**1. Overview**

Apache Flink is a Big Data processing framework that allows programmers to process a vast amount of data in a very efficient and scalable manner.

In this article, we’ll introduce some of the **core API concepts and standard data transformations available in the**[***Apache Flink***](https://ci.apache.org/projects/flink/flink-docs-release-1.2/index.html)**Java API**. The fluent style of this API makes it easy to work with Flink’s central construct – the distributed collection.

First, we will take a look at Flink’s *DataSet* API transformations and use them to implement a word count program. Then we will take a brief look at Flink’s *DataStream* API, which allows you to process streams of events in a real-time fashion.

**2. Maven Dependency**

To get started we’ll need to add Maven dependencies to *[flink-java](https://mvnrepository.com/artifact/org.apache.flink/flink-java)* and *[flink-test-utils](https://mvnrepository.com/artifact/org.apache.flink/flink-test-utils)* libraries:

<**dependency**>

<**groupId**>org.apache.flink</**groupId**>

<**artifactId**>flink-java</**artifactId**>

<**version**>1.16.1</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.apache.flink</**groupId**>

<**artifactId**>flink-test-utils</**artifactId**>

<**version**>1.16.1</**version**>

<**scope**>test<**scope**>

</**dependency**>

**3. Core API Concepts**

When working with Flink, we need to know a couple of things related to its API:

* **Every Flink program performs transformations on distributed collections of data.** A variety of functions for transforming data are provided, including filtering, mapping, joining, grouping, and aggregating
* **A *sink* operation in Flink triggers the execution of a stream to produce the desired result of the program**, such as saving the result to the file system or printing it to the standard output
* **Flink transformations are lazy, meaning that they are not executed until a *sink* operation is invoked**
* **The Apache Flink API supports two modes of operations — batch and real-time.**If you are dealing with a limited data source that can be processed in batch mode, you will use the *DataSet* API. Should you want to process unbounded streams of data in real time, you would need to use the *DataStream* API

**4. DataSet API Transformations**

The entry point to the Flink program is an instance of the *[ExecutionEnvironment](https://ci.apache.org/projects/flink/flink-docs-release-1.2/api/java/org/apache/flink/api/scala/ExecutionEnvironment.html)* class — this defines the context in which a program is executed. Let’s create an *ExecutionEnvironment* to start our processing:

**ExecutionEnvironment** env

= ExecutionEnvironment.getExecutionEnvironment();

**Note that when you launch the application on the local machine, it will perform processing on the local JVM.**Should you want to start processing on a cluster of machines, you would need to install[Apache Flink](https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/try-flink/local_installation/) on those machines and configure the *ExecutionEnvironment* accordingly.

**4.1. Creating a DataSet**

To start performing data transformations, we need to supply our program with the data. Let’s create an instance of the *DataSet*class using our *ExecutionEnvironement*:

DataSet<Integer> amounts = env.fromElements(1, 29, 40, 50);

You can create a *DataSet*from multiple sources, such as Apache Kafka, a CSV, a file or virtually any other data source.

**4.2. Filter and Reduce**

Once you create an instance of the *DataSet*class, you can apply transformations to it.

Let’s say that you want to filter numbers that are above a certain threshold and next sum them all*.*You can use the *filter()*and *reduce()* transformations to achieve this:

**int** threshold = 30;

List<Integer> collect = amounts

.filter(a -> a > threshold)

.reduce((integer, t1) -> integer + t1)

.collect();

assertThat(collect.get(0)).isEqualTo(90);

Note that the *collect()*method is a *sink* operation that triggers the actual data transformations.

**4.3. Map**

Let’s say that you have a *DataSet* of *Person*objects:

**private** **static** **class** **Person** {

**private** **int** age;

**private** String name;

}

Next, let’s create a *DataSet*of these objects:

DataSet<Person> personDataSource = env.fromCollection(

Arrays.asList(

**new** **Person**(23, "Tom"),

**new** **Person**(75, "Michael")));

Suppose that you want to extract only the *age* field from every object of the collection. You can use the *map()*transformation to get only a specific field of the *Person*class:

List<Integer> ages = personDataSource

.map(p -> p.age)

.collect();

assertThat(ages).hasSize(2);

assertThat(ages).contains(23, 75);

