

1. Intro Streaming

Exercises

Basic examples

Introduction

In this document, you will find the information required to set up Kafka + Spark Structured Streaming (Scala). All the information and code in this document is focused and tested on macOS, but it can be deployed in any other OS such as Linux or Windows.

Please read carefully the step-by-step.

Content

Introduction	1
Content	2
Prerequisites	4
Apache Spark	4
Install Apache Spark (Scala)	4
Kafka	5
IntelliJ Idea CE	5
Overview	6
Input source	7
Kafka	7
Start Confluent Service:	7
Change directory	8
Create a Kafka Topic	8
List Kafka topic	8
Create Producer to publish data to the Kafka topic	9
Create Consumer to verify the published data	10
Start writing messages	10
Spark Streaming Example	12
Scala	13
Create a new project	13
Add Spark dependencies	14
Create StreaminIntro object	14
Explanation StreamingIntro object	17

Package the code	21
Deploying	22
Kafka + Structured Streaming Spark + Scala	24
Start writing messages	24
Reading messages	25
Some best practices	29
Next steps	29
Troubleshooting	30
Glossary	31
Resources	31

Prerequisites

Apache Spark

You can follow below steps to setup Apache Spark.

- **Install Apache Spark (Scala)**

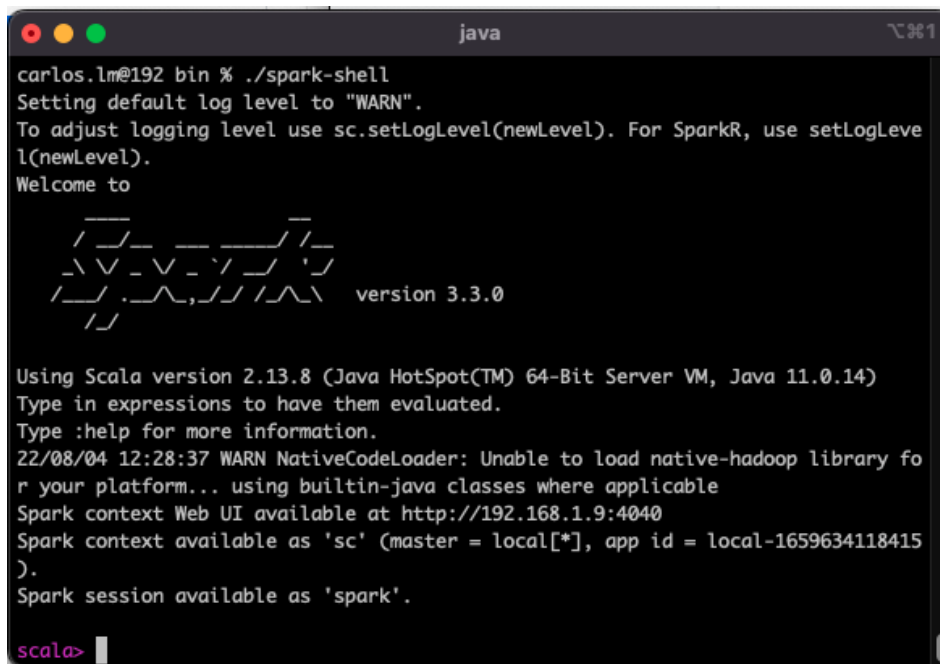
You can follow the next links to install the necessary libraries. Simply select your preferred operating system and follow the instructions.

- [Apache Spark - Installation](#)
- [Steps to Install Apache Spark](#)

- **Verify installation for macOS**

If Spark is installed successfully then you will find the following output when running the following code

```
./spark-shell
```



```
carlos.lm@192 bin % ./spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to

  ____
 /  _ \   _ \   ____
/_  /_\  /  \  /  \
 \_  _/  /_/_/  /_/_/
  /_/

version 3.3.0

Using Scala version 2.13.8 (Java HotSpot(TM) 64-Bit Server VM, Java 11.0.14)
Type in expressions to have them evaluated.
Type :help for more information.
22/08/04 12:28:37 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
Spark context Web UI available at http://192.168.1.9:4040
Spark context available as 'sc' (master = local[*], app id = local-1659634118415)
).
Spark session available as 'spark'.

scala>
```

Kafka

You can follow below steps to setup Kafka.

- **Install Kafka with Confluent Platform**
 - Refer to this [article](#) to use **Docker** and **Docker Compose** to download and run Confluent Platform.
 - Refer to this [article](#) to use a **tar archive** to install Confluent Platform.
 - If you want to test only Kafka in kraft mode (without a Zookeeper) and run tests, refer to this [documentation](#).
- **Verify installation for macOS**
 - Start the Confluent Service by executing this command

```
confluent local services start
```

You should see something like this:

```
carlos.lm@carloslm kafka % confluent local services start

The local commands are intended for a single-node development environment only,
NOT for production usage. https://docs.confluent.io/current/cli/index.html

Using CONFLUENT_CURRENT: /var/folders/wc/0tmqp2mx4j18f_xz05vy832c0000gn/T/confluent.332221
Starting ZooKeeper
ZooKeeper is [UP]
Starting Kafka
Kafka is [UP]
Starting Schema Registry
Schema Registry is [UP]
Starting Kafka REST
Kafka REST is [UP]
Starting Connect
Connect is [UP]
Starting ksqlDB Server
ksqlDB Server is [UP]
Starting Control Center
Control Center is [UP]
```

