EJERCICIOS ALMACENAMIENTO Y PROCESAMIENTO DE DATOS

1. **Objective**: Exercise you will see how quickly and easily the Elastic Stack can be used to search a dataset. You will startup Elasticsearch and Kibana, then run queries from Kibana to search an indexed dataset.

**Note**: The datasets we are going to use are on git repo on following folder: https://github.com/rlopezherrero/GFT-EDEM-MasterData/tree/master/AlmacenamientoProcesamiento/docker/data

* 1. Launch docker compose with full ELK stack

$ GFT-EDEM-MasterData/AlmacenamientoProcesamiento/docker/

$ docker-compose up -d

* 1. Wait a couple of minutes to ensure that the stack is running.
  2. Open your Web browser and go to [http://localhost:9200](http://localhost:9200/). You should see the start page that verifies Elasticsearch is running on your localhost - a simple JSON object that will look similar to:

{"name" : "elasticsearch",

"cluster\_name" : "docker-cluster",

"cluster\_uuid" : "UJuJfAFlR-m0kyuyLd0mew",

"version" : {

"number" : "7.3.0",

"build\_flavor" : "default",

"build\_type" : "docker",

"build\_hash" : "de777fa",

"build\_date" : "2019-07-24T18:30:11.767338Z",

"build\_snapshot" : false,

"lucene\_version" : "8.1.0",

"minimum\_wire\_compatibility\_version" : "6.8.0",

"minimum\_index\_compatibility\_version" : "6.0.0-beta1"

},

"tagline" : "You Know, for Search"

* 1. Based on previous output:

o What is the version of Elasticsearch instance? 7.3.0

o What is the name of your node? “elasticsearch”

o What is the name of your cluster? “docker-cluster”

* 1. Analyse the data set we are going to use (data/products). The dataset you are going to be using is a collection of products sold in stores. The fields in the dataset consist of:

o grp\_id: a unique ID for each row

o upc12: a 12-digit string containing the Universal Product Code

o brandName: the name of the company that makes the product

o productName: the name of the product

o customerRating: an integer value between 1 and 5, where 5 is the highest rating

o price: a float value representing the price of the product

o quantitySold: an integer value representing the number of units sold

Review this small sample of what the data looks:

20,204040000000,Usda Produce,Plums Black,3,1.39,68603

21,753950001954,Doctor's Best,Doctor's Best Best Curcumin C3 Complex 1000mg Tablets - 120 Ct,2,1.75,651857

22,016000288829,Betty Crocker,Betty Crocker Twin Pack Real Potatoes Scalloped 2 Pouches For 2 Meals - 2 Pk,2,4.22,527899

23,070670009658,Reese,Reese Mandarin Oranges Segments In Light Syrup,2,3.73,104348

24,688267084225,Smart Living,Smart Living Charcoal Lighter Fluid,5,3.20,637769

25,044100117428,Hood,Hood Latte Iced Coffee Drink Vanilla Latte,5,5.99,166777

* 1. The products dataset was indexed already into your Elasticsearch instance. Go to<http://localhost:9200/_cat/indices?v>. Your **products** index should appear in the list of indices. (Notice the size may vary, but it should contain 110,435 documents):

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

**green open products 7J9gdrEPS3GJqzjDBs7SIA 2 0 110435 0 26.6mb 26.6mb**

green open .kibana 2cg2sj3VTNeNfHuESfFkzA 1 0 1 0 3.7kb 3.7kb

* 1. To view the products data, use the Search API with the following URL:<http://localhost:9200/products/_search>. You should see 10 products (10 is the default search size). The products displayed will likely vary from the output below, but notice the output is not very pretty:

