Spark Streaming Project Report

Link to source code on GitHub: https://github.com/isbainemohamed/SparkStreamingProject

Done by:

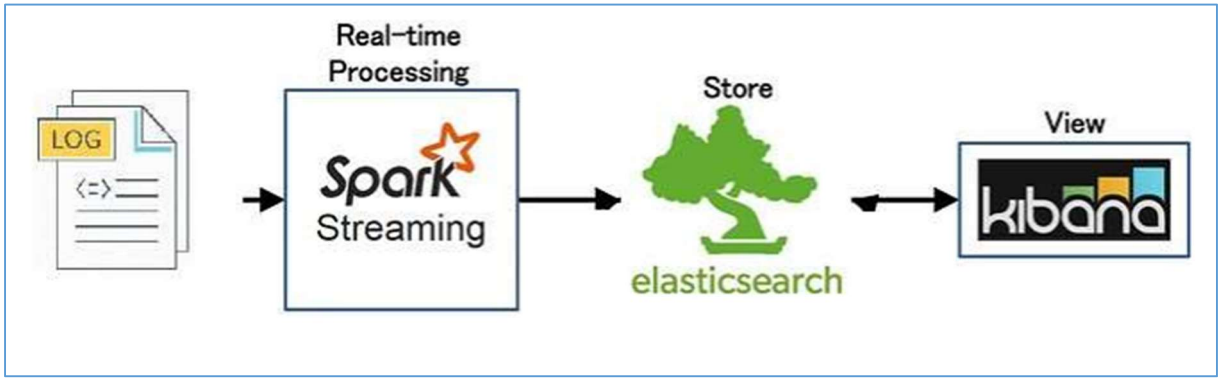
- *ISBAINE Mohamed*
- *JAWHAR Mohammed*
- *SNOUSSI Amine*

Coached by:

- *Ziyati El Houssine*

Academic year 2021-2022

The objective of this project is to inject the data coming from the logs with **Spark Streaming** and return the result in the form of a Dashboard under **Kibana**, as shown in the following figure:



1. Used technologies:

- Scala:

What is Scala?

Scala means scalable language. It is an object oriented programming language multi-paradigm. The Scala language includes features of functional programming and object-oriented programming (in Scala, each value is an object.). It is a statically typed language. Its source code is compiled to bytecode and executed by the Java Virtual Machine (JVM).

Scala is used in data processing, distributed computing and Web development. It powers the data engineering infrastructure of many companies. It is designed to grow with the demands of its user, from writing small scripts to building a massive system of data processing.

Scala uses Java Virtual Machine (JVM) to execute bytecode. Its code is compiled to bytecode and executed by Java Virtual Machine. So you don't need only JVM to start development with. Scala can also use all Java classes and allows us to create our custom class.

The Scala language is mainly used by software engineers and data engineers. You will see some data scientists using it with Apache Spark to process huge data.

How is Scala different from Java?

- Nested functions – This allows us to define a function inside from another function.
- Closures – A function whose return value depends on the variables declared outside the function.
- Each value is an object.
- Each operation is a method call.

- Spark Streaming:

What is Spark Streaming?

Spark Streaming is an extension to *Spark's core API* that allows a scalable, high-throughput, fault-tolerant stream processing of data streams live. Data can be ingested from many sources such as Kafka, Kinesis or TCP sockets, and can be occupied using algorithms complexes expressed with high-level functions such as map, reduce, join and window. Finally, data can be masked to systems files, databases and live dashboards. In fact, you can apply the machine learning and processing algorithms of Spark charts on data streams.

Internally it works as follows. Spark Streaming receives streams from live input data and divides the data into batches, which are then processed by the Spark engine to generate the final result stream in batches.

Spark Streaming provides a high-level abstraction called discretized streaming or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka and Kinesis, either by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

- Elastic Search:

What is Elastic Search?

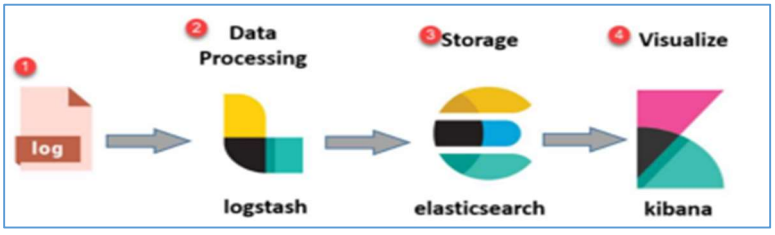
Elasticsearch is a free and open distributed search and analytics engine for any type of data, including textual, numeric, geospatial, structured and unstructured. Elasticsearch was built from of Apache Lucene and was launched in 2010 by Elasticsearch N. V. (now called Elastic). Renowned for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the core component of the Elastic Stack, a set of free and open tools for data ingestion, enrichment, storage, analysis and visualization. Commonly called the ELK Suite (for Elasticsearch, Logstash, and Kibana), the Elastic Stack now includes a rich collection of lightweight transfer agents, called Beats Agents, for sending data to Elasticsearch.

How does Elasticsearch work?

You can send data to Elasticsearch in the form of JSON documents using the API or ingest tools such as Logstash and Amazon Kinesis Firehose. Elasticsearch automatically stores the original document and adds a searchable reference to the document in the cluster index. We can then search and retrieve the document using the Elasticsearch API. We can also use Kibana, a visualization tool, with Elasticsearch to visualize your data and build interactive dashboards

What is the main use of Elasticsearch?

Elasticsearch is an open source full-text search and analytics engine highly scalable source. It allows you to store, search and analyze large volumes of data quickly and in near real time. It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.



- Kibana:

What is Kibana?

Kibana is a data visualization and exploration tool, used for log and time series analytics, application monitoring, and operational intelligence. It offers powerful and easy-to-use features such as histograms, line charts, pie charts, heat maps, and built-in geospatial support. It also provides tight integration with Elasticsearch, a popular search and analytics engine, making Kibana the default choice for viewing data stored in **Elasticsearch**.

What are the advantages of Kibana?

➔ **INTERACTIVE CHARTS:**

Kibana offers intuitive charts and reports that you can use to interactively navigate large amounts of log data. You can dynamically drag windows timelines, zoom in and out on subsets of specific data, and explore reports to extract insights usable from your data.

➔ **MAPPING SUPPORT:**

Kibana comes with powerful geospatial features that allow you to allow information to be layered seamlessly maps to your data and visualize the results on maps.

➔ **PRE-INTEGRATED AGGREGATIONS AND FILTERS:**

With **Kibana's** pre-built aggregations and filters, you can run various analytics such as histograms, queries "top N" and trends in just a few clicks.

➔ **EASY ACCESS DASHBOARDS:**

You can easily configure *dashboards* and reports and share them with other people. You only need a browser to view and explore the data.

- SBT:

What is sbt?

sbt is an open source build tool for Scala and Java projects, similar to Apache's Maven and Ant. Its main features are the following:

o Native support for compiling Scala code and integration with many Scala test frameworks.

o Continuous build, test and deployment.

Why sbt?

- *For Scala and Java:*

↬ sbt is designed for Scala and Java projects.

↬ It's the build tool of choice for 93.6% of developers Scala (2019).

↬ One of the examples of Scala-specific functionality is the possibility to create your project with several versions of Scala.

- *Typesafe and parallel:*

↬ build.sbt is a Scala-based DSL for expressing a graph of parallel processing tasks. typos in

↬ build.sbt will be detected as a compilation error.

- *Fast iteration:*

↬ With Zinc incremental compiler and file monitoring (~), the edit-compile-test loop is fast and incremental.

- *Expandable:*

↬ Added support for new tasks and platforms (like Scala.js) is as easy as writing build.sbt.

