



Big Data Processing. Spark y Scala

Big Data Machine Learning Bootcamp

1.- SPARK STREAMING. KAFKA:

Vamos a crear una estructura de Kafka STREAMING. Para ello vamos a usar la estructura de código PRODUCTOR y código CONSUMIDOR.

Trabajo que debe desarrollar el PRODUCTOR: deberá mandar el contenido de un JSON que se os facilita para que el consumidor que lo escuche pueda tratarlo.

Para la creación de dicho productor se os pedirá que lo hagáis usando la sintaxis vista en clase con el método Kafka console producer.sh.

Trabajo que debe desarrollar el CONSUMIDOR: deberá tener dos entornos distintos de trabajo, para poder chequear que Kafka funciona correctamente:

1. Usando la sintaxis de Kafka console consumer.sh. En éste caso solo queremos comprobar que nuestro consumidor puede leerlo sin problema.
2. Creando un código en Scala donde vamos a ampliar nuestro trabajo. Deberá leerlo igual que en el caso anterior, pero en éste caso también deberá hacer un tratamiento especial de los datos que vayamos leyendo: queremos que filtre (que no aparezcan) del fichero JSON dos palabras que elijáis cada uno.

En este caso vamos a trabajar con el entorno de desarrollo integrado IntelliJ IDEA y como hemos visto en clase, para poder utilizar de forma correcta kafka streaming (también sql-kafka) tenemos que asegurarnos de tener añadidas las dependencias de librería necesarias en el fichero build.sbt de nuestro proyecto:

```
TestScalaNuevo [~/IdeaProjects/TestScalaNuevo] - .../build.sbt [TestScalaNuevo]
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
TestScalaNuevo build.sbt
Project TestScalaNuevo ~/IdeaProjects/TestScalaNuevo
├── .idea
├── project [TestScalaNuevo]
├── spark-warehouse
└── src
    ├── main
    ├── resources
    └── scala
        ├── casefichero
        ├── Consumidor
        └── Ejemplo
└── build.sbt
    1 name := "TestScalaNuevo"
    2
    3 version := "0.1"
    4
    5 scalaVersion := "2.11.12"
    6
    7
    8 libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.4.0" % "Provided"
    9 libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.4.0"
    10 libraryDependencies += "org.apache.spark" %% "spark-sql-kafka-0-10" % "2.4.0"
```

A continuación hay que lanzar el servicio de zookeeper y de Kafka. Para ello usamos dos terminales distintos de LINUX donde ejecutamos los siguientes comandos:

```
root@debian:/home/kafka/kafka_2.11-2.4.0# bin/zookeeper-server-start.sh
config/zookeeper.properties
```

```
root@debian:/home/kafka/kafka_2.11-2.4.0# bin/kafka-server-start.sh
config/server.properties
```

En Kafka, las comunicaciones se producen a través de temas (topic) a los que tienen que estar suscritos tanto el productor como el consumidor. De esta forma el siguiente paso es crear dicho tema, y lo hacemos con el siguiente comando:

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

1.1 PRODUCTOR (terminal)

Nuestro productor va a mandar el contenido de un fichero json al tema que acabamos de crear. Dicho archivo es el “personal.json” que está dentro del directorio de la entrega de esta práctica.

Lo ejecutamos desde el terminal con el siguiente método:

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

Es recomendable tener activado antes el consumidor para poder ver cómo recibe la información en streaming.

1.2 CONSUMIDOR (terminal)

Por otro lado, en un terminal distinto, lanzamos el consumidor. Allí lo suscribimos al mismo tema con el que estamos trabajando. Para ello ejecutamos el siguiente comando:

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

Como podemos ver a continuación recibimos sin problemas la información mandada por el productor, por lo que podemos asegurar que la conexión en Kafka se ha establecido de forma correcta.