{"took":57,"timed\_out":false,"\_shards":{"total":2,"successful":2,"skipped":0,"failed":0},"hits":{"total":110435,"max\_score":1.0,"hits":[{"\_index":"products","\_type":"product","\_id":"50550","\_score":1.0,"\_source":{"brandName":"Riceland","productName":"Riceland Extra Long Grain Rice","quantitySold":733733,"price":11.87,"grp\_id":"50550","type":"csv","upc12":"035200055154","customerRating":2}},{"\_index":"products","\_type":"product","\_id":"50551","\_score":1.0,"\_source":{"brandName":"Silverpoint","productName":"Silverpoint Quad Planning Pad","quantitySold":450550,"price":11.8,"grp\_id":"50551","type":"csv","upc12":"075755510708","customerRating":1}},

……

* 1. Change the URL to include the pretty argument: <http://localhost:9200/products/_search?pretty>. The output should now look much nicer:

{

{

"took" : 5,

"timed\_out" : false,

"\_shards" : {

"total" : 2,

"successful" : 2,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : 110435,

"max\_score" : 1.0,

"hits" : [

{

"\_index" : "products",

"\_type" : "product",

"\_id" : "50550",

"\_score" : 1.0,

"\_source" : {

"brandName" : "Riceland",

"productName" : "Riceland Extra Long Grain Rice",

"quantitySold" : 733733,

"price" : 11.87,

"grp\_id" : "50550",

"type" : "csv",

"upc12" : "035200055154",

"customerRating" : 2

}

},

{

"\_index" : "products",

"\_type" : "product",

"\_id" : "50551",

"\_score" : 1.0,

"\_source" : {

"brandName" : "Silverpoint",

"productName" : "Silverpoint Quad Planning Pad",

"quantitySold" : 450550,

"price" : 11.8,

"grp\_id" : "50551",

"type" : "csv",

"upc12" : "075755510708",

"customerRating" : 1

}

},

….

* 1. Based on the output of previous search:

o How many documents were indexed? 10 (default) of all the 110435.

o What is the \_type of each document? CSV.

* 1. Next, you are going to use Kibana.
  2. To verify Kibana is running, open your Web browser and go to [http://localhost:5601](http://localhost:5601/) . The Kibana application should appear.
  3. Click on the **Dev Tools** button (left side tool icon) in the side navigation pane to open the **Console** application:
  4. Notice there is a **match\_all** query already written in the **Console**. Go ahead and run it by clicking the green "play" button that appears to the right of the command, or using the **Ctrl/Cmd + Enter** keyboard shortcut. This search hits all documents in all indexes of your cluster. Notice the output is similar to the output you saw a few minutes ago, but displayed nicely in the Console screen:
  5. Now let's search for products. Enter the following query into the **Console**, just below your first command. It answers the question "Which products have peanut or butter in the product name?" Feel free to copy-and-paste the command:

GET products/\_search

{

"size": 25,

"query": {

"match": {

"productName": "peanut butter"

}

}

}

Click the green play button next to this new query, or you can use the keyboard shortcut **Command+Enter**. The output will show the top 25 results of products with "peanut" or "butter" in the **productName** field

* 1. Based on the results of your previous search:

o How many products in the dataset match "peanut butter" in the product name? 2011

o What was the max\_score of the results? 11.242394

o Would you say that your top results were relevant to "peanut butter"? Yes

* 1. Change the term **"peanut butter"** to **"Peanut Butter"** and run the query again. What is different in the results, if anything? Nothing, no case sensitive.
  2. Run a search for **"kasmati rice"** in the **"productName"**, then answer the following questions:

o How many hits are there? 1618

o What was the \_score of the top hit? 19.586052

o View the scores of the next few hits. Why do you think the top hit had a much higher score than all the other hits? Both words in the product name.

* 1. Compare the following search to the previous ones you executed. Notice the field being searched on is now customerRating. Run the following query, which answers the query: "Find all products with a customer rating of 4":

GET products/\_search

{

"query": {

"match" : {

"customerRating" : 4

}

}

}

Notice the score for each hit is simply **1**. Why do you think all matching documents have the same score? All same (4 rating).

* 1. Write a query that finds all products whose price field is exactly 10.00. You should get 44 hits.

