamsBuilder.build();
KafkaStreams streams = new KafkaStreams(topology, props);
logger.info( s: "Starting stream.");
streams.start();
```

Topology

That's all. That's want we wanted to achieve in the Step-2.

However, we implemented the computational logic by creating some intermediate objects such as KStream, and Topology.

A typical application may not be creating and holding these intermediate objects. Instead, we prefer to define all these operations as a chain of methods to avoid unnecessary code clutter as shown in the below code implemented as a chain of methods.


**Apache Kafka**

```
StreamsBuilder builder = new StreamsBuilder();
builder.stream(topicName)
        .foreach((k, v) -> System.out.println("Key = " + k + " Value
= " + v));
KafkaStreams streams = new KafkaStreams(builder.build(),
props);
streams.start();
```

**Step 3 (Line 12 – 14):**

Once we have the Properties and the Topology, we are ready to instantiate the KafkaStreams and start() it. That's all we do in a typical Kafka Streams application.

```
29        KafkaStreams streams = new KafkaStreams(topology, props);
30        logger.info( s: "Starting stream.");
31        streams.start();
```

We can summarize a Kafka Streams application in just three steps.

1. Configure using Properties
2. Create Topology using StreamBuilder
3. Create and start KafkaStreams using Properties and Topology.

We have seen a simple Kafka Streams application. We must have realized that the core of our Kafka streams application is the Topology of our application.

A typical Kafka Streams application define its computational logic through one or more processor typologies.

***The next important point to notice is that our Kafka Streams application does not execute inside the Kafka cluster or the Kafka Broker. Instead, it runs in a separate JVM instance, or in a different machine entirely.***

For reading and writing data to the Kafka cluster, it creates a remote connection using the bootstrap servers configuration. The simplicity to run our Kafka Streams application allows us to use our favourite packaging and deployment infrastructure such as Docker and Kubernetes container platforms.

Finally, we can run a single instance of our application or alternatively, we can start multiple instances of the same application to scale our production deployment horizontally.

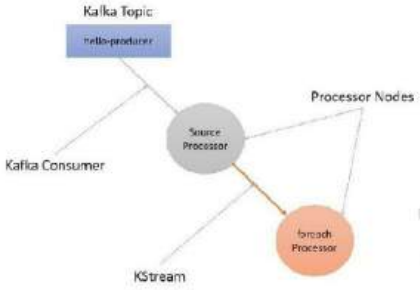

**Apache Kafka**

Kafka Streams API is designed to help us elastically scale and parallelize our application by simply starting more and more instances.

These additional instances not only contribute to share workload but also provide automatic fault-tolerance.

**Streams Topology**

A processor topology or simply topology defines a step-by-step computational logic of the data processing that needs to be performed by a stream processing application. We can easily represent these steps using a DAG(Directed Acyclic Graph)




Hello Streams topology is reasonably straightforward. However, it helps us to visualize and uncover several concepts.

**Source Processor**

Every Kafka streams application or the topology starts by subscribing to some Kafka topics, and that's fair because we need to consume a data stream to begin our processing.

So, our topology starts by creating a source processor. Creating a source processor is straightforward. The below figure shows the code snippet from the Hello Streams example that creates a source processor.

```
KStream<Integer, String> kStream = builder.stream(topicName);
```

Source Processor


**Apache Kafka**

As shown above, a source processor internally implements a Kafka Consumer to consume data from one or multiple Kafka topics, and it produces a KStream object that will be used by its down-stream processors as an input stream.

Once we have a source processor and our first KStream object, we are ready to add more processors in the topology. Adding a new processor node is as simple as calling a transformation method on the KStream object.

For example, we take kStream that was returned by the source node and make a call to the kStream.foreach() method. This operation simply adds a foreach() processor node to the topology.