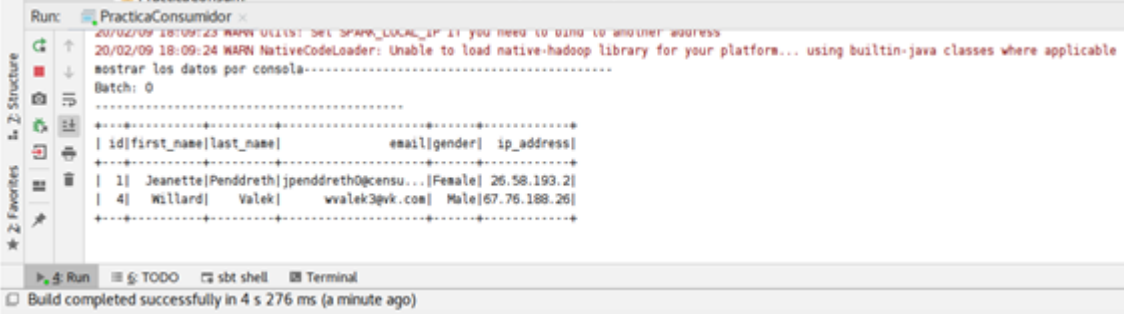
```
root@debian:/home/kafka/kafka_2.11-2.4.0# bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic PracticaKafka
Created topic PracticaKafka.
root@debian:/home/kafka/kafka_2.11-2.4.0# bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic PracticaKafka --from-beginning
{"id":1,"first_name":"Jeanette","last_name":"Pendreth","email":"jpendreth@census.gov","gender":"Female","ip_address":"26.58.193.2"},
{"id":2,"first_name":"Giovanni","last_name":"Frediani","email":"gfrediani@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":"188.66.162.255"},
{"id":4,"first_name":"Willard","last_name":"Valek","email":"wvalek3@vk.com","gender":"Male","ip_address":"67.76.188.26"}
```

1.3 CONSUMIDOR (scala) Y FILTRADO

Ahora vamos a crear un código en scala donde vamos a implementar nuestro consumidor. La idea es que lea el mensaje de igual forma que hemos hecho antes y además vamos a añadir un procesado extra de la información recibida. En concreto vamos a eliminar dos palabras, que en nuestro caso son el first name 'Giavani' y el last name 'Bea'. El código (también adjunto en el directorio de la entrega) es el siguiente:

```
PracticaContarRegistros.scala x build.sbt x PracticaConsumidor.scala x
1 //importamos los paquetes necesarios para la ejecución del código
2 import org.apache.spark.sql.SparkSession
3 import org.apache.spark.sql.types.{IntegerType, StringType, StructType}
4 import org.apache.spark.sql.functions.{col, from_json}
5
6 //nuestro objeto Consumidor para la práctica
7 object PracticaConsumidor {
8   def main(args:Array[String]): Unit = {
9
10     //creamos la sparkSession
11     val spark = SparkSession.builder().appName( name = "kafkajson").master( master = "local[2]").getOrCreate()
12
13     //leemos un formato kafka, definimos la conexión al clúster, nos subscribimos al tema del que queremos leer
14     //y empezamos a leer desde el principio
15     val df = spark.readStream
16       .format( source = "kafka")
17       .option("kafka.bootstrap.servers","localhost:9092")
18       .option("subscribe","PracticaKafka")
19       .option("startingOffsets","earliest")
20       .load()
21
22     //casteamos todos los datos leídos en formato kafka para convertirlos en strings
23     val res = df.selectExpr( exprs = "CAST(value AS STRING)")
24
25     //creamos el schema según el formato de nuestro fichero
26     val schemaPersonal = new StructType().add( name = "id", IntegerType, nullable = true)
27       .add( name = "first_name", StringType, nullable = true)
28       .add( name = "last_name", StringType, nullable = true)
29       .add( name = "email", StringType, nullable = true)
30       .add( name = "gender", StringType, nullable = true)
31       .add( name = "ip_address", StringType, nullable = true)
32
33     //relacionamos los datos leídos del tema de Kafka con el schema
34     //y eliminamos las dos palabras elegidas
35     val persona = res.select(from_json(col( colName = "value"), schemaPersonal).as ( alias = "data"))
36       .select( col = "data.*")
37       .filter( conditionExpr = "data.first_name != 'Giavani'" )
38       .filter( conditionExpr = "data.last_name != 'Bea'" )
39
40     //mostramos por consola añadiendo los datos según vayan llegando
41     print("mostrar los datos por consola")
42
43     //escribimos por consola añadiendo la información según la vamos recibiendo
44     //y dejamos el consumidor a la espera de que siga llegando información al tema suscrito
45     persona.writeStream
46       .format( source = "console")
47       .outputMode( outputMode = "append")
48       .start()
49       .awaitTermination()
50   }
51 }
52
53
PracticaConsumidor > main(args: Array[String])
```