2. Preparing environment:

First Step: Preparing data source:

First thing to do is generate our data using a python script. The output is a log file located in /tmp/log-generator.log

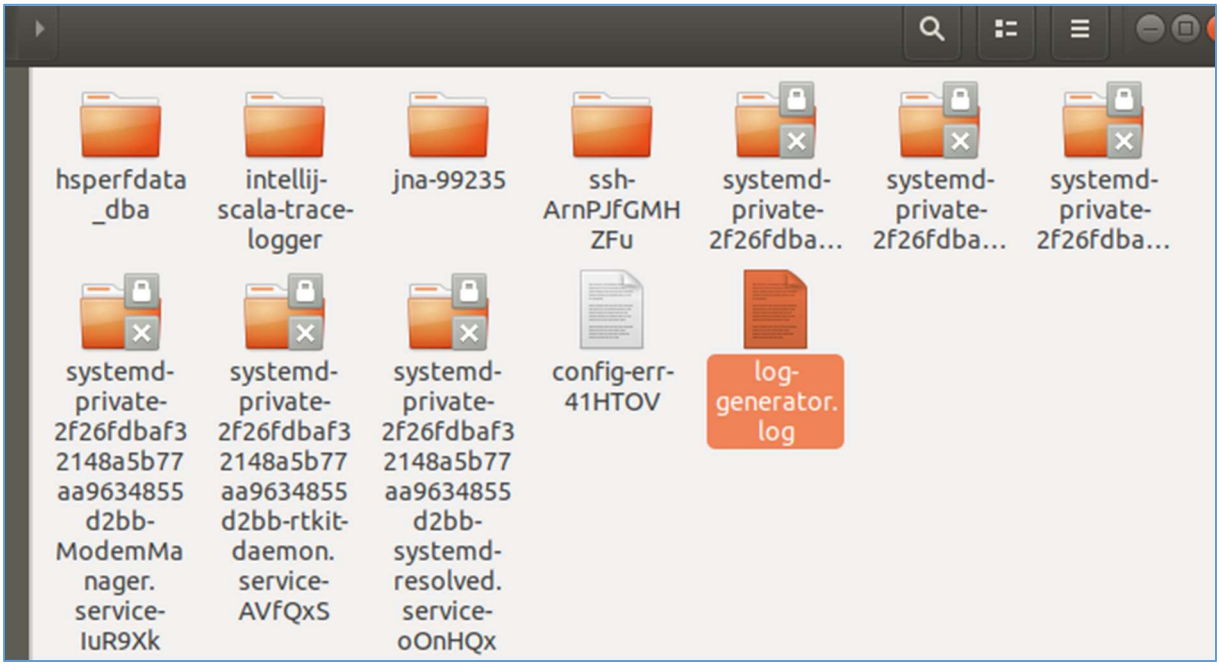
- **Installing Python:**

```
dba@vm:~/Desktop/SparkStreamngProject$ sudo apt install python
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  bridge-utils ubuntu-fan
Use 'sudo apt autoremove' to remove them.
```

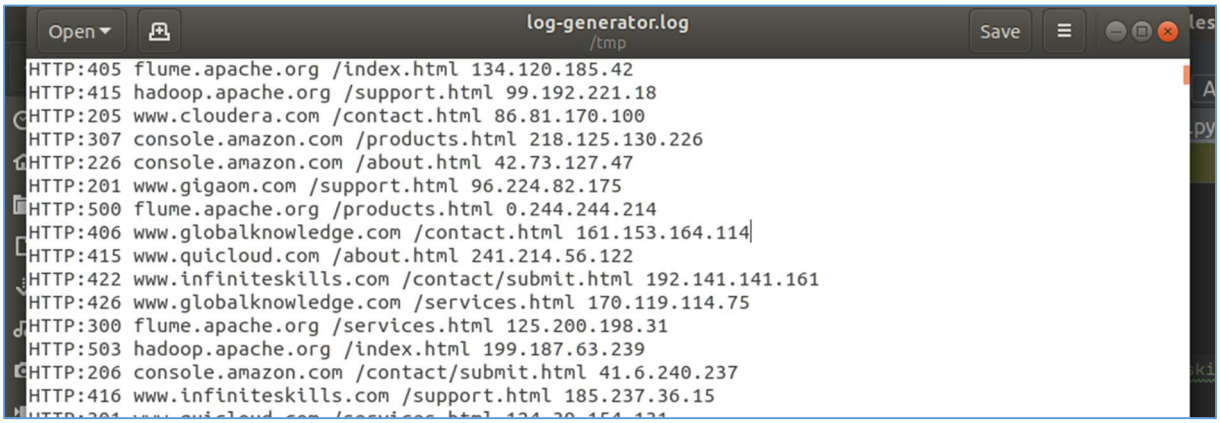
- **Running the script:** we use python3 command to run our script

```
dba@vm:~/Desktop/SparkStreamngProject$ python3 log-generator-files.py
```

- **Checking the output:** we go to /tmp directory to check if the output file is created.



The output file contains many rows, each row has 4 fields: An http code, a website link, an URI and an IP address.



For now, everything is fine!

- **Splitting the data:** Now let's split the input data.

Data generated by the python script is unlimited a continuous. That's why we need to split the log file into multiple smaller log files with timestamps. In order to permit spark streaming to treat each log file as a batch.
We can split data into batches using a shell script:

1. First, we declare two variables: **LinesCounter** & **TempBuffer**.
   LinesCounter is created to count 1000 Line of data, so we can generate a log file after each 1000 line.
   And **TempBuffer** is a string variable which used to store lines that are going to be written into a log file. After each 1000 line it is initialized again with empty string.

```
1  LinesCounter=0
2  TempBuffer=""
```

2. Tail command:

   The tail command is used to print the last 10 lines of a file. In our case, it is used to parse the generated log file, then we are going to redirect the default output (terminal – console) into a while loop which will read each line. and check if the LinesCounter is equal to 0. If it is, we initialize the Splitter again, if LinesCounter is not equal to 0 then we add the line to the TempBuffer. When the LinesCounter is equal to 1000 then we

create a file in the specified directory with the current date time as name to distinguish between files. Then we put the TempBuffer content into the created file using echo command. And finally reset the LinesCounter to 0 to permit the creation of the next log file.

Here is the whole script:

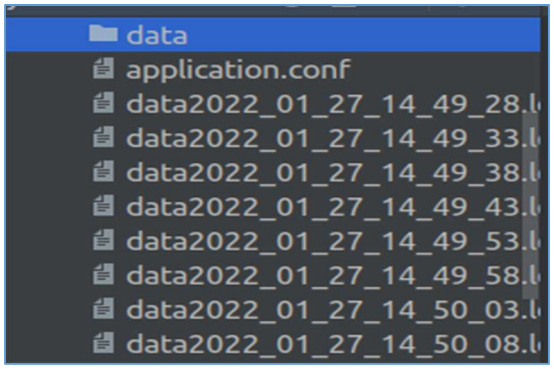
```
LinesCounter=0
TempBuffer=""
tail -f "/tmp/log-generator.log" | while read line; do
    if [ $LinesCounter -eq 0 ]
    then
        TempBuffer=""
    fi
    TempBuffer="$TempBuffer\n$line"
    LinesCounter=$((LinesCounter+1))
    if [ $LinesCounter -eq 1000 ]
    then
        today=`date +%Y_%m_%d_%H_%M_%S`
        filename="/home/Isbaine_Jawhar_Snoussi/Desktop/SparkStreamingProject/src/main/resources/data$today.l
        echo -e "$TempBuffer" > $filename
        LinesCounter=0
    fi
done
```

This script will generate files into data folder. Each file will be treated by spark steaming

To run this script. We open another terminal, then we run the ./Data-splitter command as follows:

```
dba@vm:~/Desktop/SparkStreamngProject$ ./Data-Splitter.sh
```

We can see our files in data folder:



Second Step: Prepare Kibana and Elastic search

In order to install and use Elasticsearch and Kibana, we choose to use **dockerized Elasticsearch and Kibana** for the main reason, which is avoiding complex configurations of these two applications. With Docker containers, everything is pre-configured. We need just to download the container and run it on Docker engine. Note that with Docker we don't have to care about our machine configuration or OS. Docker run containers everywhere in an isolated way. Each of ES and Kibana runs on his own container. Which means we got 2 containers to run. We can do it manually by running each one separately then connect them using a private Docker network. Or we can use Docker-compose which allows us to run multiple containers and connect them using a single command.

We are going to use the second choice.