IntelliJ Idea CE

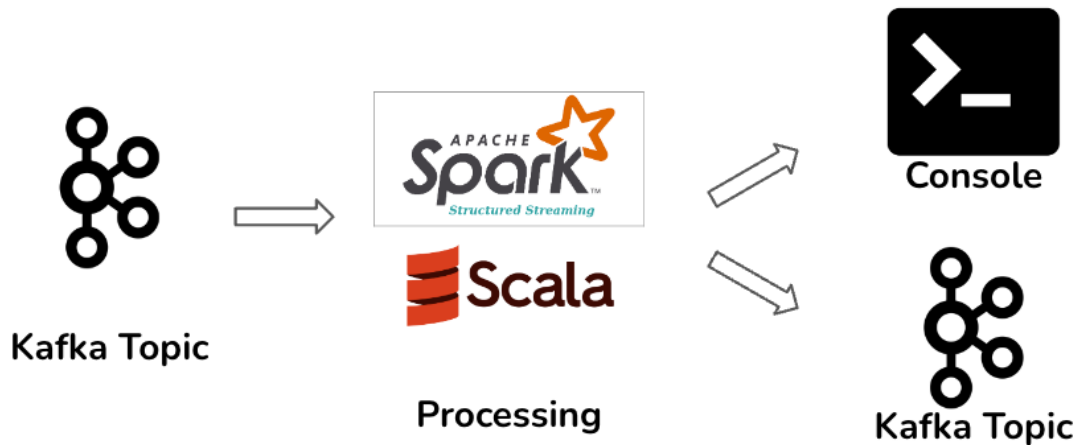
You can follow below steps to setup IntelliJ Idea Community Edition.

- Install IntelliJ Idea CE
 - Please refer to this [website](#)
- Install Scala plugin

- Take a look at this [website](#)
- Verify that the sbt shell is installed.

Overview

The following diagram shows the workflow planned for this Intro to Streaming. The input will be a topic in Kafka, that is feeded with a Kafka subscriber. Then, the messages are going to be processed with Structured Streaming Spark + Scala. The output will be two options: 1) Print in console and 2) Another Kafka topic.



The Confluent Control Center platform will help us to visualize the messages, although they can be displayed with the Console too.

Input source

Spark Streaming ingests data from different types of input sources for processing in real-time. Although there are several sources such as sockets, files, rate, we will see only **Kafka** for this Intro Streaming, that will read data from Apache Kafka and is compatible with Kafka broker versions 0.10.0 or higher.

Kafka

We will read data from Kafka and display it on the console. In order to read data from Kafka, first we need to set up Kafka and publish messages to a Kafka topic which we will then read into Spark Streaming.

- Go to the Confluent directory where you exported the tar archive
- **Start Confluent Service:**
 - Run the following command

```
confluent local services start
```

Expected output

```
carlos.lm@carloslm kafka % confluent local services start
The local commands are intended for a single-node development environment only,
NOT for production usage. https://docs.confluent.io/current/cli/index.html

Using CONFLUENT_CURRENT: /var/folders/wc/0tmqp2mx4j18f_xz05vy832c0000gn/T/confluent.332221
Starting ZooKeeper
ZooKeeper is [UP]
Kafka is [UP]
Starting Schema Registry
Schema Registry is [UP]
Starting Kafka REST
Kafka REST is [UP]
Starting Connect
Connect is [UP]
Starting ksqlDB Server
ksqlDB Server is [UP]
Control Center is [UP]
carlos.lm@carloslm kafka %
```

- **Change directory**

```
cd confluent-7.2.1/bin
```

- **Create a Kafka Topic**

- Run the following command

```
./kafka-topics --create --bootstrap-server  
localhost:9092 --replication-factor 1 --partitions 1  
--topic test
```

Expected output

```
carlos.lm@carloslm bin % ./kafka-topics --create \  
--bootstrap-server localhost:9092 \  
--replication-factor 1 \  
--partitions 1 \  
--topic test  
Created topic test.  
carlos.lm@carloslm bin %
```

- **List Kafka topic**

- Run the following command


```
carlos.lm@carloslm bin % ./kafka-topics \
  --bootstrap-server localhost:9092 \
  --list
__consumer_offsets
__transaction_state
_confluent-command
_confluent-controlcenter-7-2-1-1-AlertHistoryStore-changelog
_confluent-controlcenter-7-2-1-1-AlertHistoryStore-repartition
_confluent-controlcenter-7-2-1-1-Group-ONE_MINUTE-changelog
_confluent_balancer_partition_samples
_schemas
connect-configs
connect-offsets
connect-statuses
default_ksql_processing_log
pageviews
test
testConsumer
users
```

- **Create Producer to publish data to the Kafka topic**

Producers are those client applications that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events.