**4.4. Join**

When you have two datasets, you may want to join them on some *id* field. For this, you can use the *join()* transformation. Let’s create collections of transactions and addresses of a user:

Tuple3<Integer, String, String> address = **new** **Tuple3**<>(1, "5th Avenue", "London");

DataSet<Tuple3<Integer, String, String>> addresses = env.fromElements(address);

Tuple2<Integer, String> firstTransaction = **new** **Tuple2**<>(1, "Transaction\_1");

DataSet<Tuple2<Integer, String>> transactions = env.fromElements(firstTransaction, **new** **Tuple2**<>(12, "Transaction\_2"));

The first field in both tuples is of an *Integer* type, and this is an *id* field on which we want to join both data sets.

To perform the actual joining logic, we need to implement a *[KeySelector](https://ci.apache.org/projects/flink/flink-docs-release-1.2/api/java/org/apache/flink/api/java/functions/KeySelector.html)*interface for address and transaction:

**private** **static** **class** **IdKeySelectorTransaction**

**implements** **KeySelector**<Tuple2<Integer, String>, Integer> {

@Override

**public** Integer **getKey**(Tuple2<Integer, String> value) {

**return** value.f0;

}

}

**private** **static** **class** **IdKeySelectorAddress**

**implements** **KeySelector**<Tuple3<Integer, String, String>, Integer> {

@Override

**public** Integer **getKey**(Tuple3<Integer, String, String> value) {

**return** value.f0;

}

}

Each selector is only returning the field on which the join should be performed.

**Unfortunately, it’s not possible to use lambda expressions here because Flink needs generic type info.**

Next, let’s implement merging logic using those selectors:

List<Tuple2<Tuple2<Integer, String>, Tuple3<Integer, String, String>>>

joined = transactions.join(addresses)

.where(**new** **IdKeySelectorTransaction**())

.equalTo(**new** **IdKeySelectorAddress**())

.collect();

assertThat(joined).hasSize(1);

assertThat(joined).contains(**new** **Tuple2**<>(firstTransaction, address));

**4.5. Sort**

Let’s say that you have the following collection of *Tuple2:*

Tuple2<Integer, String> secondPerson = **new** **Tuple2**<>(4, "Tom");

Tuple2<Integer, String> thirdPerson = **new** **Tuple2**<>(5, "Scott");

Tuple2<Integer, String> fourthPerson = **new** **Tuple2**<>(200, "Michael");

Tuple2<Integer, String> firstPerson = **new** **Tuple2**<>(1, "Jack");

DataSet<Tuple2<Integer, String>> transactions = env.fromElements(fourthPerson, secondPerson, thirdPerson, firstPerson);

If you want to sort this collection by the first field of the tuple, you can use the *sortPartitions()*transformation:

List<Tuple2<Integer, String>> sorted = transactions.sortPartition(**new** **IdKeySelectorTransaction**(), Order.ASCENDING).collect();

assertThat(sorted).containsExactly(firstPerson, secondPerson, thirdPerson, fourthPerson);

**5. Word Count**

The word count problem is one that is commonly used to showcase the capabilities of Big Data processing frameworks. The basic solution involves counting word occurrences in a text input. Let’s use Flink to implement a solution to this problem.

As the first step in our solution, we create a *LineSplitter* class that splits our input into tokens (words), collecting for each token a *Tuple2*of key-value pairs. In each of these tuples, the key is a word found in the text, and the value is the integer one (1). This class implements the *[FlatMapFunction](https://ci.apache.org/projects/flink/flink-docs-release-1.1/api/java/org/apache/flink/api/common/functions/FlatMapFunction.html)*interface that takes *String*as an input and produces a [*Tuple2*](https://nightlies.apache.org/flink/flink-docs-release-1.3/api/java/org/apache/flink/api/java/tuple/Tuple2.html)*<String, Integer>:*

**public** **class** **LineSplitter** **implements** **FlatMapFunction**<String, Tuple2<String, Integer>> {

@Override

**public** **void** **flatMap**(String value, Collector<Tuple2<String, Integer>> out) {

Stream.of(value.toLowerCase().split("\\W+"))

.filter(t -> t.length() > 0)

.forEach(token -> out.collect(**new** **Tuple2**<>(token, 1)));