1. First step we need to install Docker.
2. Then install Docker-compose
3. After that, we have to write a yaml file which will contain

everything about the two containers.
In our case, we named this file ***elasticsearch-kibana-compose.yaml***

```yaml
version: '3'
services:

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:$ELASTIC_VERSION
    environment:
      - bootstrap.memory_lock=true
      - discovery.type=single-node
      - "ES_JAVA_OPTS=-Xms2g -Xmx2g"
      - ELASTIC_PASSWORD=$ELASTIC_PASSWORD
      - xpack.security.enabled=$ELASTIC_SECURITY
    ulimits:
      memlock:
        soft: -1
        hard: -1
    ports:
      - 9200:9200
    networks: ['stack']
```

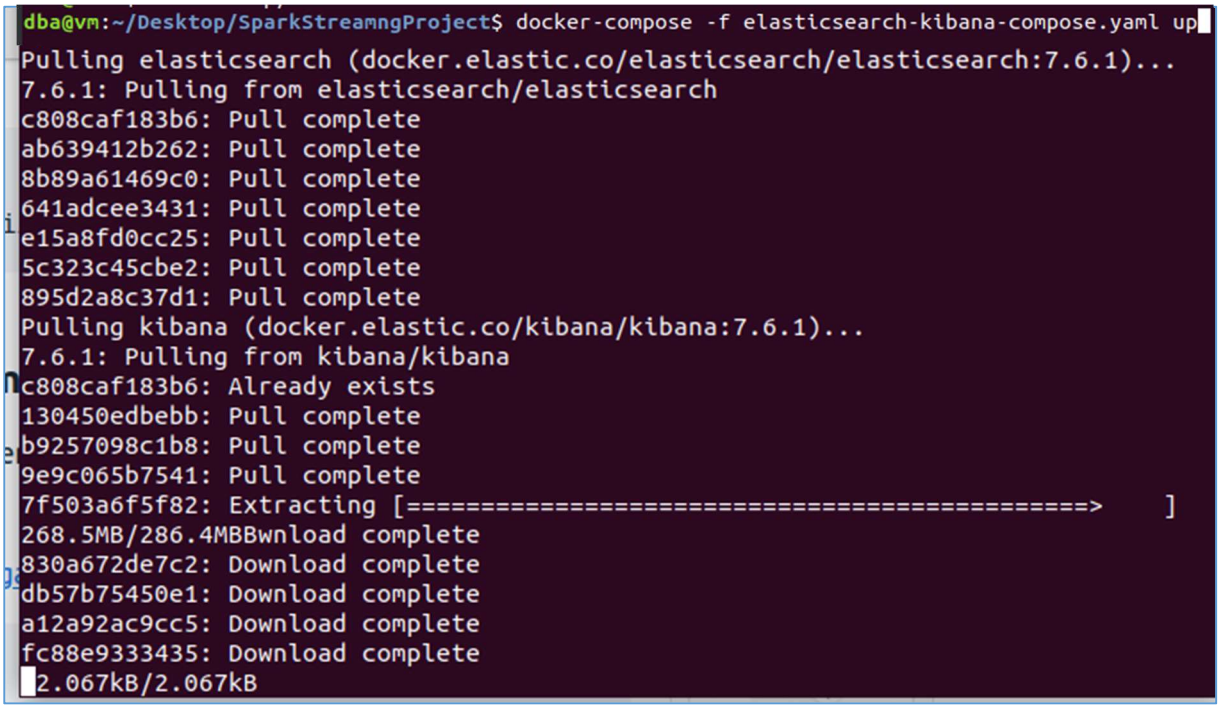
```
  kibana:
    image: docker.elastic.co/kibana/kibana:$ELASTIC_VERSION
    environment:
      - ELASTICSEARCH_USERNAME=elastic
      - ELASTICSEARCH_PASSWORD=$ELASTIC_PASSWORD
    ports: ['5601:5601']
    networks: ['stack']
    links: ['elasticsearch']
    depends_on: ['elasticsearch']

networks:
  stack: {}
```

First, we declare the version of the compose file (in our case it version 3). Then, we move to services (containers), in our case there are two services Elasticsearch and Kibana:

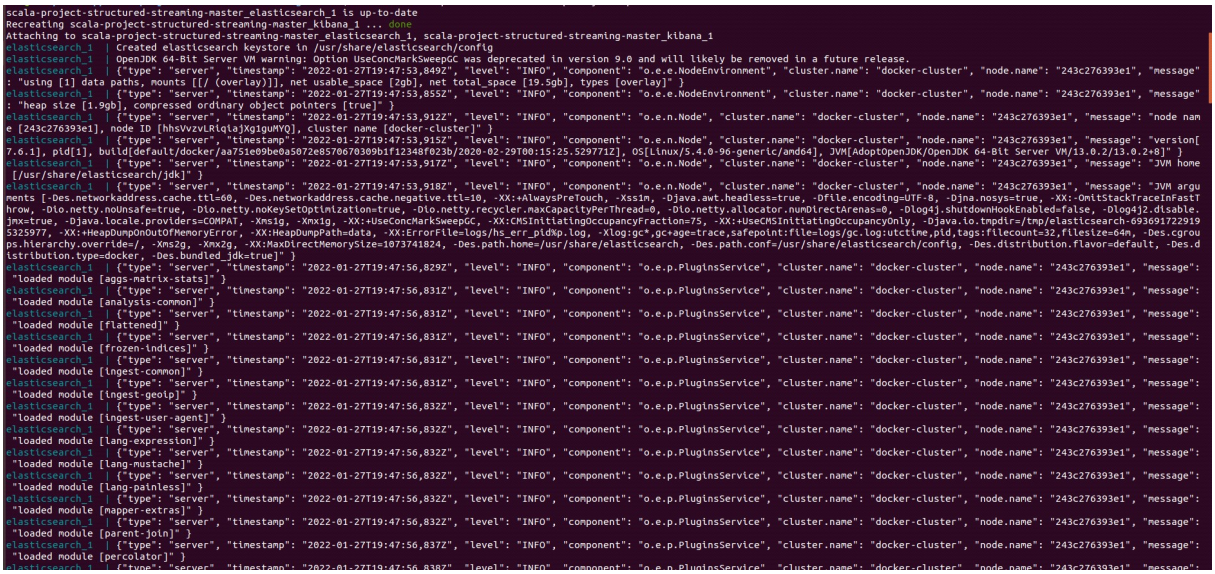
1. ES service: we declare the source of the container image (in our case we download the image from ES official website). Then, we set the environment:
   - Disabling the memory swapping which bad for container performances. This is done by locking the process address into RAM
   - Declaring that we are working on a single node by setting discovery.type to single node
   - Setting JVM options.
   - Declaring ES credentials (defined at the .env file see below)
   - Port forwarding: ES runs on port 9200
   - Networking: we will use a private network named stack
2. Kibana service:
   - Declare the image source
   - Afford ES credentials to enable connection between ES and Kibana
   - Declare the port: 5601
   - Networking: use the stack network

To run the containers. We are going to run the following command:

docker-compose –f elasticsearch-kibana-compose.yaml up

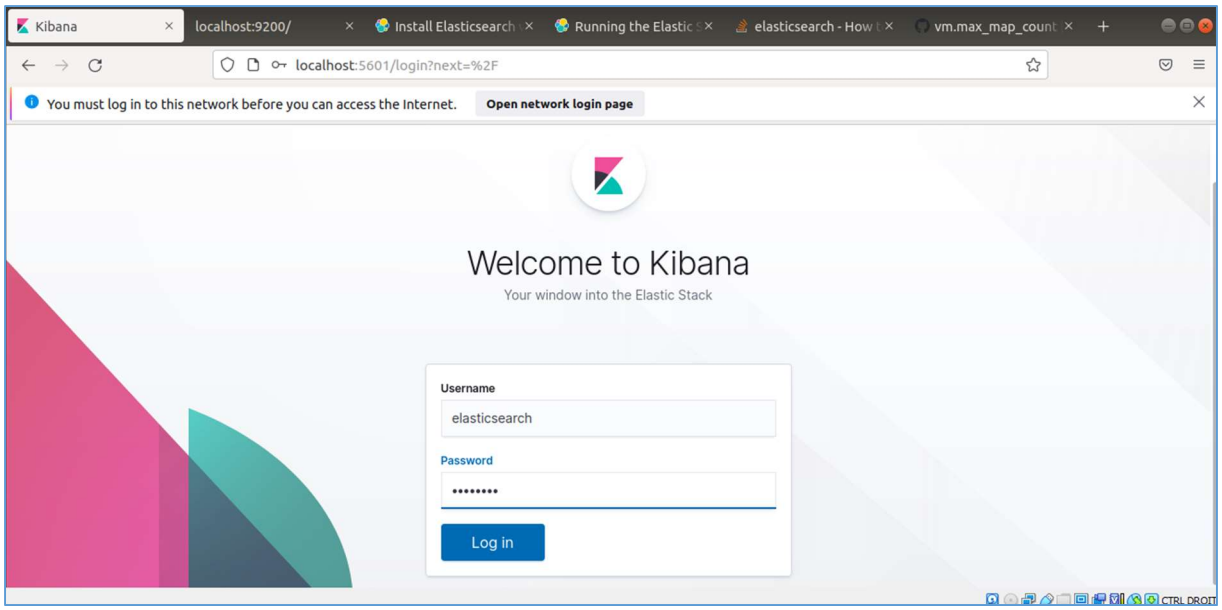


After downloading images, containers are built.

