1.- Utilizando la sintaxis de Kafka-console-consumer.sh, comprobar que el consumidor puede leer el fichero json enviado por el productor.

1.- Creamos el topic

Bin/Kafka-topics.sh –create –zookeeper localhost:2181 –replication-factor 1 –partitions 1 –topic nametopic

```
root@debian:/home/kafka/kafka_2.11-2.4.0# bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic topicprueba1
Created topic topicprueba1.
```

2.- Guardamos el fichero personal.json en el directorio bin de Kafka.

```
root@debian:/home/kafka/kafka_2.11-2.4.0/bin# ls
connect-distributed.sh                kafka-producer-perf-test.sh
connect-mirror-maker.sh               kafka-reassign-partitions.sh
connect-standalone.sh                 kafka-replica-verification.sh
datojson.json                         kafka-run-class.sh
kafka-acls.sh                         kafka-server-start.sh
kafka-broker-api-versions.sh          kafka-server-stop.sh
kafka-configs.sh                      kafka-streams-application-reset.sh
kafka-console-consumer.sh             kafka-topics.sh
kafka-console-producer.sh             kafka-verifiable-consumer.sh
kafka-consumer-groups.sh              kafka-verifiable-producer.sh
kafka-consumer-perf-test.sh           personal.json
kafka-delegation-tokens.sh            prueba1.json
kafka-delete-records.sh               trogdor.sh
kafka-dump-log.sh                     windows
kafka-leader-election.sh              zookeeper-security-migration.sh
kafka-log-dirs.sh                     zookeeper-server-start.sh
kafka-mirror-maker.sh                 zookeeper-server-stop.sh
kafka-preferred-replica-election.sh   zookeeper-shell.sh
```

3.- Lanzamos el consumidor

```
root@debian:/home/kafka/kafka_2.11-2.4.0# bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topicprueba1 --from-beginning
```

4.- Lanzamos el productor

```
root@debian:/home/kafka/kafka_2.11-2.4.0# cat bin/personal.json | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topicprueba1
```

5 Visualización de datos en tiempo real en el consumidor.

```
{"id": 1,"first_name": "Jeanette","last_name": "Penddreth","email": "jpenddreth0@census.gov","gender": "Female","ip_address": "26.58.193.2"},
{"id": 2,"first_name": "Giavani","last_name": "Frediani","email": "gfrediani1@senate.gov","gender": "Male","ip_address": "229.179.4.212"},
{"id": 3,"first_name": "Noell","last_name": "Bea","email": "nbea2@imageshack.us","gender": "Female","ip_address": "180.66.162.255"},
{"id": 4,"first_name": "Willard","last_name": "Valek","email": "wvalek3@vk.com","gender": "Male","ip_address": "67.76.188.26"}
```

2.- Creando un código en Scala, comprobar que el consumidor puede leer el fichero json enviado por el productor.

En este apartado, se hace lo mismo que en el apartado 1 pero con código en Scala. Introducimos los datos desde el productor en la terminal y son leídos por un consumidor creado por un código en Scala, donde mediante una consulta se mostrarán los datos en la consola.

A continuación, adjunto los recortes de pantalla del código del consumidor en Scala con sus respectivos comentarios.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{col, from_json}
import org.apache.spark.sql.types.{IntegerType, StringType, StructType}

object  lectorjson {
  def main(args:Array[String]):Unit={

    /*Establecemos la conexión utilizando el SparkSession, dando nombre a la app
    y utilizando 2 local para que se produzca el streaming.*/
    val spark=SparkSession.builder().appName( name = "lectorjson").master( master = "local[2]").getOrCreate()

    //Lectura en streaming con formato kafka.
    val df=spark.readStream
      .format( source = "kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("subscribe","topicjson")
      .option("startingOffSets","earliest")
      .load()

    /*El formato kafka no es legible, por lo que debemos hacer un casteo
    de todos los datos para convertirlo en algo legible, en este caso en String.*/
    val res=df.selectExpr( exprs = "CAST(value AS STRING)")

    //Creamos la estructura que queremos que tenga ese esquema
    val schema= new StructType().add( name = "id",IntegerType)
      .add( name = "first_name",StringType)
      .add( name = "last_name",StringType)
      .add( name = "email",StringType)
      .add( name = "gender",StringType)
      .add( name = "ip_address",StringType)


    /*Hacer que el res y el esquema sean legibles. Vamos a recibir
    un casteo de algo que tiene formato json, aplicale el esuqema y dale un alias.*/
    val personal=res.select(from_json(col( colName = "value"),schema).as( alias = "data"))
      .select( col = "data.*")//Consulta para que nos muestre los datos
      print("Mostrar datos por consola")

    //Escribir la consulta que acabamos de realizar por consola
    val query=personal.writeStream
      .format( source = "console")
      .outputMode( outputMode = "append")
      .start()
      .awaitTermination()

  }
}
```

Una vez tenemos el código del consumidor listo, para que la aplicación empiece a funcionar debemos:

1.- Crear un topic. En este caso utilizamos el topicprueba3.

2.- Lanzamos el consumidor en Scala.

3.- Lanzamos el productor desde la terminal, con el fichero que queremos que se muestre en la consola.