}

}

We call the *collect()*method on the [*Collector*](https://ci.apache.org/projects/flink/flink-docs-release-1.0/api/java/org/apache/flink/util/class-use/Collector.html) class to push data forward in the processing pipeline. Our next and final step is to group the tuples by their first elements (words) and then perform a *sum* aggregate on the second element to produce a count of the word occurrences:

**public** **static** DataSet<Tuple2<String, Integer>> **startWordCount**(

ExecutionEnvironment env, List<String> lines) **throws** Exception {

DataSet<String> text = env.fromCollection(lines);

**return** text.flatMap(**new** **LineSplitter**())

.groupBy(0)

.aggregate(Aggregations.SUM, 1);

}

**We are using three types of Flink transformations: *flatMap()*, *groupBy()*, and *aggregate()*.**

Let’s write a test to assert that the word count implementation is working as expected:

List<String> lines = Arrays.asList(

"This is a first sentence",

"This is a second sentence with a one word");

DataSet<Tuple2<String, Integer>> result = WordCount.startWordCount(env, lines);

List<Tuple2<String, Integer>> collect = result.collect();

assertThat(collect).containsExactlyInAnyOrder(

**new** **Tuple2**<>("a", 3), **new** **Tuple2**<>("sentence", 2), **new** **Tuple2**<>("word", 1),

**new** **Tuple2**<>("is", 2), **new** **Tuple2**<>("this", 2), **new** **Tuple2**<>("second", 1),

**new** **Tuple2**<>("first", 1), **new** **Tuple2**<>("with", 1), **new** **Tuple2**<>("one", 1));

**6. DataStream API**

**6.1. Creating a DataStream**

Apache Flink also supports the processing of streams of events through its DataStream API. If we want to start consuming events, we first need to use the *StreamExecutionEnvironment*class:

**StreamExecutionEnvironment** executionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment();

Next, we can create a stream of events using the *executionEnvironment* from a variety of sources. It could be some message bus like *Apache Kafka*, but in this example, we will simply create a source from a couple of string elements:

DataStream<String> dataStream = executionEnvironment.fromElements(

"This is a first sentence",

"This is a second sentence with a one word");

We can apply transformations to every element of the *DataStream* like in the normal *DataSet*class:

SingleOutputStreamOperator<String> upperCase = text.map(String::toUpperCase);

To trigger the execution, we need to invoke a sink operation such as *print()*that will just print the result of transformations to the standard output, followed with the *execute()*method on the *StreamExecutionEnvironment*class:

upperCase.print();

env.execute();

It will produce the following output:

1> THIS IS A FIRST SENTENCE

2> THIS IS A SECOND SENTENCE WITH A ONE WORDCopy

**6.2. Windowing of Events**

When processing a stream of events in real-time, you may sometimes need to group events together and apply some computation on a window of those events.

Suppose we have a stream of events, where each event is a pair consisting of the event number and the timestamp when the event was sent to our system, and that we can tolerate events that are out-of-order but only if they are no more than twenty seconds late.

For this example, let’s first create a stream simulating two events that are several minutes apart and define a timestamp extractor that specifies our lateness threshold:

SingleOutputStreamOperator<Tuple2<Integer, Long>> windowed = env.fromElements(

**new** **Tuple2**<>(16, ZonedDateTime.now().plusMinutes(25).toInstant().getEpochSecond()),

**new** **Tuple2**<>(15, ZonedDateTime.now().plusMinutes(2).toInstant().getEpochSecond()))

.assignTimestampsAndWatermarks(

**new** **BoundedOutOfOrdernessTimestampExtractor**

<Tuple2<Integer, Long>>(Time.seconds(20)) {

@Override

**public** **long** **extractTimestamp**(Tuple2<Integer, Long> element) {

**return** element.f1 \* 1000;

}

});

Next, let’s define a window operation to group our events into five-second windows and apply a transformation on those events:

SingleOutputStreamOperator<Tuple2<Integer, Long>> reduced = windowed

.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)))

.maxBy(0, true);

reduced.print();

It will get the last element of every five-second window, so it prints out:

1> (15,1491221519)

Note that we do not see the second event because it arrived later than the specified lateness threshold.

**7. Conclusion**

In this article, we introduced the Apache Flink framework and looked at some of the transformations supplied with its API.