1. **Objective**: Learn Logstash basics implementing several pipelines and working with several plugins.
   1. First test that your logstash installation is working properly, writing on the standard input:

$ Docker exec -it logstash bash

$ bin\logstash –e “input { stdin { } } output { stdout {} }” –path.data /tmp/

* 1. Create a Logstash pipeline called my-first-pipeline.conf to consume from the filebeat on port 5044 and write to the std output. Add following line to the output section for this: stdout { codec => rubydebug }:

vi my-first-pipeline.conf

input {

beats {

port => "5044"

}

}

# The filter part of this file is commented out to indicate that it is

# optional.

# filter {

#

# }

output {

stdout { codec => rubydebug }

}

* 1. Before launching it, verify that your pipeline configuration is fine.

bin/logstash -f my-first-pipeline.conf --config.test\_and\_exit --path.data /tmp

* 1. If the configuration is fine, launch this pipeline.

bin/logstash -f my-first-pipeline.conf --config.reload.automatic --path.data /tmp

* 1. Configure a filebeat logging, first logon filebeat container:

Docker exec --user root -it filebeat bash

* 1. Create a filebeat config to capture and publish logs.

cp filebeat.yml filebeat-training.yml

* 1. Edit filebeat-training.yml

filebeat.inputs:

- type: log

enabled: true

paths:

- /tmp/datasets/apache\_logs/logstash-tutorial-dataset.log

output.logstash:

# Array of hosts to connect to.

hosts: ["logstash:5044"]

* 1. Launch filebeat process:

filebeat -e -c filebeat-training.yml -d "publish"

* 1. Launch filebeat process:

filebeat -e -c filebeat-training.yml -d "publish"

* 1. Take a look to the logstash, you should see following events on the console:

{

"offset" => 24464,

"message" => "86.1.76.62 - - [04/Jan/2015:05:30:37 +0000] \"GET /style2.css HTTP/1.1\" 200 4877 \"http://www.semicomplete.com/projects/xdotool/

\" \"Mozilla/5.0 (X11; Linux x86\_64; rv:24.0) Gecko/20140205 Firefox/24.0 Iceweasel/24.3.0\"",

"source" => "C:\\Users\\role\\Documents\\Formacion\\Introduction ELK\\course-ELK-introduction\\datasets\\apache\_logs\\logstash-tutorial-datase

t.log",

"@version" => "1",

"prospector" => {

"type" => "log"

},

"beat" => {

"name" => "NBVAL730",

"hostname" => "NBVAL730",

"version" => "6.2.3"

},

"host" => "NBVAL730",

"tags" => [

[0] "beats\_input\_codec\_plain\_applied"

],

"@timestamp" => 2018-04-16T11:36:42.598Z

}

….

* 1. Remove data registry in order to repopulate log messages (/usr/share/filebeat/data/registry)

rm -r /usr/share/filebeat/data/registry

* 1. Now, that we see pipeline is working, let’s tune logstash config. We are going to update logstash pipeline using a specific filter that parses unstructured data and generates a structured json. For doing this we will use grok plugin with the following property : "message" => "%{COMBINEDAPACHELOG}" .

vi my-first-pipeline.conf

input {

beats {

port => "5044"

}

}

filter {

grok {

match => { "message" => "%{COMBINEDAPACHELOG}"}

}

}

output {

stdout { codec => rubydebug }

}

* 1. You should see something like this on the logstash console:

{

"request" => "/presentations/logstash-monitorama-2013/images/kibana-search.png",

"agent" => "\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.77 Safari/537.36\"",

"offset" => 325,

"auth" => "-",

"ident" => "-",

"verb" => "GET",

"prospector" => {

"type" => "log"

},

"source" => "/path/to/file/logstash-tutorial.log",

"message" => "83.149.9.216 - - [04/Jan/2015:05:13:42 +0000] \"GET /presentations/logstash-monitorama-2013/images/kibana-search.png HTTP/1.1\" 200 203023 \"http://semicomplete.com/presentations/logstash-monitorama-2013/\" \"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_9\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.77 Safari/537.36\"",