```
root@debian:/home/kafka/kafka_2.11-2.4.0# cat bin/personal.json | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topicprueba3
>>>>>root@debian:/home/kafka/kafka_2.11-2.4.0#
```

4. Resultados de la consulta que lanzamos para que nos muestren los datos que leímos del fichero mandado por el productor.

```
+---+----------+---------+--------------------+------+--------------+
| id|first_name|last_name|               email|gender|    ip_address|
+---+----------+---------+--------------------+------+--------------+
|  1|  Jeanette|Penddreth|jpenddreth0@censu...|Female|   26.58.193.2|
|  2|   Giavani| Frediani|gfredianil@senate...|  Male| 229.179.4.212|
|  3|     Noell|      Bea| nbea2@imageshack.us|Female|180.66.162.255|
|  4|   Willard|    Valek|      wvalek3@vk.com|  Male|  67.76.188.26|
+---+----------+---------+--------------------+------+--------------+
```

3.- Ampliar el código del apartado 2 para que, a parte de leer el fichero realice un tratamiento especial de los datos que vamos leyendo. En este caso, queremos que filtre del fichero JSON dos palabras que elegimos cada uno.

En este caso, utilizamos el código del apartado 2 y realizamos una consulta con una transformación para borrar los nombres Noell y Giavani. Por lo que, introduciremos el fichero json al completo y solamente nos mostraran en pantalla los 2 registros que no tienen los nombres anteriores.

A continuación, adjunto los recortes de pantalla del código del consumidor en Scala con sus respectivos comentarios.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{col, from_json}
import org.apache.spark.sql.types.{IntegerType, StringType, StructType}


object  FinalPrueba1 {
  def main(args:Array[String]):Unit={

    /*Establecemos la conexión utilizando el SparkSession, dando nombre a la app
    y utilizando 2 local para que se produzca el streaming.*/
    val spark=SparkSession.builder().appName( name = "Practica1").master( master = "local[2]").getOrCreate()

    //Lectura en streaming con formato kafka.
    val df=spark.readStream
      .format( source = "kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("subscribe","topicprueba4")
      .option("startingOffSets","earliest")
      .load()

    /*El formato kafka no es legible, por lo que debemos hacer un casteo
    de todos los datos para convertirlo en algo legible, en este caso en String.*/
    val res=df.selectExpr( exprs = "CAST(value AS STRING)")

    //Creamos la estructura que queremos que tenga ese esquema
    val schema= new StructType().add( name = "id",IntegerType)
      .add( name = "first_name",StringType)
      .add( name = "last_name",StringType)
      .add( name = "email",StringType)
      .add( name = "gender",StringType)
      .add( name = "ip_address",StringType)

    /*Hacer que el res y el esquema sean legibles. Vamos a recibir
    un casteo de algo que tiene formato json, aplicale el esugema y dale un alias.*/
    val persona=res.select(from_json(col( colName = "value"),schema).as( alias = "data"))
      .select( col = "data.*")
      print("Mostrar datos por consola")

    /*Realizamos una consulta con una transformacion, en este caso con un filter para que nos elimine
    los registros de las siguientes personas*/
    val filtrado= persona.filter(col( colName = "first_name").notEqual( other = "Noell") && col( colName = "first_name").notEqual( other = "Giavani"))

    //Queremos que nos escriba la consulta que acabamos de realizar por consola
    val query=filtrado.writeStream
      .format( source = "console")
      .outputMode( outputMode = "append")
      .start()
      .awaitTermination()
  }
}
```

Una vez tenemos el código del consumidor listo, para que la aplicación empiece a funcionar debemos:

1.- Crear un topic. En este caso utilizamos el topicprueba4.

2.- Lanzamos el consumidor en Scala.