We implemented a word count program using Flink’s fluent and functional DataSet API. Then we looked at the DataStream API and implemented a simple real-time transformation on a stream of events.

**1. Overview**

[Apache Flink](https://flink.apache.org/) is a stream processing framework that can be used easily with Java. [Apache Kafka](https://kafka.apache.org/) is a distributed stream processing system supporting high fault-tolerance.

In this tutorial, we-re going to have a look at how to build a [data pipeline](https://www.baeldung.com/cs/data-pipelines) using those two technologies.

**2. Installation**

To install and configure Apache Kafka, please refer to the [official guide](https://kafka.apache.org/quickstart). After installing, we can use the following commands to create the new topics called *flink\_input* and *flink\_output:*

bin/kafka-topics.sh --create \

--zookeeper localhost:2181 \

--replication-factor 1 --partitions 1 \

--topic flink\_output

bin/kafka-topics.sh --create \

--zookeeper localhost:2181 \

--replication-factor 1 --partitions 1 \

--topic flink\_input

For the sake of this tutorial, we’ll use default configuration and default ports for Apache Kafka.

**3. Flink Usage**

Apache Flink allows a real-time stream processing technology. **The framework allows using multiple third-party systems as stream sources or sinks**.

In Flink – there are various connectors available :

* Apache Kafka (source/sink)
* Apache Cassandra (sink)
* Amazon Kinesis Streams (source/sink)
* Elasticsearch (sink)
* Hadoop FileSystem (sink)
* RabbitMQ (source/sink)
* Apache NiFi (source/sink)
* Twitter Streaming API (source)

To add Flink to our project, we need to include the following Maven dependencies :

<**dependency**>

<**groupId**>org.apache.flink</**groupId**>

<**artifactId**>flink-core</**artifactId**>

<**version**>1.16.1</**version**>

</**dependency**>

<**dependency**>

<**groupId**>org.apache.flink</**groupId**>

<**artifactId**>flink-connector-kafka</**artifactId**>

<**version**>1.16.1</**version**>

</**dependency**>

Adding those dependencies will allow us to consume and produce to and from Kafka topics. You can find the current version of Flink on [Maven Central](https://mvnrepository.com/artifact/org.apache.flink).

**4. Kafka String Consumer**

**To consume data from Kafka with Flink we need to provide a topic and a Kafka address.** We should also provide a group id which will be used to hold offsets so we won’t always read the whole data from the beginning.

Let’s create a static method that will make the creation of *FlinkKafkaConsumer*easier:

**public** **static** FlinkKafkaConsumer011<String> **createStringConsumerForTopic**(

String topic, String kafkaAddress, String kafkaGroup ) {

**Properties** props = **new** **Properties**();

props.setProperty("bootstrap.servers", kafkaAddress);

props.setProperty("group.id",kafkaGroup);

FlinkKafkaConsumer011<String> consumer = **new** **FlinkKafkaConsumer011**<>(

topic, **new** **SimpleStringSchema**(), props);

**return** consumer;

}

This method takes a *topic, kafkaAddress,*and *kafkaGroup* and creates the *FlinkKafkaConsumer* that will consume data from given topic as a *String* since we have used *SimpleStringSchema* to decode data.

The number *011* in the name of class refers to the Kafka version.

**5. Kafka String Producer**

**To produce data to Kafka, we need to provide Kafka address and topic that we want to use.** Again, we can create a static method that will help us to create producers for different topics:

**public** **static** FlinkKafkaProducer011<String> **createStringProducer**(

String topic, String kafkaAddress){

**return** **new** **FlinkKafkaProducer011**<>(kafkaAddress,

topic, **new** **SimpleStringSchema**());

}

This method takes only *topic* and *kafkaAddress* as arguments since there’s no need to provide group id when we are producing to Kafka topic.

**6. String Stream Processing**

When we have a fully working consumer and producer, we can try to process data from Kafka and then save our results back to Kafka. The full list of functions that can be used for stream processing can be found [here](https://www.baeldung.com/apache-flink). In this example, we’re going to capitalize words in each Kafka entry and then write it back to Kafka.