"tags" => [

[0] "beats\_input\_codec\_plain\_applied"

],

"referrer" => "\"http://semicomplete.com/presentations/logstash-monitorama-2013/\"",

"@timestamp" => 2017-11-09T02:51:12.416Z,

"response" => "200",

"bytes" => "203023",

"clientip" => "83.149.9.216",

"@version" => "1",

"beat" => {

"name" => "My-MacBook-Pro.local",

"hostname" => "My-MacBook-Pro.local",

"version" => "6.0.0"

},

"host" => "My-MacBook-Pro.local",

"httpversion" => "1.1",

"timestamp" => "04/Jan/2015:05:13:42 +0000"

}

* 1. Modify the pipeline to index all logs on local elastic instance. Check on your local kibana instance if an index for logstash data has been created (Management --> create index pattern). You should see 101 log events loaded with proper json format.

input {

beats {

port => "5044"

}

}

# The filter part of this file is commented out to indicate that it is

# optional.

filter {

grok {

match => { "message" => "%{COMBINEDAPACHELOG}"}

}

}

output {

elasticsearch {

hosts => [ "elasticsearch:9200" ]

}

}

* 1. Check the status of the logstash to see your pipeline running. You should get just one pipeline running an output similar to this. Go to this link <http://localhost:9600/_node/pipelines?pretty>

{

"host" : "NBVAL730",

"version" : "6.2.3",

"http\_address" : "127.0.0.1:9600",

"id" : "ac74dbff-9d8f-4b8f-99b5-0f3a71e5365b",

"name" : "NBVAL730",

"pipelines" : {

"main" : {

"workers" : 4,

"batch\_size" : 125,

"batch\_delay" : 50,

"config\_reload\_automatic" : true,

"config\_reload\_interval" : 3000000000,

"dead\_letter\_queue\_enabled" : false

}

}

}

1. In this exercise we will play with Kibana
   1. Create index patterns for 3 datasets. You can use following patterns: bank\*, shakespeare\* and logstash-2015.05\*. Take into consideration that logstash dataset contains a time series.
   2. Now, go to the discover tab and get from bank index, the accounts that have a balance higher than 47500 and the account number is lower than 100. (You should bet 5 results).
   3. Using Shakespeare index, get the entries that are from play “Henry IV” and on the text contains “London”. (You should get 9 hits).
   4. Using account data, create a Pie chart defining following Ranges:

0 999

1000 2999

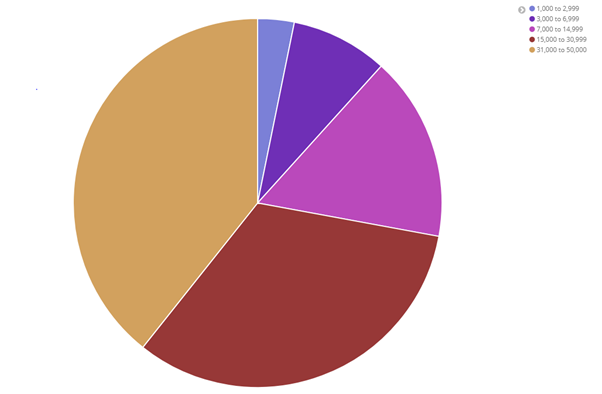
3000 6999

7000 14999

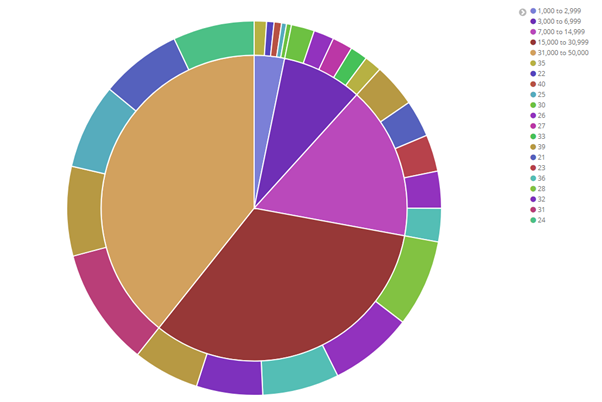
15000 30999

31000 50000

* 1. You should get something similar to this:

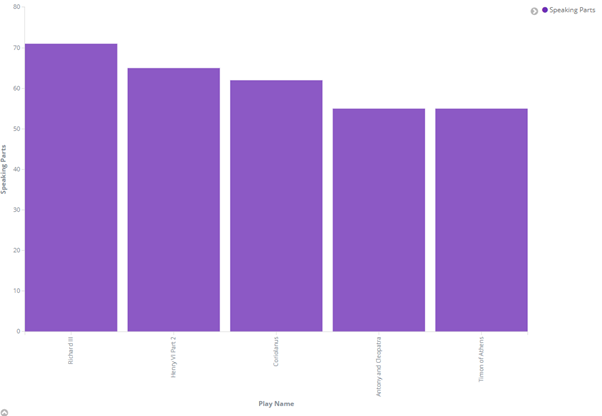


* 1. On same Pie chart add another bucket aggregation to get the age of each account holder inside each balance range. You should get something similar to this:

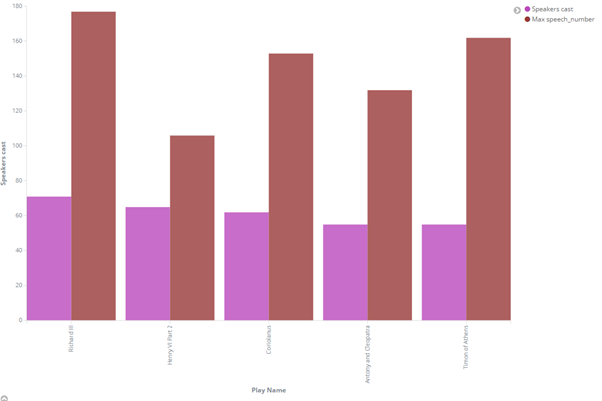


Click on save to use it later, and save it as “Pie Example”.

* 1. Now let’s create a Vertical Bar Chart using Shakespeare dataset. We want so see number of speakers cast per play. You should get something similar to this:



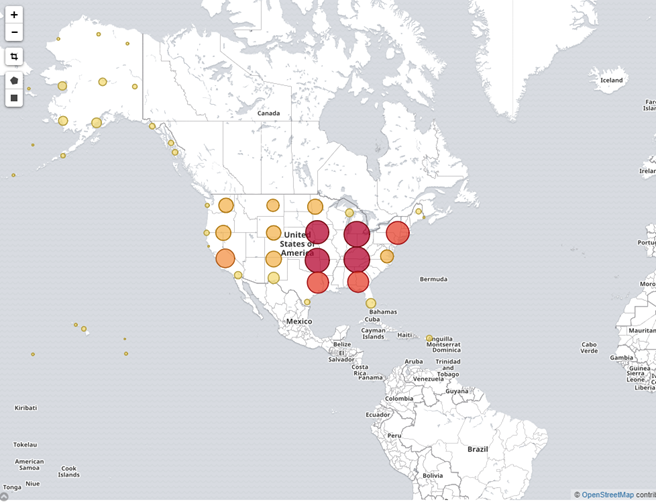
* 1. Using same bar char, we want to get the maximum number of speeches for an individual actor on every play.



Click save, and save it as “Bar chart example”.

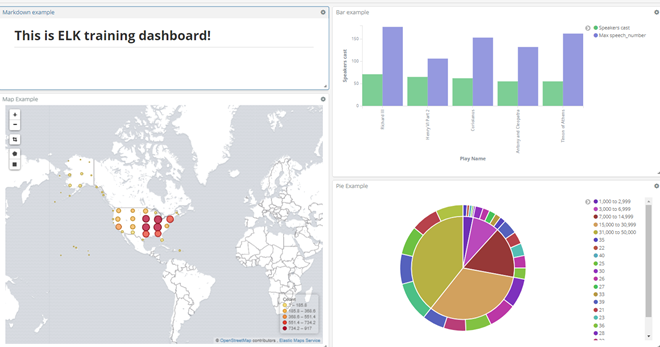
* 1. Now, using Logstash dataset, create a Coordinate Map with following parameters:
     1. Use geo.coordinates as the geo\_point.
     2. Configure the Time window to absolute between May 18 2015 and May 20 2015.