El resultado obtenido es el siguiente:



```
Run: PracticaConsumidor x
20/02/09 18:09:23 WARN UtilList: Set SPARK_LOCAL_IP if you need to bind to another address
20/02/09 18:09:24 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
mostrar los datos por consola-----
Batch: 0
-----
+-----+-----+-----+-----+
| id|first_name|last_name|email|gender| ip_address|
+-----+-----+-----+-----+
| 1| Jeanette|Pendreth|jpendreth@censu...|Female| 26.58.193.2|
| 4| Willard| Valek|wvalek3gvk.com| Male|67.76.188.26|
+-----+-----+-----+-----+
```

Run | TODO | sbt shell | Terminal

Build completed successfully in 4 s 276 ms (a minute ago)

2.- ZEPPELIN

Hay que instalar Zeppelin en la máquina virtual usada durante el módulo y configurar SPARK para que se pueda trabajar en el note de Zeppelin.

Una vez montado hay que utilizar el archivo “amigos.csv” facilitado y calcular el número de registros que tiene.

En la máquina virtual ya tenemos descargado Zeppelin así que el primer paso es descomprimirlo. Para ello usamos el siguiente comando:

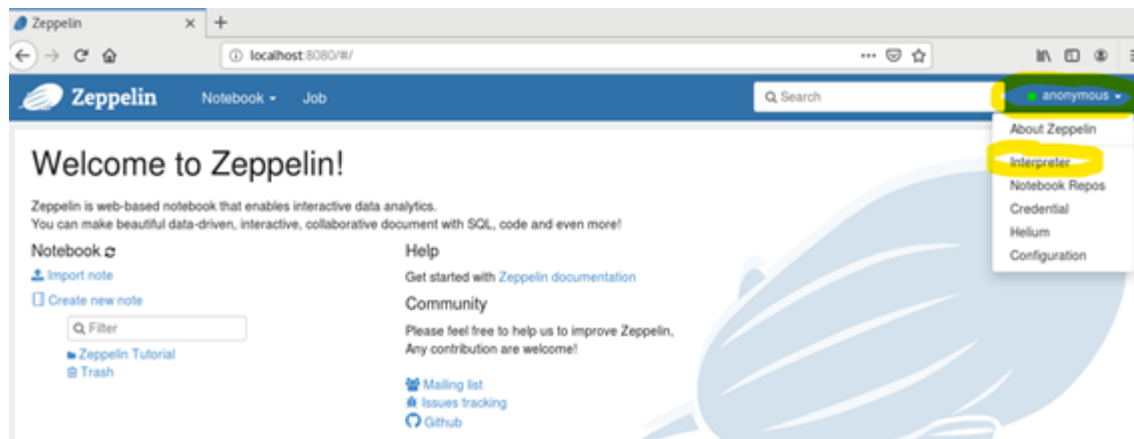
```
root@debian:/home/keepcoding# tar -xvzf zeppelin-0.8.2-bin-all.tgz
```

A continuación entramos en el directorio descomprimido y arrancamos Zeppelin:

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

Ahora desde el navegador comprobamos que lo tenemos arrancado en el localhost puerto 8080. Para verificar que está todo bien debemos tener el usuario anonymous y la luz en verde.

El siguiente paso es asociarlo al spark que tenemos instalado en la máquina, en nuestro caso spark2.4.4. Para ello vamos al menú interpreter dentro del desplegable anonymous:



A continuación creamos uno nuevo que lo vamos a llamar por ejemplo spark24 y lo guardamos:

The first screenshot shows the Zeppelin 'Interpreters' page. The 'Create' button in the top right corner is highlighted with a yellow circle. Below the header, there is a search bar and a list of existing interpreters: 'alluxio' and 'angular'. Each interpreter entry has 'edit', 'restart', and 'remove' buttons.