- Run the code below

```
./kafka-console-producer --broker-list localhost:9092
--topic test
```

Expected output

```
carlos.lm@192 bin % ./kafka-console-producer \
  --broker-list localhost:9092 \
  --topic test
> █
```

- **Create Consumer to verify the published data**

- Open a new tab and change the directory

```
cd confluent-7.2.1/bin
```

- Run the following code

```
./kafka-console-consumer \  
--bootstrap-server localhost:9092 \  
--topic test
```

Expected output

```
carlos.lm@192 bin % ./kafka-console-consumer \  
--bootstrap-server localhost:9092 \  
--topic test  
█
```

- **Start writing messages**

- Write several messages in the **Producer**, press Enter to publish each one.

```
>Hello  
>Wizeline!  
>:)   
>
```

- Go to the **Consumer** and check the messages.

Expected output

```
carlos.lm@192 bin % ./kafka-console-producer \  
  --broker-list localhost:9092 \  
  --topic test  
>Hello  
>Wizeline!  
>:)  
>█
```

Producer

```
carlos.lm@192 bin % ./kafka-console-consumer \  
  --bootstrap-server localhost:9092 \  
  --topic test  
Hello  
Wizeline!  
:)  
█
```

Consumer

Congrats! You have completed the **Input Sources – Kafka** section.

Spark Streaming Example

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

Internally, by default, Structured Streaming queries are processed using a micro-batch processing engine, which processes data streams as a series of small batch jobs thereby achieving end-to-end **latencies as low as 100 milliseconds** and exactly-once fault-tolerance guarantees. However, since Spark 2.3, we have introduced a new low-latency processing mode called Continuous Processing, which can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees.

In short, **Structured Streaming** provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.

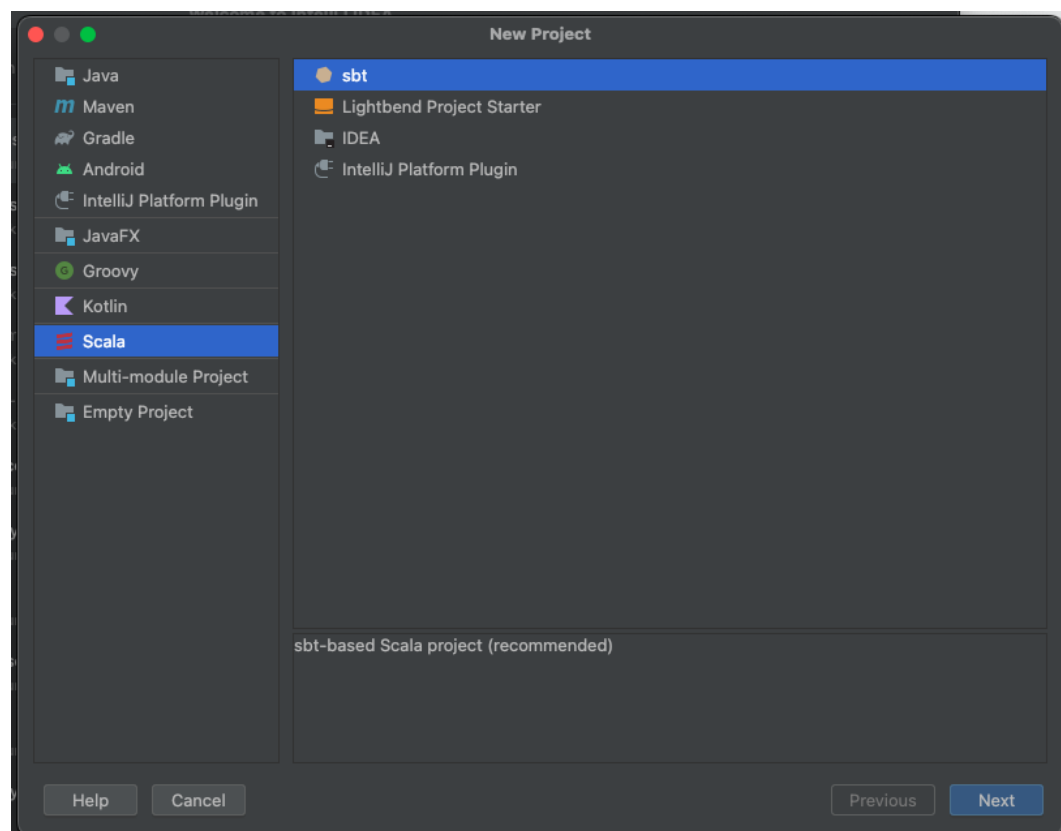
For this example, we will use the [Dataset/DataFrame API](#) in Scala, but it can be also used with Java, Python or R.

Scala

Let's get started with the example. We are using IntelliJ Idea CE to create this example. This IDE will help us to structure the project and let us focus on what matters: our code and its functionality.

Note: Make sure you have installed both IntelliJ Idea CE and Scala plugin before you proceeding.

- **Create a new project**
 - Open the IDE
 - Click “New Project”.
 - Select “Scala” and “sbt”.