3.- Lanzamos el productor desde la terminal, con el fichero que queremos que se muestre en la consola.

```
>>>>>root@debian:/home/kafka/kafka_2.11-2.4.0# cat bin/personal.json | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topicprueba4
>>>>>root@debian:/home/kafka/kafka_2.11-2.4.0#
```

4. Resultados de la consulta que lanzamos, donde podemos ver que se han eliminado los dos nombre establecidos.

```
+---+----------+---------+--------------------+------+------------+
| id|first_name|last_name|               email|gender|  ip_address|
+---+----------+---------+--------------------+------+------------+
|  1|  Jeanette|Penddreth|jpenddreth0@censu...|Female| 26.58.193.2|
|  4|   Willard|    Valek|      wvalek3@vk.com|  Male|67.76.188.26|
+---+----------+---------+--------------------+------+------------+
```

4.- Parte de investigación. Instalación de zeppelin en la máquina y realizar una consulta para calcular el número de registros introducidos.

1.- Instalación de zeppelin.

Lo primero que debemos hacer es descomprimir el fichero zeppelin-0.8.2-bin-all.tgz:

```
root@debian:/home/keepcoding# tar -xzvf zeppelin-0.8.2-bin-all.tgz
```

Añadimos una nueva entrada en el fichero .bashrc, donde definimos donde se encuentra:

```
root@debian:/home/keepcoding/zeppelin-0.8.2-bin-all# nano ~/.bashrc
```

```
export ZEPPELIN_HOME=/home/keepcoding/zeppelin-0.8.2-bin-all/
export PATH="/home/keepcoding/zeppelin-0.8.2-bin-all/bin/:$PATH"
```

Cargamos las variables definidas anteriormente en el fichero .bashrc:

```
root@debian:/home/keepcoding/zeppelin-0.8.2-bin-all# source ~/.bashrc
```

Por último, cambiamos el puerto por defecto (8080), ya que también es usado por Spark.

```
root@debian:/home/keepcoding/zeppelin-0.8.2-bin-all/conf# nano zeppelin-site.xml
```

```
<property>
  <name>zeppelin.server.port</name>
  <value>8081</value>
  <description>Server port.</description>
</property>
```

Iniciamos Zeppelin:

```
root@debian:/home/keepcoding/zeppelin-0.8.2-bin-all# bin/zeppelin-daemon.sh start
Zeppelin start                                             [  OK  ]
```

Una vez ya estamos en zeppelin en localhost:8081, creamos un nuevo interpreter llamado spark2 para poder lanzar consultas con la versión de spark 2.4.4

Properties

name	value
SPARK_HOME	/home/keepcoding/spark-2.4.4-bin-hadoop2.7
args	

2.- Calcular el número de registros en el fichero amigos.csv

1.- Importamos las librerías necesarias.

```
import org.apache.spark._
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{StringType, StructType, IntegerType}

import org.apache.spark._
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{StringType, StructType, IntegerType}
```

2.- Establecemos la conexión con SparkSession.

```
val spark: SparkSession = SparkSession.builder().appName("schemaCSV").master("local").getOrCreate()

spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@44e063a3
```

3.- El fichero no tiene cabecera, por lo que tenemos que crear un schema para poder darle una. Una vez establecida creamos el dataframe, leemos el fichero con formato csv, delimitado por comas y le decimos que use el esquema creado. Cargamos el fichero, para ello debemos decirle la ubicación del mismo:

```
val schema = new StructType()
      .add("id",IntegerType,true)
      .add("nombre",StringType,true)
      .add("edad",IntegerType,true)
      .add("relacion",IntegerType,true)

val df=spark.read.format("csv").option("delimiter",",").schema(schema).load("file:///home/keepcoding/Descargas/amigos.csv")
```

4.- Mostramos los datos guardados en el dataframe. Como vemos es el fichero amigos.csv con las columnas creadas con el schema.

```
df.show()

+---+--------+----+--------+
| id|  nombre|edad|relacion|
+---+--------+----+--------+
|  0|    Will|  33|     385|
|  1|Jean-Luc|  26|       2|
|  2|    Hugh|  55|     221|
|  3|  Deanna|  40|     465|
|  4|   Quark|  68|      21|
|  5|  Weyoun|  59|     318|
|  6|  Gowron|  37|     220|
|  7|    Will|  54|     307|
|  8|  Jadzia|  38|     380|
|  9|    Hugh|  27|     181|
| 10|     Odo|  53|     191|
| 11|     Ben|  57|     372|
| 12|   Keiko|  54|     253|
| 13|Jean-Luc|  56|     444|
```

5.- Mediante una consulta lanzada con sparq sql, realizamos un count para ver cuántos registros tenemos.

```
val df1 = spark.sql("SELECT COUNT(*) from csv.`file:///home/keepcoding/Descargas/amigos.csv`")
df1.show()

+--------+
|count(1)|
+--------+
|     500|
+--------+
```

Adjunto enlace de zeppelin con el código:

http://localhost:8081/#/notebook/2F224SU2G