The second screenshot shows the 'Create new interpreter' form. The form is highlighted with a yellow circle. It contains the following fields:

- Interpreter Name:** A text input field containing 'spark24'.
- Interpreter group:** A dropdown menu with 'spark' selected.

Ahora lo buscamos en la lista de interpreters y lo editamos. Dentro del apartado properties añadimos la ruta donde tenemos nuestro spark en la máquina (SPARK_HOME):

This screenshot shows the Zeppelin 'Interpreters' page with the 'spark24' interpreter selected in the search bar. The 'spark24' entry in the list has an 'edit' button highlighted with a yellow circle. The configuration for 'spark24' is shown below, including the 'Option' section with a dropdown set to 'process' and checkboxes for 'Connect to existing process' and 'Set permission'.

Zeppelin

localhost:8080/#/interpreter

zeppelin.spark.concurrentSQL	<input type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.enableSupportedVersionCheck	<input checked="" type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.importImplicit	<input checked="" type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.maxResult	1000	<input type="button" value="x"/>
zeppelin.spark.printREPLOutput	<input checked="" type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.sql.interpolation	<input type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.sql.stacktrace	<input type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.ui.hidden	<input type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.ui.WebUrl		<input type="button" value="x"/>
zeppelin.spark.useHiveContext	<input checked="" type="checkbox"/>	<input type="button" value="x"/>
zeppelin.spark.useNew	<input checked="" type="checkbox"/>	<input type="button" value="x"/>
SPARK_HOME	/home/keepcoding/spark-2.4.4-bin-hadoop2.7	<input type="button" value="x"/>

Dependencies

These dependencies will be added to classpath when interpreter process starts.

artifact	exclude	action
group:artifactId:version or local file path	(Optional) comma separated group:artifactId list	<input type="button" value="+"/>

Para asegurarnos entramos de nuevo para ver que ya lo tenemos añadido en properties y reiniciamos el interpreter:

Zeppelin

Notebook Job

Search

anonymous

Interpreters

Manage interpreters settings. You can create / edit / remove settings. Note can bind / unbind these interpreter settings.