- Configure it as follows:
 - Name: “intro-streaming”

- JDK: 11 (we recommend this version)
 - sbt: 1.7.1 (Uncheck “Download sources”)
 - Scala: 2.13.8 (Uncheck “Download sources”)
 - Click “Finish”.
- **Add Spark dependencies**
 - Once you have your project structure created automatically by IntelliJ, go to “target” and open the “build.sbt” file. Replace everything with the following code and save changes:

```
ThisBuild / version := "0.1.0-SNAPSHOT"

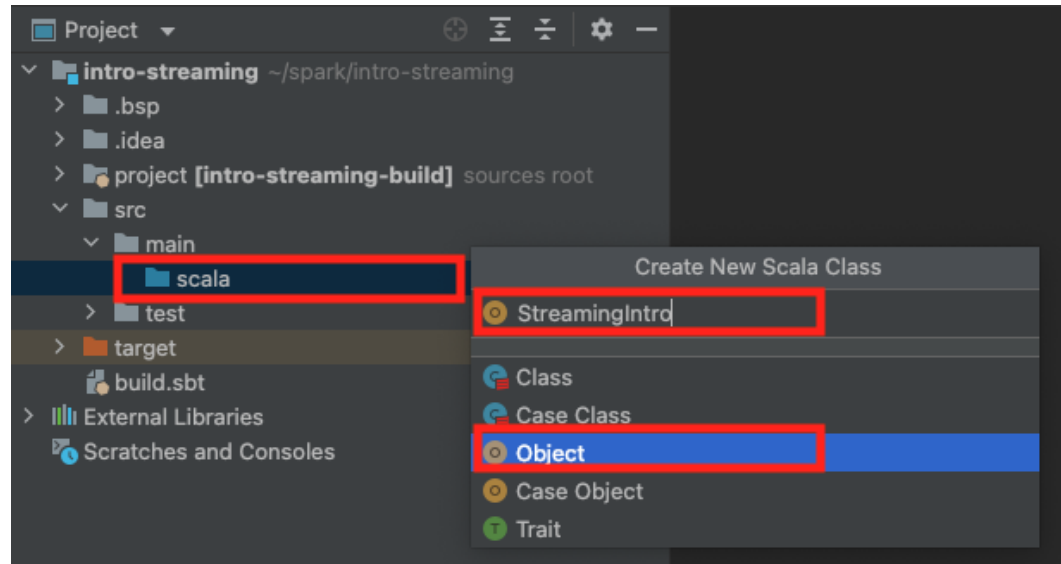
ThisBuild / scalaVersion := "2.13.8"

lazy val root = (project in file("."))
  .settings(
    name := "intro-streaming"
  )

val sparkVersion = "3.3.0"

libraryDependencies += "org.apache.spark" %%
"spark-core" % sparkVersion % "provided"
libraryDependencies += "org.apache.spark" %%
"spark-sql" % sparkVersion % "provided"
```

- **Create StreaminIntro object**
 - Go to the src>main>scala, right click and select “Scala Class”.
 - Then write “StreamingIntro”, and select “Object”.



- Now, replace the generated object with this code. We will see in a moment step by step.

```
import org.apache.spark.sql.Session
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming.OutputMode

object StreamingIntro {
  def main(args: Array[String]): Unit = {
    // Create Spark Session
    val spark = SparkSession
      .builder()
      .master("local")
      .appName("Kafka Source")
      .getOrCreate()

    // Set Spark logging level to ERROR.
    spark.sparkContext.setLogLevel("ERROR")
    import spark.implicits._

    val initDf = spark
      .readStream
      .format("kafka")
```

```

        .option("kafka.bootstrap.servers",
"localhost:9092")
        .option("subscribe", "test")
        .option("failOnDataLoss","false") // Whether to fail
the query when it's possible that data is lost (e.g.,
topics are deleted, or offsets are out of range).
        .load()
        .select(col("value").cast("string"))

    val wordCount = initDf
        .select(explode(split(col("value"), "
")).alias("word"))
        .groupBy("word")
        .count()

    val result =
wordCount.select(to_json(struct($"word",
$"count")).alias(("value")))

    result
        .writeStream
        .outputMode(OutputMode.Update())
        .format("kafka")
        .option("kafka.bootstrap.servers",
"localhost:9092")
        .option("topic", "testConsumer")
        .option("checkpointLocation",
"checkpoint/kafka_checkpoint")
        .start()
        .awaitTermination()
    }
}

```


- Explanation StreamingIntro object

- Import the librariesDependencies

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming.OutputMode
```

- Create Scala object, and the function main

```
object StreamingIntro {
  def main(args: Array[String]): Unit = {
    ...
  }
}
```

- Create the Spark session, and name it "Kafka Source".

```
val spark = SparkSession
  .builder()
  .master("local")
  .appName("Kafka Source")
  .getOrCreate()
```

- Set Spark logging level to ERROR

```
spark.sparkContext.setLogLevel("ERROR")
import spark.implicits._
```

- Create Spark dataframe.

This code reads a stream of data from Kafka and converts it to a Spark DataFrame.

At the end, the code selects the value column and casts it to a string.

```
val initDf = spark
    .readStream
    .format("kafka")
    .option("kafka.bootstrap.servers",
"localhost:9092")
    .option("subscribe", "test")
    .option("failOnDataLoss", "false")
    .load()
    .select(col("value").cast("string"))
```

Optional

```
streaming_df = (
    spark
        .readStream
        .schema(schema)
        .option("maxFilesPerTrigger", 1)
        .csv("/data")
)
```

- Create the word counter that counts the number of times each word appears in a text file.

The code takes a dataframe as an input. The dataframe has one column called "value" and each row contains text. Each word in this dataset will be counted by the code (i.e., if there are two rows with the same word, it will count that twice). Finally, it explodes the dataset into a new column.