For this purpose we need to create a custom *MapFunction*:

**public** **class** **WordsCapitalizer** **implements** **MapFunction**<String, String> {

@Override

**public** String **map**(String s) {

**return** s.toUpperCase();

}

}

After creating the function, we can use it in stream processing:

**public** **static** **void** **capitalize**() {

**String** inputTopic = "flink\_input";

**String** outputTopic = "flink\_output";

**String** consumerGroup = "baeldung";

**String** address = "localhost:9092";

**StreamExecutionEnvironment** environment = StreamExecutionEnvironment

.getExecutionEnvironment();

FlinkKafkaConsumer011<String> flinkKafkaConsumer = createStringConsumerForTopic(

inputTopic, address, consumerGroup);

DataStream<String> stringInputStream = environment

.addSource(flinkKafkaConsumer);

FlinkKafkaProducer011<String> flinkKafkaProducer = createStringProducer(

outputTopic, address);

stringInputStream

.map(**new** **WordsCapitalizer**())

.addSink(flinkKafkaProducer);

}

**The application will read data from the *flink\_input* topic, perform operations on the stream and then save the results to the*flink\_output*topic in Kafka.**

We’ve seen how to deal with Strings using Flink and Kafka. But often it’s required to perform operations on custom objects. We’ll see how to do this in the next chapters.

**7. Custom Object Deserialization**

The following class represents a simple message with information about sender and recipient:

@JsonSerialize

**public** **class** **InputMessage** {

String sender;

String recipient;

LocalDateTime sentAt;

String message;

}

Previously, we were using *SimpleStringSchema* to deserialize messages from Kafka, but now **we want to deserialize data directly to custom objects**.

To do this, we need a custom *DeserializationSchema:*

**public** **class** **InputMessageDeserializationSchema** **implements**

**DeserializationSchema**<InputMessage> {

**static** **ObjectMapper** objectMapper = **new** **ObjectMapper**().registerModule(**new** **JavaTimeModule**());

@Override

**public** InputMessage **deserialize**(**byte**[] bytes) **throws** IOException {

**return** objectMapper.readValue(bytes, InputMessage.class);

}

@Override

**public** **boolean** **isEndOfStream**(InputMessage inputMessage) {

**return** false;

}

@Override

**public** TypeInformation&lt;InputMessage&gt; getProducedType() {

**return** TypeInformation.of(InputMessage.class);

}

}

We are assuming here that the messages are held as JSON in Kafka.

Since we have a field of type *LocalDateTime*, we need to specify the *JavaTimeModule,*which takes care of mapping *LocalDateTime* objects to JSON.

**Flink schemas can’t have fields that aren’t serializable** because all operators (like schemas or functions) are serialized at the start of the job.

There are similar issues in Apache Spark. One of the known fixes for this issue is initializing fields as *static*, as we did with *ObjectMapper* above. It isn’t the prettiest solution, but it’s relatively simple and does the job.

The method *isEndOfStream* can be used for the special case when stream should be processed only until some specific data is received. But it isn’t needed in our case.

**8. Custom Object Serialization**

Now, let’s assume that we want our system to have a possibility of creating a backup of messages. We want the process to be automatic, and each backup should be composed of messages sent during one whole day.

Also, a backup message should have a unique id assigned. For this purpose, we can create the following class:

**public** **class** **Backup** {

@JsonProperty("inputMessages")

List<InputMessage> inputMessages;

@JsonProperty("backupTimestamp")

LocalDateTime backupTimestamp;

@JsonProperty("uuid")

UUID uuid;

**public** **Backup**(List<InputMessage> inputMessages,

LocalDateTime backupTimestamp) {

this.inputMessages = inputMessages;

this.backupTimestamp = backupTimestamp;

this.uuid = UUID.randomUUID();

}

}

Please mind that the UUID generation mechanism isn’t perfect, as it allows duplicates. However, this is enough for the scope of this example. We want to save our *Backup* object as JSON to Kafka, so we need to create our *SerializationSchema*:

**public** **class** **BackupSerializationSchema** **implements** **SerializationSchema**<Backup> {

ObjectMapper objectMapper;

**Logger** logger = LoggerFactory.getLogger(BackupSerializationSchema.class);

@Override

**public** **byte**[] serialize(Backup backupMessage) {

**if**(objectMapper == null) {

objectMapper = **new** **ObjectMapper**().registerModule(**new** **JavaTimeModule**());