spark24

spark24 %spark24, %sql, %idep, %pyspark, %ipySpark, %lr

Option

The interpreter will be instantiated Globally in shared process

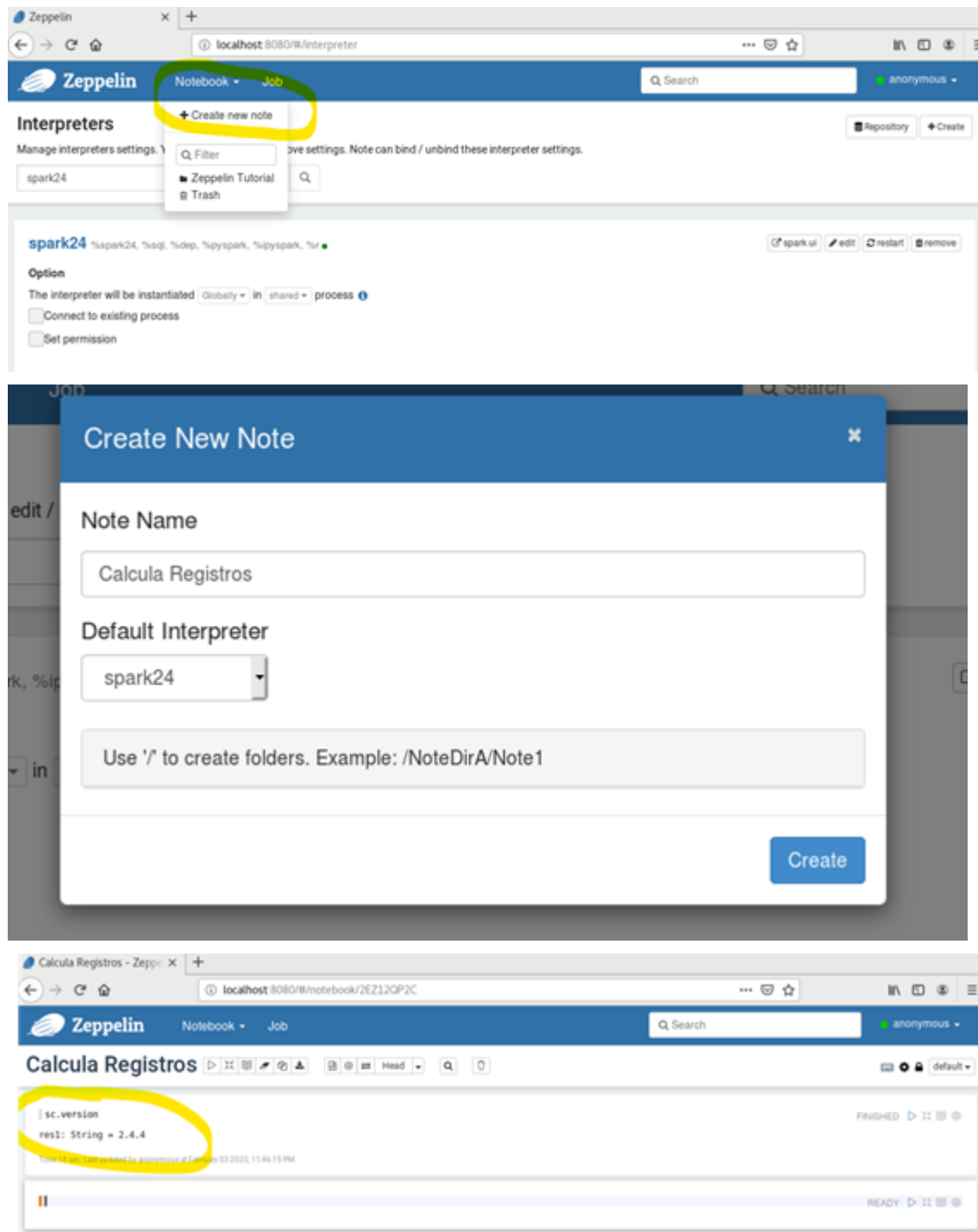
☐ Connect to existing process

☐ Set permission

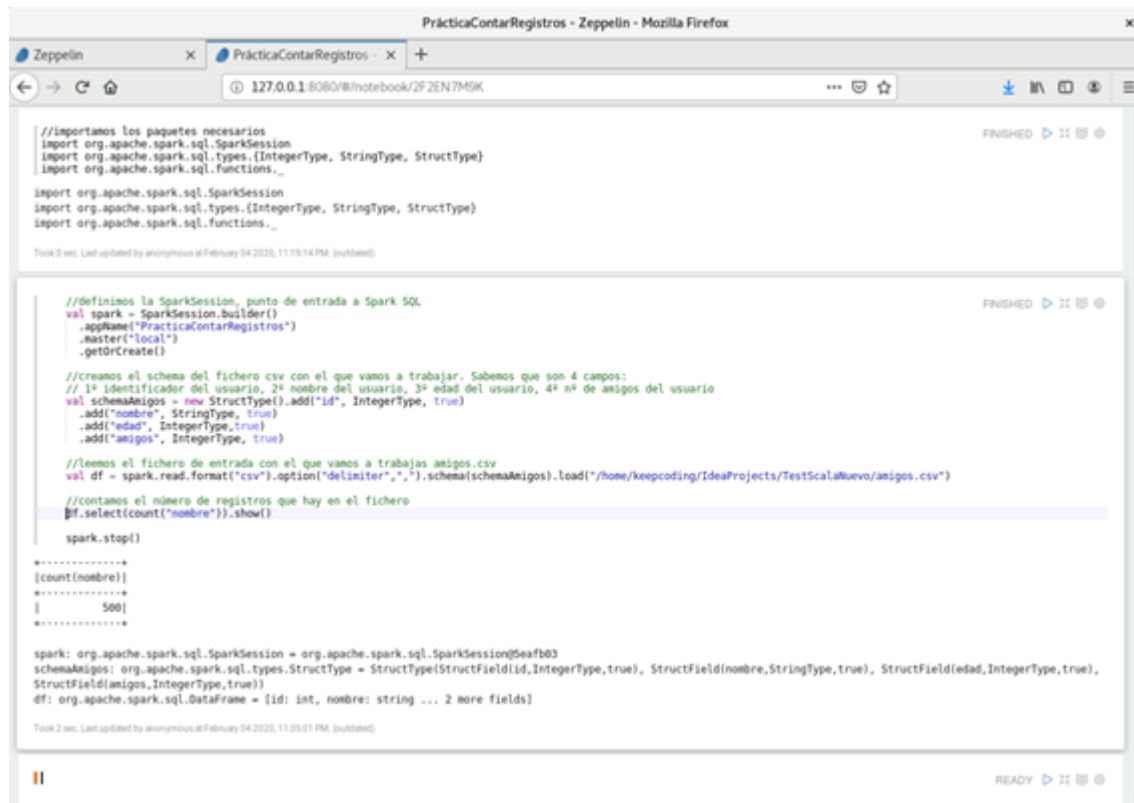
Properties

name	value
SPARK_HOME	/home/keepcoding/spark-2.4.4-bin-hadoop2.7
args	
master	local[*]

De esta forma ya tenemos vinculado nuestro Spark con Zeppelin, así que vamos a comprobar que todo funciona correctamente. Para ello creamos una nueva nota en el desplegable Notebook y lanzamos el comando `sc.version`:



Por último, sobre este note vamos a realizar el ejercicio que se nos pide en la práctica. Vamos a cargar el fichero que se nos facilita (amigos.csv, adjunto en el directorio de entrega) y calculamos el número de registros que tiene.



```
//importamos los paquetes necesarios
import org.apache.spark.sql.{SparkSession, IntegerType, StringType, StructType}
import org.apache.spark.sql.functions._

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

//definimos la SparkSession, punto de entrada a Spark SQL
val spark = SparkSession.builder()
  .appName("PracticaContarRegistros")
  .master("local")
  .getOrCreate()

//creamos el schema del fichero csv con el que vamos a trabajar. Sabemos que son 4 campos:
// 1º identificador del usuario, 2º nombre del usuario, 3º edad del usuario, 4º nº de amigos del usuario
val schemaAmigos = new StructType().add("id", IntegerType, true)
  .add("nombre", StringType, true)
  .add("edad", IntegerType, true)
  .add("amigos", IntegerType, true)

//leemos el fichero de entrada con el que vamos a trabajar amigos.csv
val df = spark.read.format("csv").option("delimiter", ";").schema(schemaAmigos).load("/home/keepcoding/IdeaProjects/TestScalaNuevo/amigos.csv")