```
val wordCount = initDf
    .select(explode(split(col("value"), "
")).alias("word"))
    .groupBy("word")
    .count()
```

- The code counts the number of words in a text file and outputs it to a JSON format.

```
val result = wordCount.select(to_json(struct($"word",  
$"count"))).alias("value"))
```

It's a struct with two fields: "word" and "count". The \$ sign is used to reference the column names.

- The code writes the output of a Spark Structured Streaming query to a Kafka topic.

```
result.writeStream  
  .outputMode(OutputMode.Update())  
  .format("kafka")  
  .option("kafka.bootstrap.servers",  
"localhost:9092")  
  .option("topic", "testConsumer")  
  .option("checkpointLocation",  
"checkpoint/kafka_checkpoint")  
  .start()  
  .awaitTermination()
```

- `.outputMode(OutputMode.Update())`

Output mode determines how the data is processed and pushed to the sink. There are three types:

- **Append:** only the new rows in the streaming DataFrame/Dataset will be written to the sink.
- **Complete:** all the rows in the streaming DataFrame/Dataset will be written to the sink every time there are some updates.

- **Update:** only the rows that were updated in the streaming DataFrame/Dataset will be written to the sink every time there are some updates.
- `.format("kafka")`

The `format("kafka")` is a method that sets the output format of the streaming query to Kafka.
- `.option("kafka.bootstrap.servers", "localhost:9092")`

The option `"kafka.bootstrap.servers"` is the Kafka bootstrap servers to connect to, default port is 9092.
- `.option("topic", "testConsumer")`

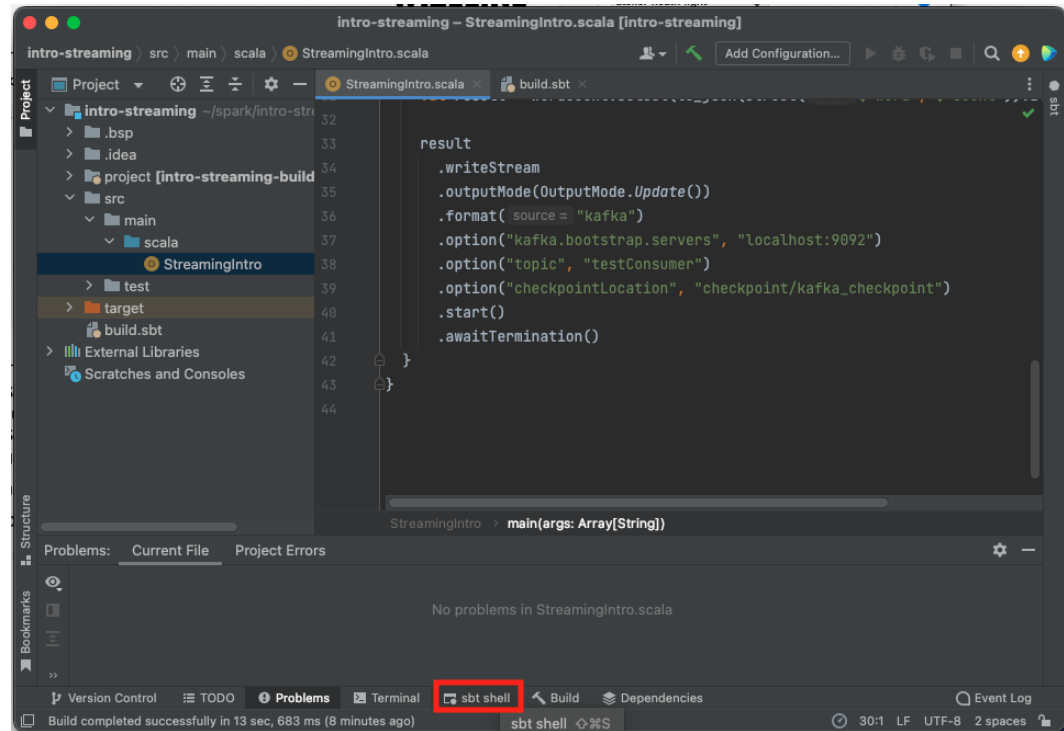
The option(`"topic"`, `"testConsumer"`) sets the topic name to `testConsumer`, that can be automatically created.
- `.option("checkpointLocation", "checkpoint/kafka_checkpoint")`

The option `checkpointLocation` is the path to a directory where Spark Structured Streaming stores checkpoints. A checkpoint is a snapshot of the state of your streaming query at some point in time. This will help the application to be fault tolerant.
- `.start()`

The `start()` method starts the streaming query.
- `.awaitTermination()`

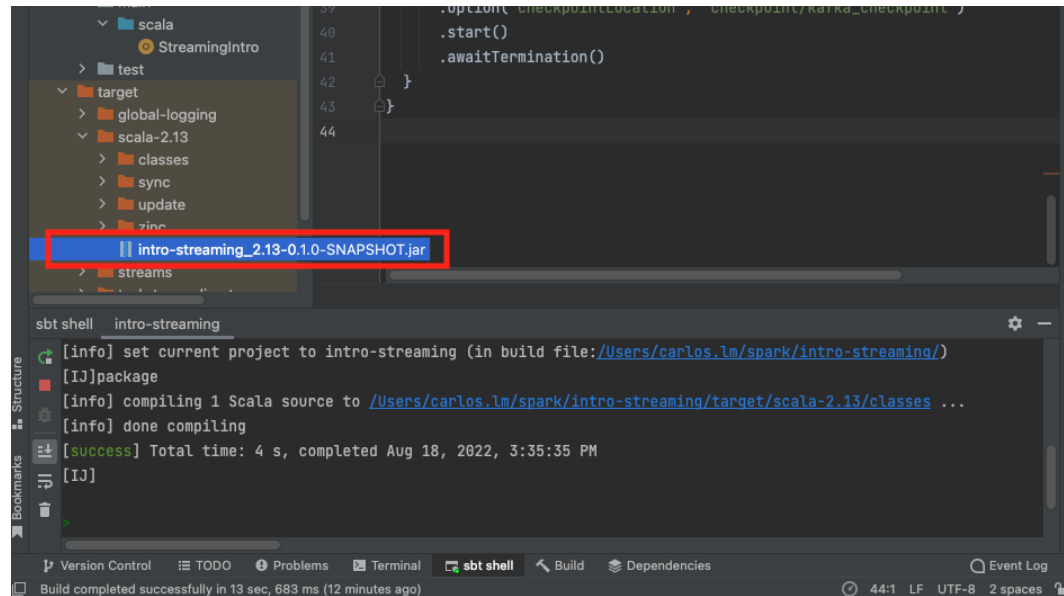
The `awaitTermination()` method is a blocking call that waits for the stream to finish.

- Package the code
 - Open the sbt shell



- Write the command "package", a new jar file will be created