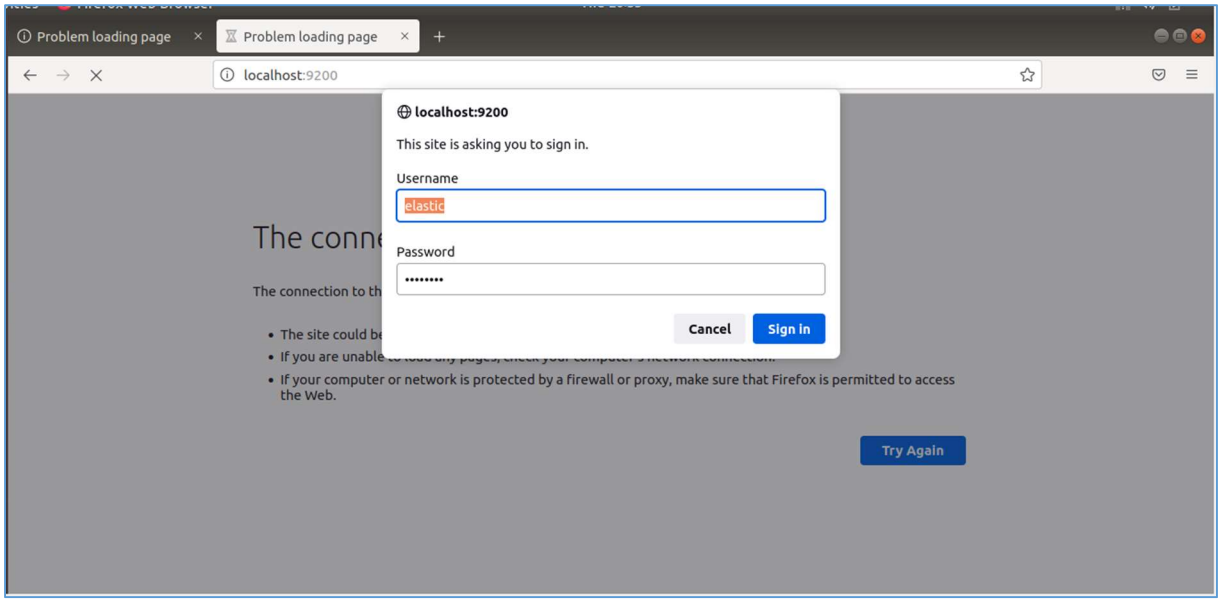


Kibana and ES are lunched now

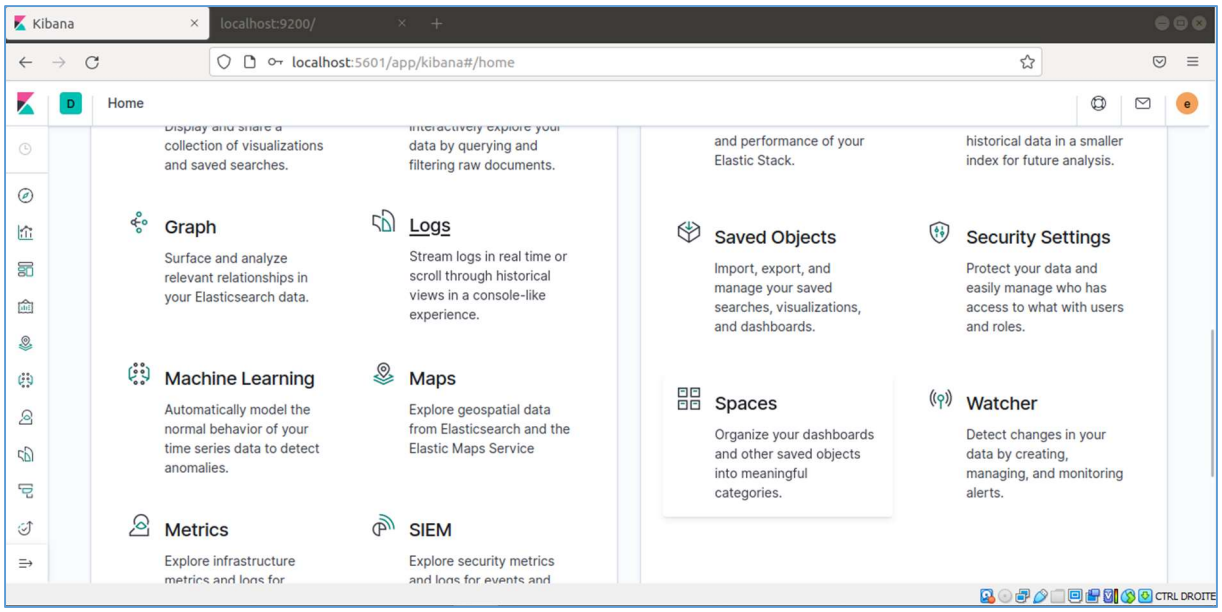
We can access Kibana dashboard by typing ***localhost :5601*** in the browser.



The login page of Elasticsearch is shown below. We have to enter credentials **(username: elastic, password: changeme)**



After logging in into ES, we need to log in into Kibana with the same credentials. A welcome page is shown then:

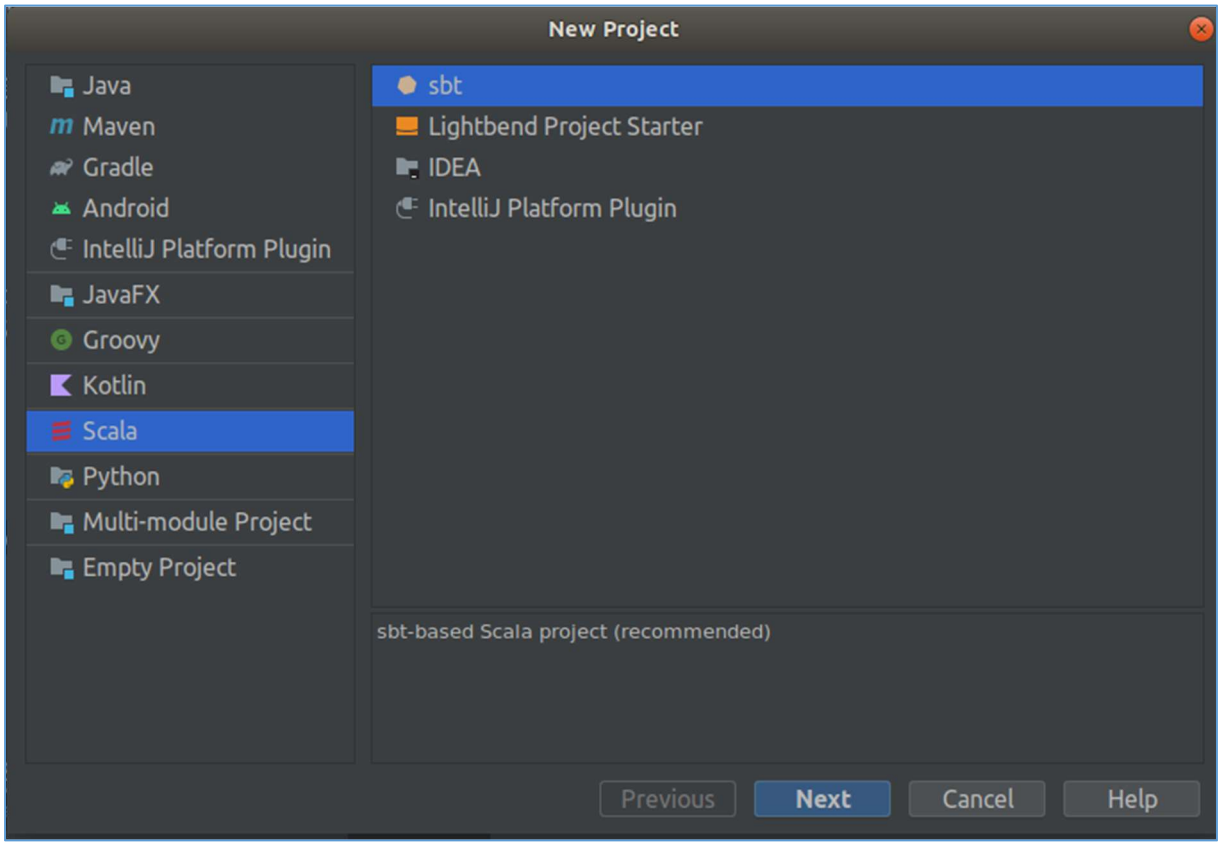


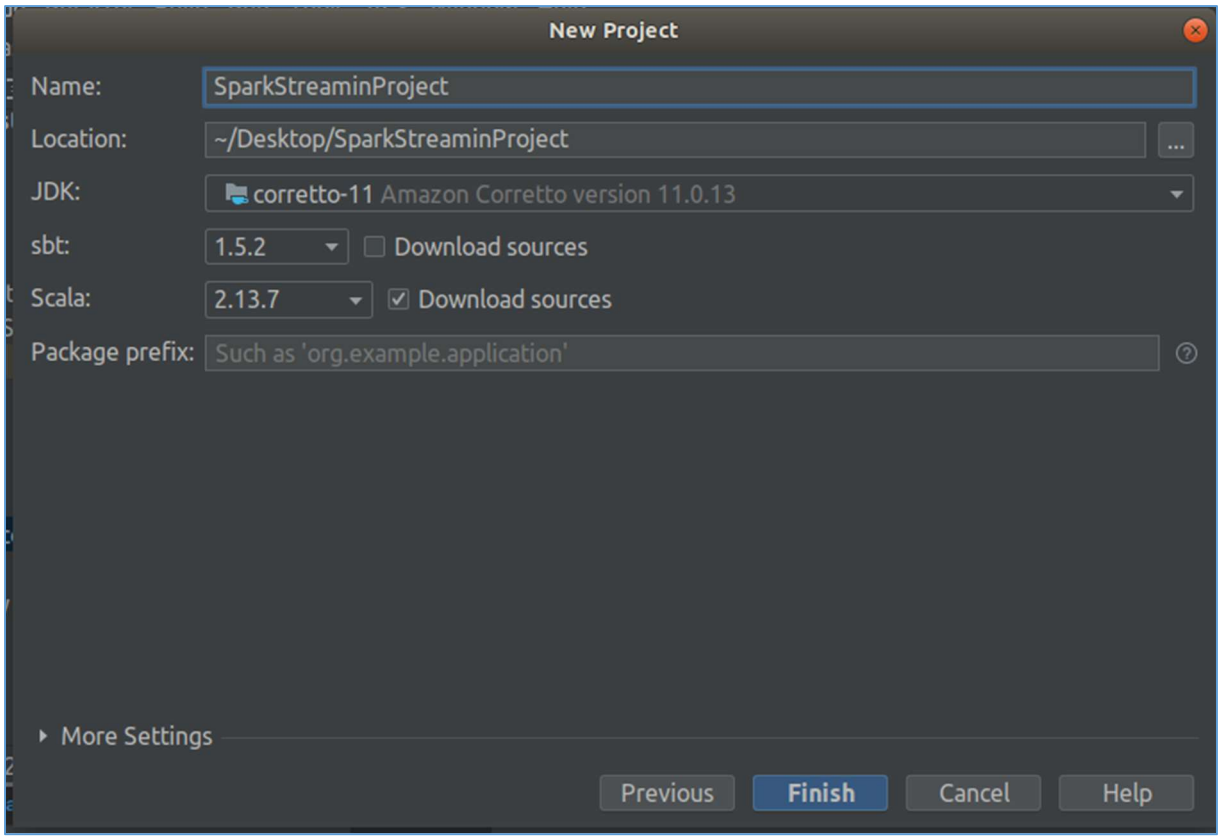
Third Step: Running Spark Streaming

Now after setting up our environment, it's time to run spark streaming.

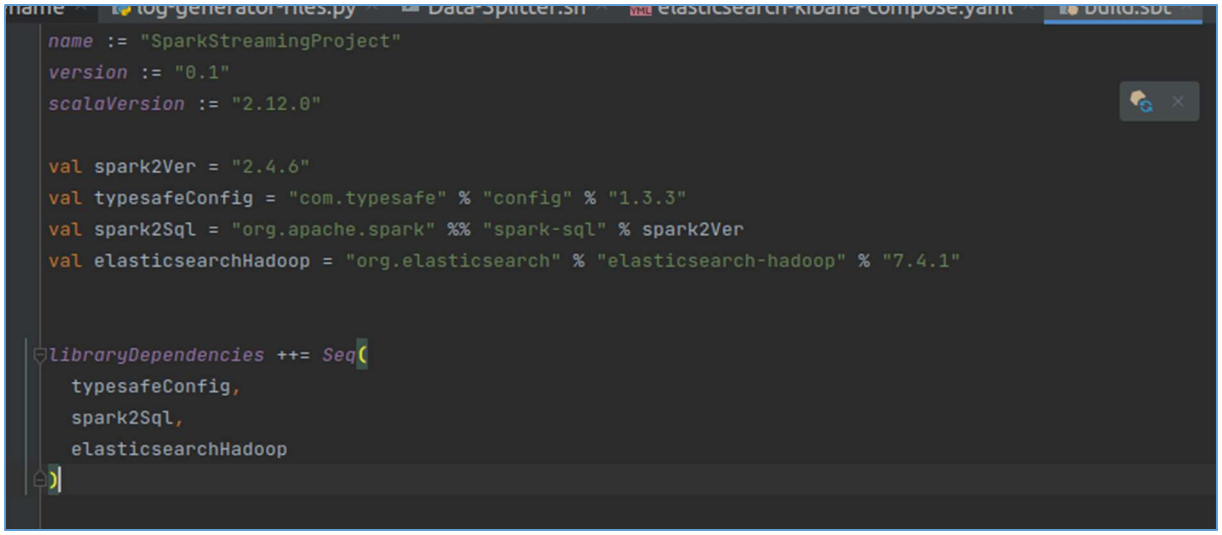
- **Creating a SBT project:**

We are going to create a sbt project with IntelliJ





We add necessary dependencies into build.sbt file



```
name := "SparkStreamingProject"
version := "0.1"
scalaVersion := "2.12.0"

val spark2Ver = "2.4.6"
val typesafeConfig = "com.typesafe" % "config" % "1.3.3"
val spark2Sql = "org.apache.spark" %% "spark-sql" % spark2Ver
val elasticsearchHadoop = "org.elasticsearch" % "elasticsearch-hadoop" % "7.4.1"


libraryDependencies ++= Seq(
  typesafeConfig,
  spark2Sql,
  elasticsearchHadoop
)
```

Finally, we Create main Scala Object:

- First thing, we import necessary classes…
- Second, we load environment configuration:

  In Scala, ***typesafe.config.ConfigFactory*** provides us a way to store environment configuration when we have multiple application. So in our example, we need to store configuration

related to spark, **Elasticsearch** and **Kibana**. These data are stored in ***/data/application.conf*** file

```
spark{

  master = "local[*]"
  master = ${?SPARK_MASTER}
  app.name = "sample-structured-streaming"
  app.name = ${?SPARK_APP_NAME}

  json.resource.path = "src/main/resources/json-resources"
  json.resource.path = ${?JSON_PATH}

  elasticsearch{
    username = "elastic"
    username = ${?ELASTICSEARCH_USER}
    password = "changeme"
    password = ${?ELASTICSEARCH_PASSWORD}
    host = "127.0.0.1"
    host = ${?ELASTICSEARCH_HOST}
    port = "9200"
    port = ${?ELASTICSEARCH_PORT}
    data.source = "org.elasticsearch.spark.sql"
    data.source = ${?ELASTICSEARCH_SPARK_DATASOURCE}
    output.mode = "append"
    output.mode = ${?ELASTICSEARCH_SPARK_OUTPUT_MODE}
    checkpoint.location = "src/main/resources/checkpoint-location-elasticsearch"
    checkpoint.location = ${?ELASTICSEARCH_CHECKPOINT}
    index = "employee"
    index = ${?ELASTICSEARCH_INDEX}
    doc.type = "personal"
    doc.type = ${?ELASTICSEARCH_DOCTYPE}
```

These configuration data can be loaded using CongifFactory.load() method and stored at config val

- Getting configuration data:

  After loading configuration data from ***application.conf*** file we need to store each value in a single val as follows :

```
import com.typesafe.config.ConfigFactory
import org.apache.spark.sql.types._
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.elasticsearch.hadoop.cfg.ConfigurationOptions

object Main extends App{

  private val config = ConfigFactory.load()

  private val MasterNode = config.getString( path = "spark.master")
  private val ApplicationName  = config.getString( path = "spark.app.name")
  private val ESUser = config.getString( path = "spark.elasticsearch.username")
  private val ESPassword = config.getString( path = "spark.elasticsearch.password")
  private val ESHost = config.getString( path = "spark.elasticsearch.host")
  private val ESPort = config.getString( path = "spark.elasticsearch.port")
  private val outputMode = config.getString( path = "spark.elasticsearch.output.mode")
  private val destination = config.getString( path = "spark.elasticsearch.data.source")
```

```
private val destination = config.getString( path = "spark.elasticsearch.data.source")          ⚠ 9  ✓ 1
private val checkpointLocation = config.getString( path = "spark.elasticsearch.checkpoint.location")
private val index = config.getString( path = "spark.elasticsearch.index")
private val docType = config.getString( path = "spark.elasticsearch.doc.type")



private val indexAndDocType = s"log/$docType"

private val pathToJSONResource = config.getString( path = "spark.json.resource.path")
```

- Creating a SparkSession Object using Elasticsearch configuration:

SparkSession is the entry point to Spark SQL, it is one of the very first objects we create while developing a Spark SQL application. We can create a SparkSession using the SparkSession.builder() method which gives us the access to Builder API the we use to configure the session.

- Then, we create a schema for our data, which describes the format of manipulated data:

In our case, each line has 4 columns (http code, Site, URI, IP address)

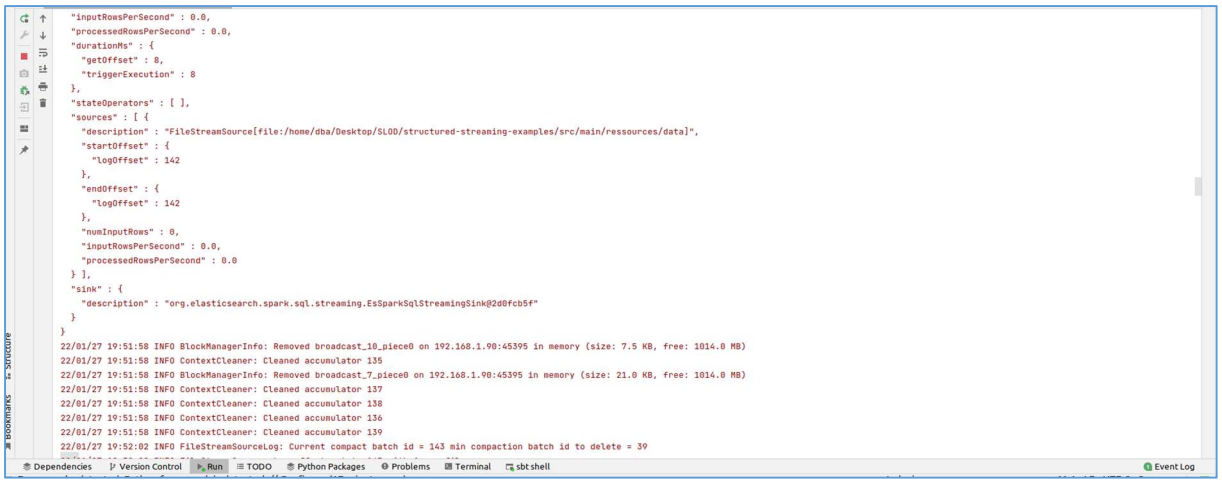
```
//creating SparkSession object with Elasticsearch configuration
val sparkSession = SparkSession.builder()
  .config(ConfigurationOptions.ES_NET_HTTP_AUTH_USER, ESUser)
  .config(ConfigurationOptions.ES_NET_HTTP_AUTH_PASS, ESPassword)
  .config(ConfigurationOptions.ES_NODES, ESHost)
  .config(ConfigurationOptions.ES_PORT, ESHost)
  .master(MasterNode)
  .appName(ApplicationName)
  .getOrCreate()



val schema = StructType(List(
  StructField("HTTPCODE", StringType, true),
  StructField("SITE", StringType, true),
  StructField("URI", StringType, true),
  StructField("IP", StringType, true)
))
```

- Create a **SparkDataframe** with SparkSession.readStream method, which returns a DataStreamreader that can be used streaming data in as a **dataframe**.
- Write the program output into **Elasticsearch** data source org.elasticsearch.spark.sql. which will allow us to store results

```
val StreamDF = sparkSession.readStream.option("delimiter", " ").schema(schema)
  .csv( path = "/home/dba/Desktop/SparkStreamingProject/src/main/resources/data")


StreamDF.writeStream
  .outputMode(outputMode)
  .format(destination)
  .option("checkpointLocation", checkpointLocation)
  .start(indexAndDocType)
  .awaitTermination()



}
```

