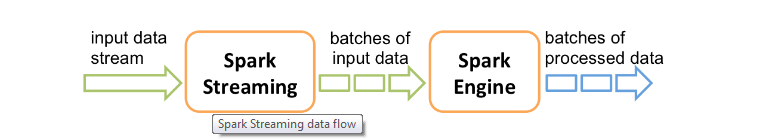
## **Spark Streaming + Kafka Integration Guide**

Before diving in to spark and kafka integration lets first understand spark streaming.

*Spark Streaming :-*

* Spark Streaming is an extension of the core Spark API.
* Enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
* Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
* Processed data can be pushed out to filesystems, databases, and live dashboards.

*How it works :*



Spark receives live data streams called as Dstream from any sources like kafka, flume, Twitter etc and it divides this data in to batches which are then processed by spark engine to generate the final streams of result in batches.

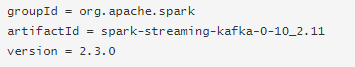
**DStream** is represented as a sequence of [RDDs](https://spark.apache.org/docs/2.2.0/api/scala/index.html#org.apache.spark.rdd.RDD).



In above examples we can see rdd @time1 holds the data during time period 0 to 1 interval and similarly for rdd@time2 holds data for 1 to 2 interval.

*Kafka Integration with spark:*

To link kafka with spark streaming application we need to add dependencies in maven or sbt as-



*Creating a direct stream with kafka examples-*

**

First we need to create the required kakfa parameter that need to pass in to steaming context for creating the kafka stream.

In Kafka parameter we pass the list of brokers,consumers group id and some other parameters.

Then we need to create the steams for collecting the messages from the topics in spark by passing the topic list and the streaming context.

### *LocationStrategies:*

The new Kafka consumer API will pre-fetch messages into buffers. Therefore it is important for performance reasons that the Spark integration keep cached consumers on executors (rather than recreating them for each batch), and prefer to schedule partitions on the host locations that have the appropriate consumers.

Note - we should use **LocationStrategies.PreferConsistent** as shown in above examples. This will distribute partitions evenly across available executors. If your executors are on the same hosts as your Kafka brokers, use **PreferBrokers**, which will prefer to schedule partitions on the Kafka leader for that partition. Finally, if you have a significant skew in load among partitions, use **PreferFixed**. This allows you to specify an explicit mapping of partitions to hosts

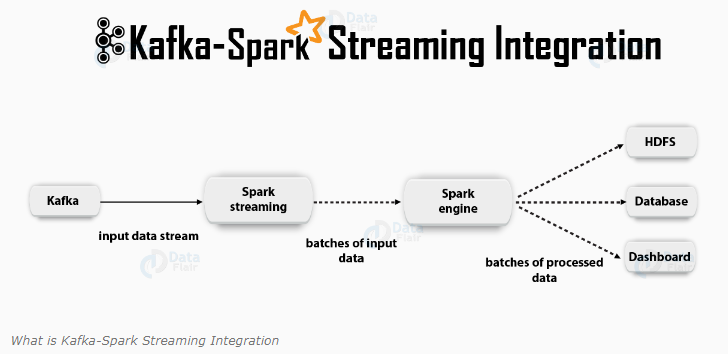
### *Deploying-*

As with any Spark applications, spark-submit is used to launch your application.

For Scala and Java applications, if you are using SBT or Maven for project management, then package spark-streaming-kafka-0-10\_2.11 and its dependencies into the application JAR. Make sure spark-core\_2.11 and spark-streaming\_2.11 are marked as provided dependencies as those are already present in a Spark installation.

*Recommended kafka integration with spark :*

In Apache Kafka Spark Streaming Integration, there are two approaches to configure Spark Streaming to receive data from Kafka i.e. Kafka Spark Streaming Integration. First is by using Receivers and Kafka’s high-level API, and a second, as well as new approach, is without using Receivers.



### **-***Receiver-Based Approach:*

In this approach, we use a Receiver to receive the data. So, by using the Kafka high-level [**consumer**](https://data-flair.training/blogs/kafka-consumer/)API, we implement the Receiver. Further, the received data is stored in [Spark executors](https://data-flair.training/blogs/spark-executor/). Then jobs launched by Kafka – Spark Streaming processes the data.

*Although, it is a possibility that this approach can lose data under failures under default configuration. Hence, we have to additionally enable write-ahead logs in Kafka Spark Streaming, to ensure zero-data-loss. That saves all the received Kafka data into write-ahead logs on a distributed file system synchronously. In this way, it is possible to recover all the data on failure.*

Note – This approach works only with spark version 2.2.0

To link this approach with spark we need to add the dependencies in pom or sbt as below –



### *-Direct Approach (No Receivers)*

After Receiver-Based Approach, new receiver-less “direct” approach has been introduced. It ensures stronger end-to-end guarantees. This approach periodically queries Kafka for the latest offsets in each topic+partition, rather than using receivers to receive data. Also, defines the offset ranges to process in each batch, accordingly. Moreover, to read the defined ranges of offsets from Kafka, it’s simple consumer API is used, especially when the jobs to process the data are launched. However, it is similar to read files from a file system.

*Advantages of Direct Approach-*

### *Parallelism*

Kafka – Spark Streaming will create as many RDD partitions as there are Kafka partitions to consume, with the direct stream. That will read data from Kafka in parallel. Hence, we can say, it is a one-to-one mapping between Kafka and RDD partitions, which is easier to understand and tune.

### *Efficiency*

Achieving zero-data-loss in the first approach required the data to be stored in a write-ahead log, which further replicated the data. This is actually inefficient as the data effectively gets replicated twice – once by Kafka, and a second time by the write-ahead log. The second approach eliminates the problem as there is no receiver, and hence no need for write-ahead logs. As long as we have sufficient Kafka retention, it is possible to recover messages from Kafka.

### *Exactly-Once Semantics*

Basically, we used Kafka’s high-level API to store consumed offsets in Zookeeper in the first approach. However, to consume data from Kafka this is a traditional way. Even if it can ensure zero data loss, there is a small chance some records may get consumed twice under some failures. It happens due to inconsistencies between data reliably received by Kafka – Spark Streaming and offsets tracked by Zookeeper. Therefore, we use a simple Kafka API that does not use Zookeeper, in this second approach. Here, Kafka – Spark Streaming tracks the offsets, through its checkpoints. That removes inconsistencies between Spark Streaming and Zookeeper/Kafka.