```
package
```



• Deploying

As with any Spark applications, `spark-submit` is used to launch your application. `spark-sql-kafka-0-10_2.13` and its dependencies can be directly added to `spark-submit` using `--packages`.

- Get your jar file path.
- Open a new terminal, change the directory to the folder where you installed Spark. Then go to the “`spark-3.3.0-bin-hadoop3-scala2.13/bin`” folder. You should see something like this when you execute the “`pwd`” command, that prints the current working directory path.

```
carlos.lm@carloslm bin % pwd
/Users/carlos.lm/spark/spark-3.3.0-bin-hadoop3-scala2.13/bin
carlos.lm@carloslm bin %
```

- Run the following command

```
./spark-submit --packages
org.apache.spark:spark-sql-kafka-0-10_2.13:3.3.0
<YOUR-JAR-FILE.jar>
```

Note:

The *spark-submit* command is a utility to run or submit a Spark or PySpark application program (or job) to the cluster by specifying options and configurations, the application you are submitting can be written in Scala, Java, or Python (PySpark). Some of them are the packages, like Kafka; others are the jars, the main class, the driver and executor memory, etc. Feel free to take a look at this [link](#) and the [official documentation](#) to know more about it.

For this example, the command would look like this

```
./spark-submit --packages  
org.apache.spark:spark-sql-kafka-0-10_2.13:3.3.0  
/Users/carlos.lm/spark/intro-streaming/target/scala-2.  
13/intro-streaming_2.13-0.1.0-SNAPSHOT.jar
```

The code runs a spark job that reads from kafka and writes to another kafka topic, as specified in the code.

Important: confirm that you have added the line “org.apache.spark:spark-sql-kafka-0-10_2.13:3.3.0” with those specific versions! Otherwise, the execution could fail.

If everything goes well, you will see an output in your console like this:

```

22/08/19 13:19:54 INFO Executor: Adding file:/private/var/folders/wc/0tmap2mx4j18f_xz05vy832c0000gn/T/spark-816e4354-cba0-499b-b894-0f0f0de5482c/userFiles-Sacd9bef-26a9-4921-afe7-b0edb66ddcaf/org.apache.hadoop_hadoop-client-api-3.3.2.jar to class loader
22/08/19 13:19:54 INFO Executor: Fetching spark://192.168.1.8:64744/jars/org.apache.commons_commons-pool2-2.11.1.jar with timestamp 1660933193308
22/08/19 13:19:54 INFO Utils: Fetching spark://192.168.1.8:64744/jars/org.apache.commons_commons-pool2-2.11.1.jar to /private/var/folders/wc/0tmap2mx4j18f_xz05vy832c0000gn/T/spark-816e4354-cba0-499b-b894-0f0f0de5482c/userFiles-Sacd9bef-26a9-4921-afe7-b0edb66ddcaf/fetchFileTemp15336605334130838336.tmp
22/08/19 13:19:54 INFO Executor: Adding file:/private/var/folders/wc/0tmap2mx4j18f_xz05vy832c0000gn/T/spark-816e4354-cba0-499b-b894-0f0f0de5482c/userFiles-Sacd9bef-26a9-4921-afe7-b0edb66ddcaf/org.apache.commons_commons-pool2-2.11.1.jar to class loader
22/08/19 13:19:54 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 64746.
22/08/19 13:19:54 INFO NettyBlockTransferService: Server created on 192.168.1.8:64746
22/08/19 13:19:54 INFO BlockManager: Using org.apache.spark.storage.RandomBlockReplicationPolicy for block replication policy
22/08/19 13:19:54 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, 192.168.1.8, 64746, None)
22/08/19 13:19:54 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.1.8:64746 with 434.4 MiB RAM, BlockManagerId(driver, 192.168.1.8, 64746, None)
22/08/19 13:19:54 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, 192.168.1.8, 64746, None)
22/08/19 13:19:54 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, 192.168.1.8, 64746, None)

```

What does that mean is that your Scala job is up and running, waiting for incoming messages, and ready to start counting.

Kafka + Structured Streaming Spark + Scala

Once you have configured the Kafka topic and verify that it is working correctly, it is time to combine it with Scala job.

In order to do this, just start writing messages in your producer, you will see that the console that has the Scala job executing is going to start registering incoming messages from the producer, the “test” topic, to be precise.

- **Start writing messages**
 - Once your Scala job is up and running, you can start writing messages in your “test” topic. Go to the console with the topic, and write some messages.

If you’ve already closed it, just re-run it. Now open a new terminal window and run the kafka-console-producer again.