}

**try** {

**return** objectMapper.writeValueAsString(backupMessage).getBytes();

} **catch** (com.fasterxml.jackson.core.JsonProcessingException e) {

logger.error("Failed to parse JSON", e);

}

**return** **new** **byte**[0];

}

}

**9. Timestamping Messages**

Since we want to create a backup for all messages of each day, messages need a timestamp. Flink provides the three different time characteristics *EventTime, ProcessingTime,*and *IngestionTime.* In our case, we need to use the time at which the message has been sent, so we’ll use *EventTime.*

To use *EventTime* **we need a *TimestampAssigner* which will extract timestamps from our input data**:

**public** **class** **InputMessageTimestampAssigner**

**implements** **AssignerWithPunctuatedWatermarks**<InputMessage> {

@Override

**public** **long** **extractTimestamp**(InputMessage element,

**long** previousElementTimestamp) {

**ZoneId** zoneId = ZoneId.systemDefault();

**return** element.getSentAt().atZone(zoneId).toEpochSecond() \* 1000;

}

@Nullable

@Override

**public** Watermark **checkAndGetNextWatermark**(InputMessage lastElement,

**long** extractedTimestamp) {

**return** **new** **Watermark**(extractedTimestamp - 1500);

}

}

We need to transform our *LocalDateTime* to *EpochSecond* as this is the format expected by Flink. After assigning timestamps, all time-based operations will use time from *sentAt* field to operate.

Since Flink expects timestamps to be in milliseconds and *toEpochSecond()* returns time in seconds we needed to multiply it by 1000, so Flink will create windows correctly.

Flink defines the concept of a *Watermark.***Watermarks are useful in case of data that don’t arrive in the order they were sent.** A watermark defines the maximum lateness that is allowed for elements to be processed.

Elements that have timestamps lower than the watermark won’t be processed at all.

**10. Creating Time Windows**

To assure that our backup gathers only messages sent during one day, we can use the *timeWindowAll* method on the stream, which will split messages into windows. However, we’ll still need to aggregate messages from each window and return them as *Backup*. To do this, we’ll need a custom *AggregateFunction*:

**public** **class** **BackupAggregator**

**implements** **AggregateFunction**<InputMessage, List<InputMessage>, Backup> {

@Override

**public** List<InputMessage> **createAccumulator**() {

**return** **new** **ArrayList**<>();

}

@Override

**public** List<InputMessage> **add**(

InputMessage inputMessage,

List<InputMessage> inputMessages) {

inputMessages.add(inputMessage);

**return** inputMessages;

}

@Override

**public** Backup **getResult**(List<InputMessage> inputMessages) {

**return** **new** **Backup**(inputMessages, LocalDateTime.now());

}

@Override

**public** List<InputMessage> **merge**(List<InputMessage> inputMessages,

List<InputMessage> acc1) {

inputMessages.addAll(acc1);

**return** inputMessages;

}

}

**11. Aggregating Backups**

After assigning proper timestamps and implementing our *AggregateFunction*, we can finally take our Kafka input and process it:

**public** **static** **void** **createBackup** () **throws** Exception {

**String** inputTopic = "flink\_input";

**String** outputTopic = "flink\_output";

**String** consumerGroup = "baeldung";

**String** kafkaAddress = "192.168.99.100:9092";

**StreamExecutionEnvironment** environment

= StreamExecutionEnvironment.getExecutionEnvironment();

environment.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

FlinkKafkaConsumer011<InputMessage> flinkKafkaConsumer

= createInputMessageConsumer(inputTopic, kafkaAddress, consumerGroup);

flinkKafkaConsumer.setStartFromEarliest();

flinkKafkaConsumer.assignTimestampsAndWatermarks(

**new** **InputMessageTimestampAssigner**());

FlinkKafkaProducer011<Backup> flinkKafkaProducer

= createBackupProducer(outputTopic, kafkaAddress);

DataStream<InputMessage> inputMessagesStream

= environment.addSource(flinkKafkaConsumer);

inputMessagesStream

.timeWindowAll(Time.hours(24))

.aggregate(**new** **BackupAggregator**())

.addSink(flinkKafkaProducer);

environment.execute();

}

**12. Conclusion**

In this article, we’ve presented how to create a simple data pipeline with Apache Flink and Apache Kafka.