//contamos el número de registros que hay en el fichero
df.select(count("nombre")).show()

spark.stop()

+-----+
|count(nombre)|
+-----+
|          500|
+-----+

spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@5eafb03
schemaAmigos: org.apache.spark.sql.types.StructType = StructType(StructField(id,IntegerType,true), StructField(nombre,StringType,true), StructField(edad,IntegerType,true), StructField(amigos,IntegerType,true))
df: org.apache.spark.sql.DataFrame = [id: int, nombre: string ... 2 more fields]
```

Ante la duda de si el pantallazo anterior es legible, copiamos el código a continuación:

```
//importamos los paquetes necesarios

import org.apache.spark.sql.SparkSession

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

import org.apache.spark.sql.functions._

//definimos la SparkSession, punto de entrada a Spark SQL

val spark = SparkSession.builder()

  .appName("PracticaContarRegistros")

  .master("local")

  .getOrCreate()

//creamos el schema del fichero csv con el que vamos a trabajar. Sabemos que son 4 campos:

// 1º identificador del usuario, 2º nombre del usuario, 3º edad del usuario, 4º nº de amigos del usuario
```

```
val schemaAmigos = new StructType().add("id", IntegerType, true)

.add("nombre", StringType, true)

.add("edad", IntegerType, true)

.add("amigos", IntegerType, true)

//leemos el fichero de entrada con el que vamos a trabajar amigos.csv

val df =
spark.read.format("csv").option("delimiter", ";").schema(schemaAmigos).load("/home/keepcoding/IdeaProjects/TestScalaNuevo/amigos.csv")

//contamos el número de registros que hay en el fichero

df.select(count("nombre")).show()

spark.stop()
```