```
./kafka-console-producer --broker-list localhost:9092
--topic test
```



```

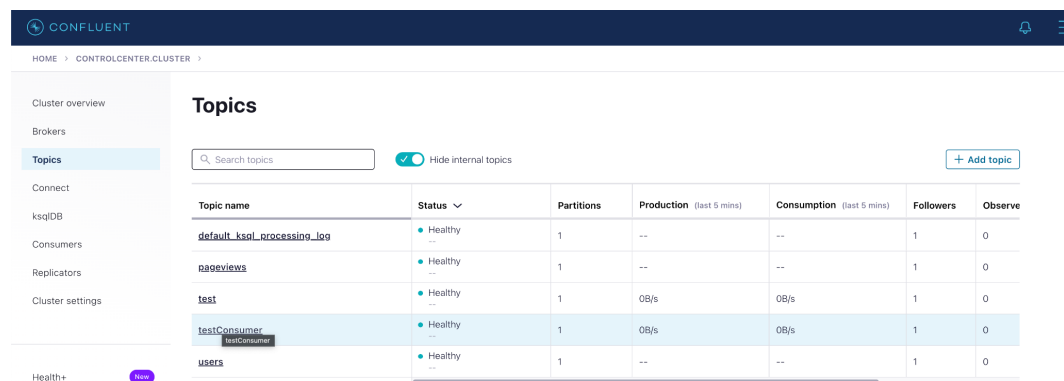
carlos.lm@192 bin % ./kafka-console-producer --broker-list localhost:9092 --topic test
>Hello
>World!
>cat
>dog
>cat
>dog
>wizeline
>streaming
>hello
>cat
>wizeline
>work
>java
>java
>cat
>dog
>dog
>

```

- **Reading messages**

There are several ways to check the count words calculated by the Scala job.

- **Confluent Control Center:** As you remember, this tutorial was created by using the installation of Kafka + Confluent Platform. Just go to <http://localhost:9021/>, click “controlcenter.cluster”, navigate to “Topics”, and you will see some topics. Click on “testConsumer”.



Topic name	Status	Partitions	Production (last 5 mins)	Consumption (last 5 mins)	Followers	Observe
default_ksql_processing_log	Healthy	1	--	--	1	0
pageviews	Healthy	1	--	--	1	0
test	Healthy	1	0B/s	0B/s	1	0
testConsumer	Healthy	1	0B/s	0B/s	1	0
users	Healthy	1	--	--	1	0

In order to see the messages, click “Messages” tab, and click on “Jump to offset”, set the number to “0 / Partition: 0”. You should see the messages there!

testConsumer

The screenshot shows the Kafka console interface for the 'testConsumer' topic. The 'Messages' tab is active. On the left, there are sections for 'Producers' and 'Consumers', both showing 0 bytes/sec. Below these is the 'Message fields' section with a list of fields: topic, partition, offset, timestamp, timestampType, headers, key, and value. The main area displays a list of messages. The first message is a JSON object: {"word": "dog", "count": 4}, with Partition: 0, Offset: 13, and Timestamp: 1660933617779. The second message is {"word": "cat", "count": 4}, with Partition: 0, Offset: 12, and Timestamp: 1660933597457. The third message is {"word": "java", "count": 2}, with Partition: 0, Offset: 11, and Timestamp: 1660933583268. A red box highlights the 'Jump to offset' dropdown menu, which is currently set to '0 / Partition: 0'.

- **Console:** If you chose other Kafka installation, you can access the messages by creating a consumer and set the offset to the earliest.

Run the following command to create the consumer

```
./kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic test \
  --offset 'earliest' \
  --partition 0
```

Then, the console will print the messages.