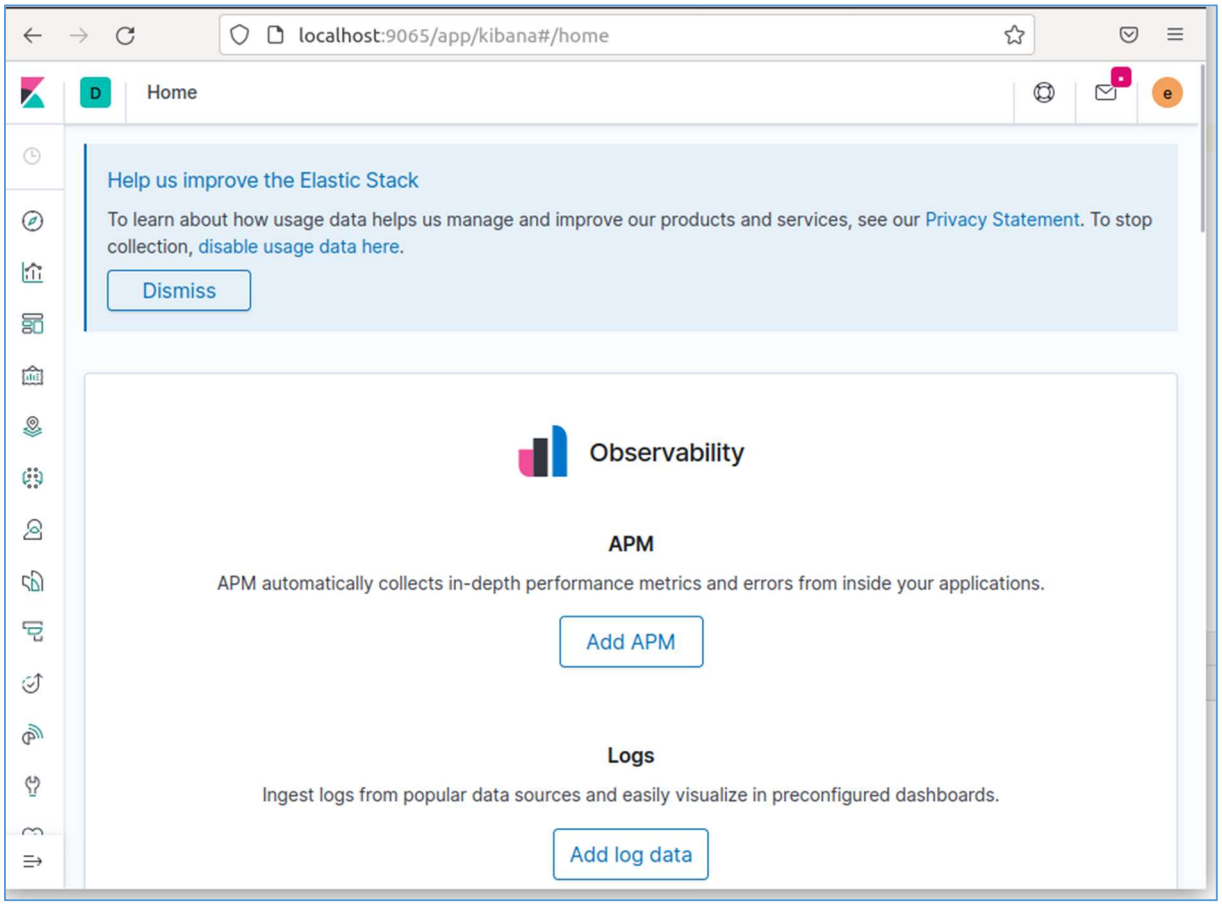
- **Running Spark Streaming.**

We run the spark application from IntelliJ:



3. Data Visualization:

After running spark, we want to show spark treatment results on a Kibana dashboard:



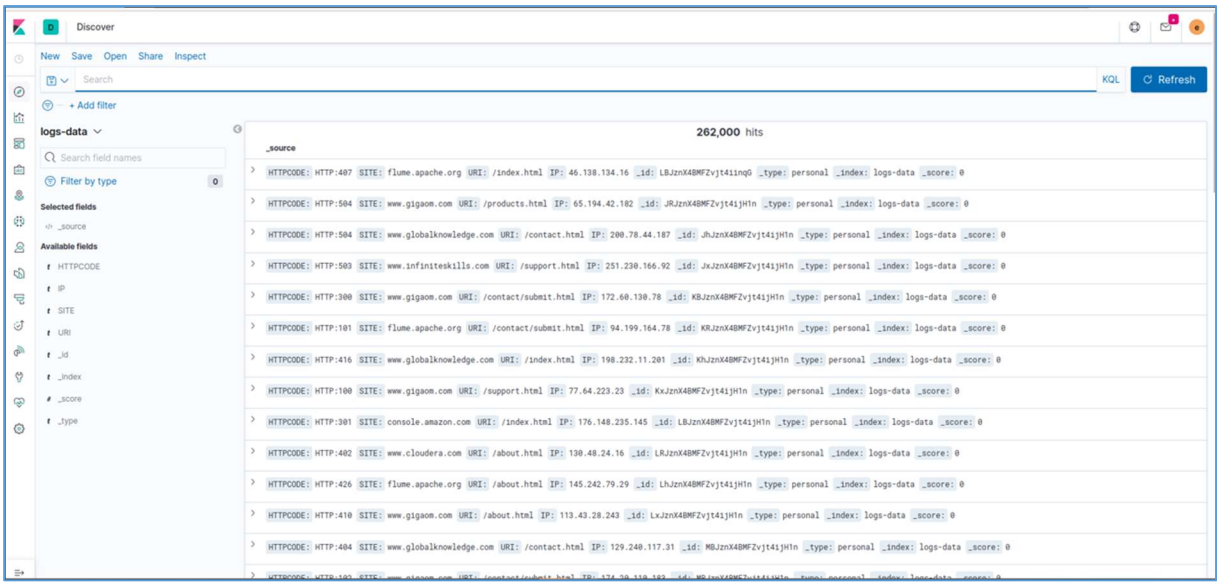
- **Creating new Dashboard on Kibana**

We select ***new Dashboard***.

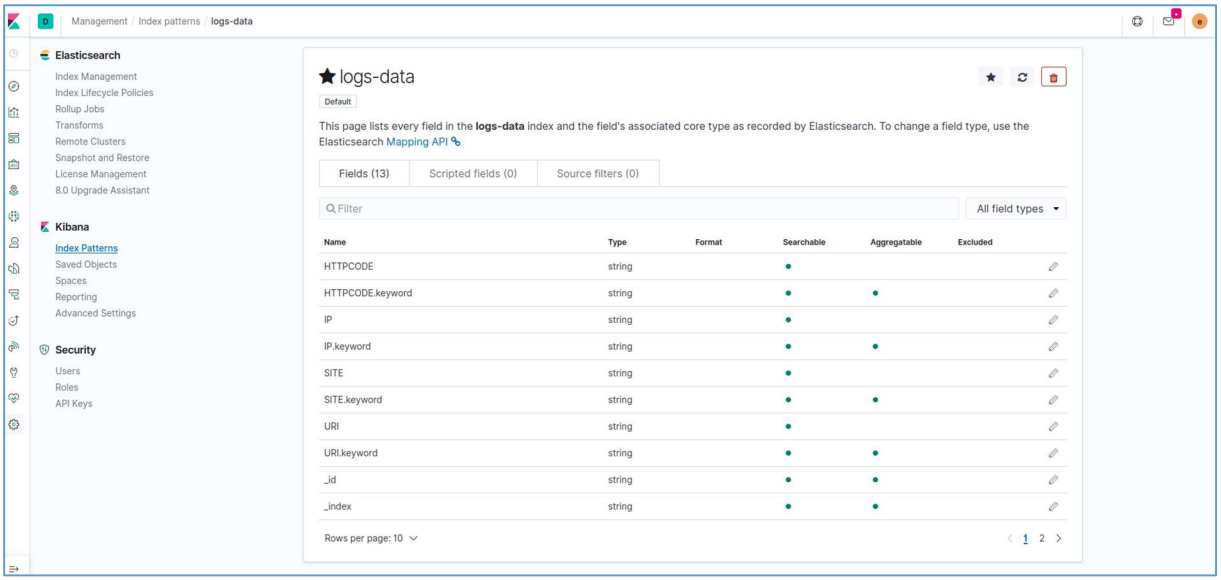
Then we click on ***Add Panels*** and we choose spark treatment output and click on ***save.***
And finally we give our dashboard a title (logs-data)

Here are the results:

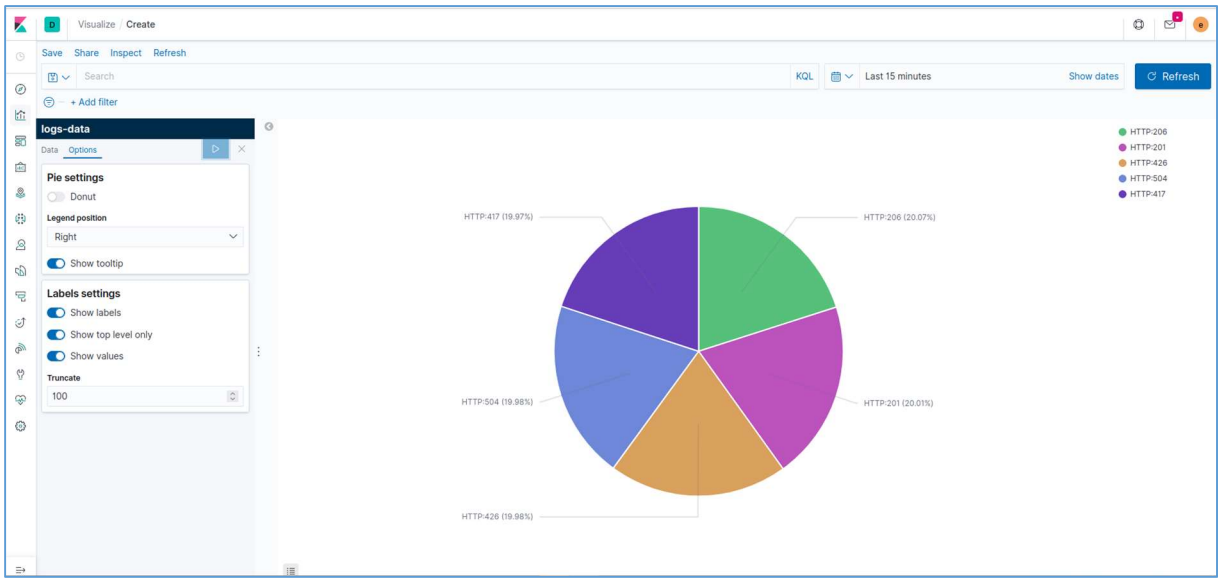
- **Visualizing All the data:**



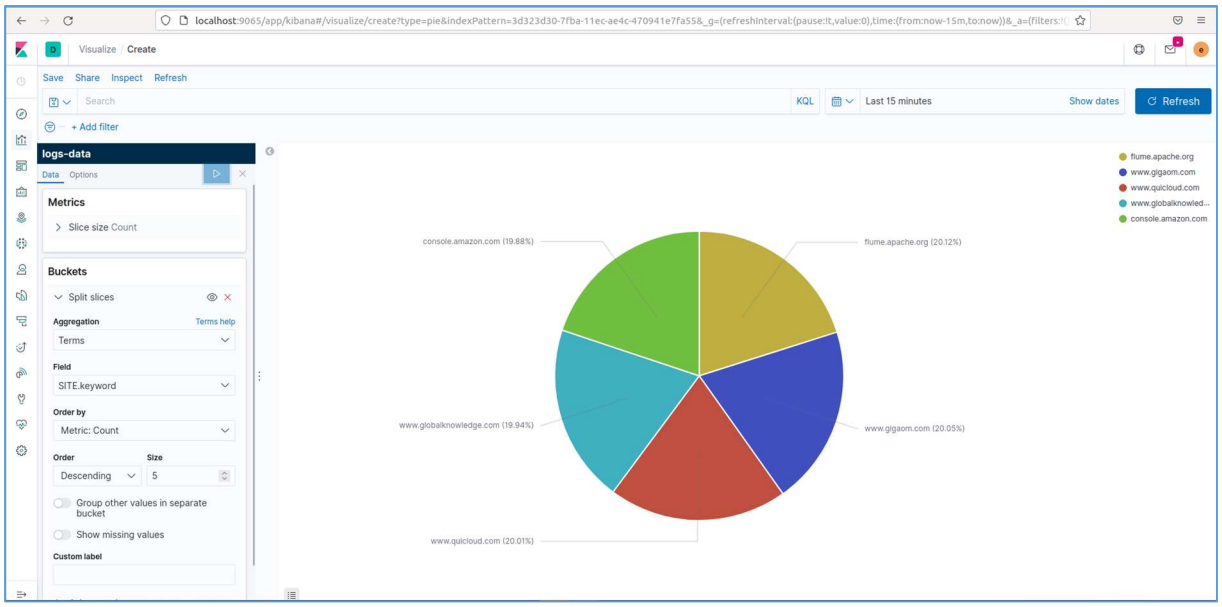
- **Visualizing the data structure:**



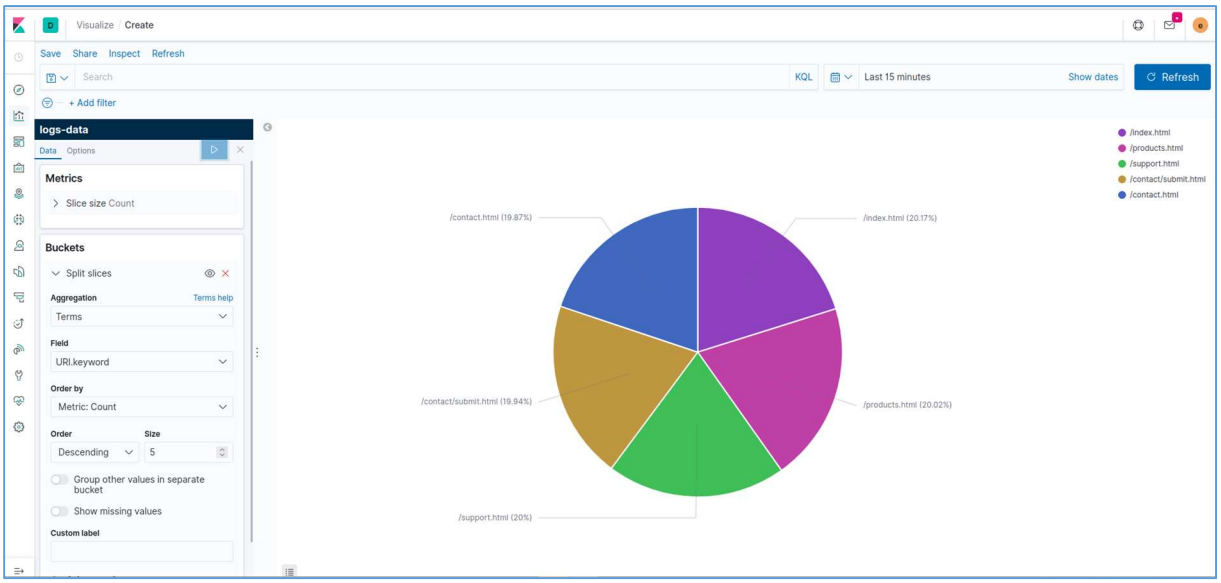
- **Visualizing data summary using http code as a keyword:**



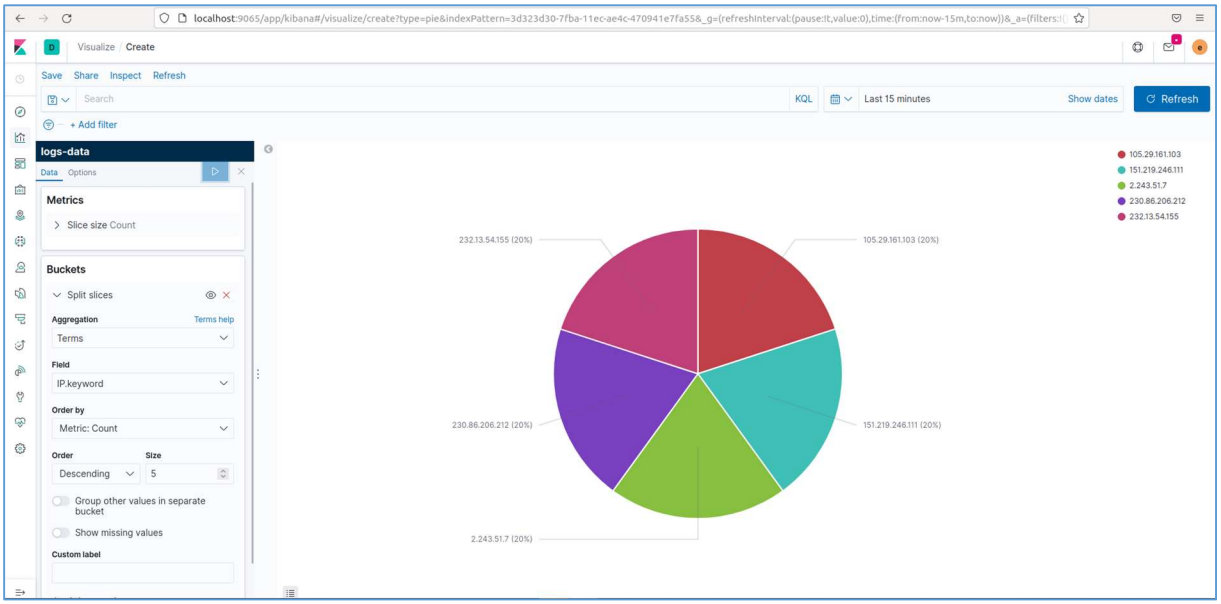
- **Visualizing the data summary using the website as the keyword:**



- **Visualizing the data summary using URI as the keyword:**



- **Visualizing the data summary using IP address as keyword:**



As we can see, there is a refresh button which allows us to refresh data summary and have a real time visualization.

Conclusion:

This project was very beneficial for us as a team. It allowed to us to discover new top market technologies such as Kibana, **Elasticsearch** which are highly demanded in job market. We thank Mr. Ziyati Houssine for his efforts and attend.