```

carlos.lm@192 bin % ./kafka-console-consumer \
--bootstrap-server localhost:9092 \
--topic testConsumer \
--offset 'earliest' \
--partition 0
{"word":"Hello","count":1}
{"word":"World!","count":1}
{"word":"dog","count":2}
{"word":"cat","count":2}
{"word":"streaming","count":1}
{"word":"wizeline","count":1}
{"word":"hello","count":1}
{"word":"cat","count":3}
{"word":"work","count":1}
{"word":"wizeline","count":2}
{"word":"java","count":1}
{"word":"java","count":2}
{"word":"cat","count":4}
{"word":"dog","count":4}

```

If you write another message “kafka” in the producer, you will see both in Confluent Platform and Consumer in the console.

testConsumer

The screenshot displays the 'testConsumer' topic page in Confluent Platform. The 'Messages' tab is active, showing a list of messages. A red box highlights the first message, which is a JSON object: {"word": "kafka", "count": 1}. The message details show it is from Partition: 0, Offset: 14, with a Timestamp of 1660937667925. Below it, two other messages are visible: {"word": "dog", "count": 4} (Offset: 13) and {"word": "cat", "count": 4} (Offset: 12). The left sidebar shows the 'Message fields' section with 'value' selected.

```
carlos.lm@192 bin % ./kafka-console-consumer \  
--bootstrap-server localhost:9092 \  
--topic testConsumer \  
--offset 'earliest' \  
--partition 0  
{ "word": "Hello", "count": 1 }  
{ "word": "World!", "count": 1 }  
{ "word": "dog", "count": 2 }  
{ "word": "cat", "count": 2 }  
{ "word": "streaming", "count": 1 }  
{ "word": "wizeline", "count": 1 }  
{ "word": "hello", "count": 1 }  
{ "word": "cat", "count": 3 }  
{ "word": "work", "count": 1 }  
{ "word": "wizeline", "count": 2 }  
{ "word": "java", "count": 1 }  
{ "word": "java", "count": 2 }  
{ "word": "cat", "count": 4 }  
{ "word": "dog", "count": 4 }  
{ "word": "kafka", "count": 1 }
```

And it is done! You have finished the integration with

Some best practices

- You can use Spark Shell to develop and debug!
- The output information can be stored in other formats, take a look at [Output Sinks](#) to know more about it.
- If you feel comfortable with the Console, you can use it to visualize the messages by creating a consumer. Otherwise, the Confluent Control Center is a good option.

Next steps

- Use other tools and frameworks to deploy the same workflow.
- Use files as incoming data.
- Try to write the stream output data to a file like PARQUET or CSV.

Troubleshooting

- **Partition test-0's offset was changed from 16 to 0, some data may have been missed.**
 - **Possible reason:** topics are deleted, or offsets are out of range.
 - **Solution:** add the option "failOnDataLoss" as "false".
 - **Explanation:** Whether to fail the query when it's possible that data is lost (e.g., topics are deleted, or offsets are out of range). This may be a false alarm. You can disable it when it doesn't work as you expected.
- **Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/spark/sql/SparkSession**
 - **Possible reason:** Spark session is not created with the Kafka package.
 - **Solution:** Use the Spark Shell with the package `org.apache.spark:spark-sql-kafka-0-10_2.13:3.3.0`
 - **Explanation:** Your job needs a Spark session in order to execute the Structured Streaming code.
- **Append output mode not supported when there are streaming aggregations on streaming DataFrames/DataSets without watermark.**
 - **Possible reason:** The `outputMode.Append()` is not compatible with the format("console") without a watermark.
 - **Solution:** You will have to implement [watermarking](#), that will be explained in the following sessions.

Glossary

Kafka

- **Topic:** A topic is a log of events. Kafka Topics are Virtual Groups or Logs that hold messages and events in a logical order.
- **Producer:** The Kafka Producer API enables an application to publish a stream of records to one or more Kafka topics.
- **Consumer:** The Kafka Consumer API enables an application to subscribe to one or more Kafka topics. It also makes it possible for the application to process streams of records that are produced to those topics.
- **Bootstrap Server:** It is the url of one of the Kafka brokers which you give to fetch the initial metadata about your Kafka cluster. You can have multiple bootstrap-servers in your producer or consumer configuration. So that if one of the broker is not accessible, then it falls back to other.
- **Partitions:** Within the Kafka cluster, topics are divided into partitions, and the partitions are replicated across brokers. From each partition, multiple consumers can read from a topic in parallel.

Spark Streaming vs Structured Streaming

- **Spark Streaming:** Spark Streaming is a separate library in Spark to process continuously flowing streaming data. It provides us the DStream API which is powered by Spark RDDs. DStreams provide us data divided in chunks as RDDs received from the source of Streaming to be processed and after processing sends it to the destination.
- **Structured Streaming:** Spark 2.x release onwards, Structured Streaming came into the picture. Built on Spark SQL library, Structures Streaming is another way to handle streaming with Spark. This model of streaming is based on Dataframe and Dataset APIs. Hence with this library, we can easily apply any SQL query (using DataFrame API) or scala operations (using DataSet API) on streaming data.
Structured Streaming is more inclined towards real-time streaming but Spark Streaming focuses more on batch processing
-

Resources

- [Structured Streaming Programming Guide](#)
- [Apache Spark Structured Streaming — First Streaming Example \(1 of 6\) | by Neeraj Bhadani | Expedia Group Technology | Medium](#)
- [Structured Streaming + Kafka Integration Guide \(Kafka broker version 0.10.0 or higher\) - Spark 2.0.2 Documentation](#)
- [Spark Submit Command Explained with Examples](#)
- [Apache Kafka® Quick Start - Confluent Cloud](#)
- [Apache Kafka® Quick Start - Docker](#)
- [Apache Kafka® Quick Start - Local Install](#)