You will get something similar to this:

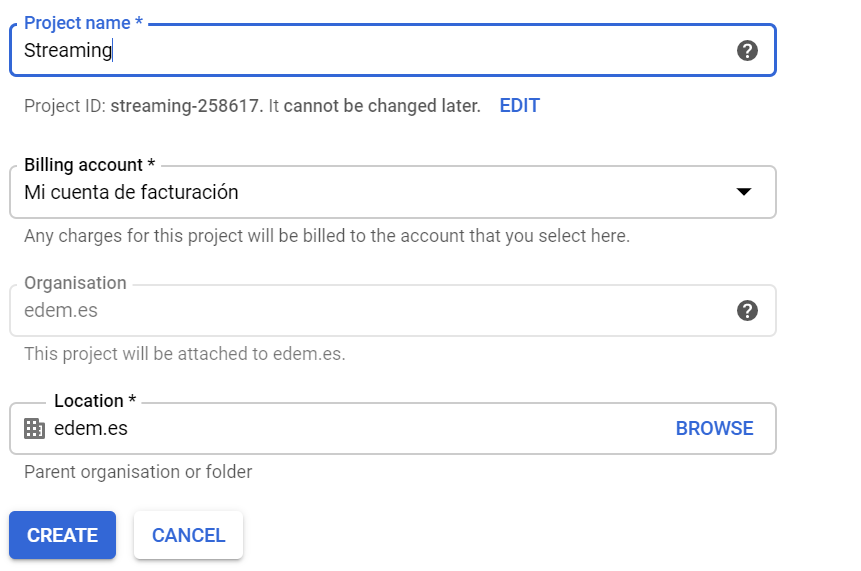


Click save, and save it as “Bar Map example”.

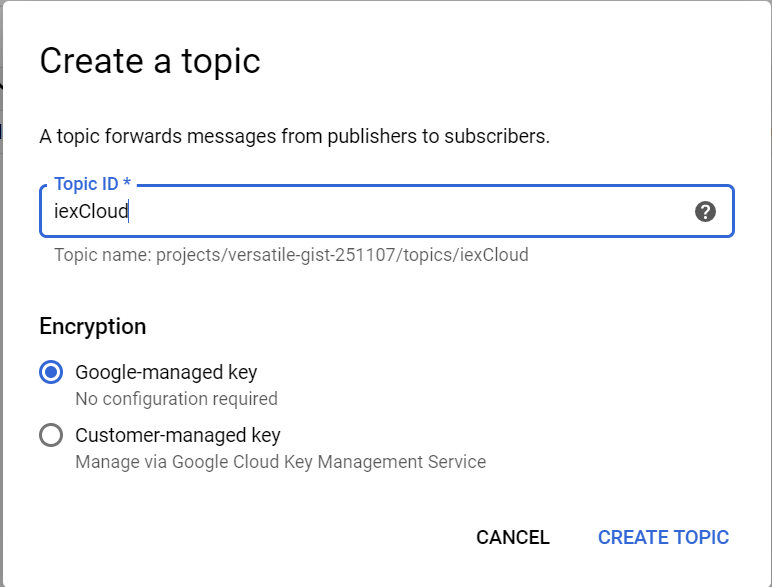
* 1. Now let’s create a dashboard and integrate all visualization panes done. You should obtain something like the below:



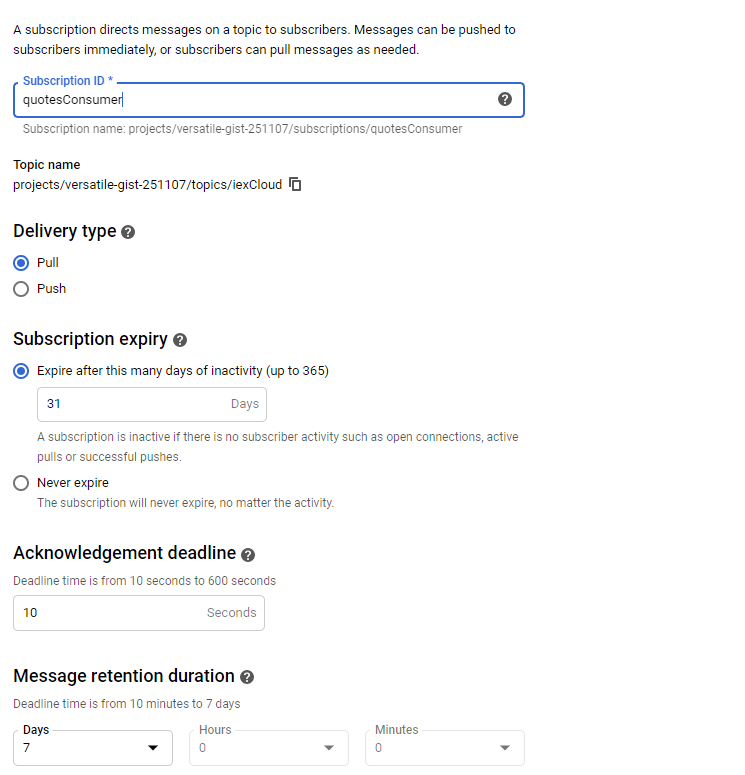
1. In this exercise we will implement an Streaming application to ingest Stock actions realtime changes and store it on a Database for analytics (Big Query). In this Exercise you will setup Google environment pubsub channel and BigQuery dataset to process and store stock prices and IexCloud account to consume this data. .
   1. Logon google cloud -> <https://console.cloud.google.com/> (Use your edem account)
   2. Click on Edem.es combo on the Top and Create a new project, call it Streaming and click on Create.



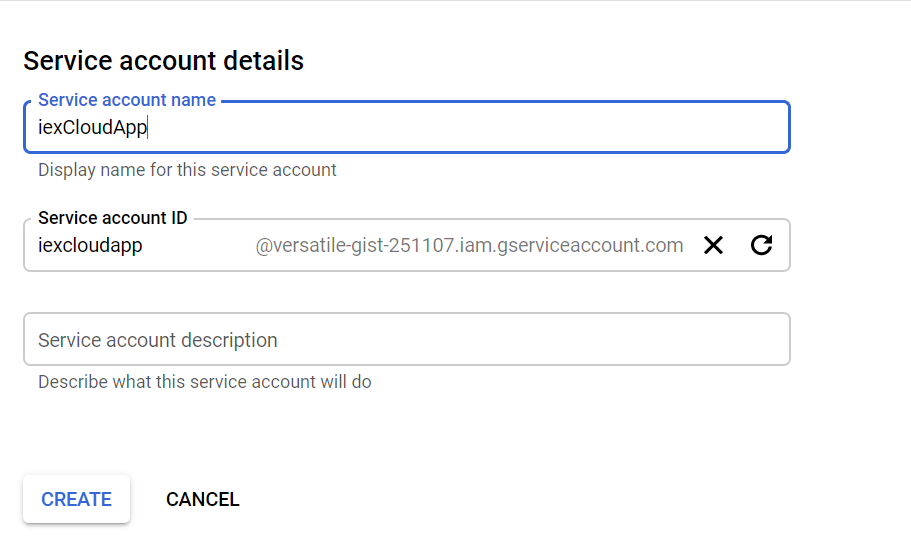
* 1. Now lets create the Streaming channel where we will send stock prices on real time. On the left panel look for Pub/Sub, and click on create Topic. Call it iexCloud.



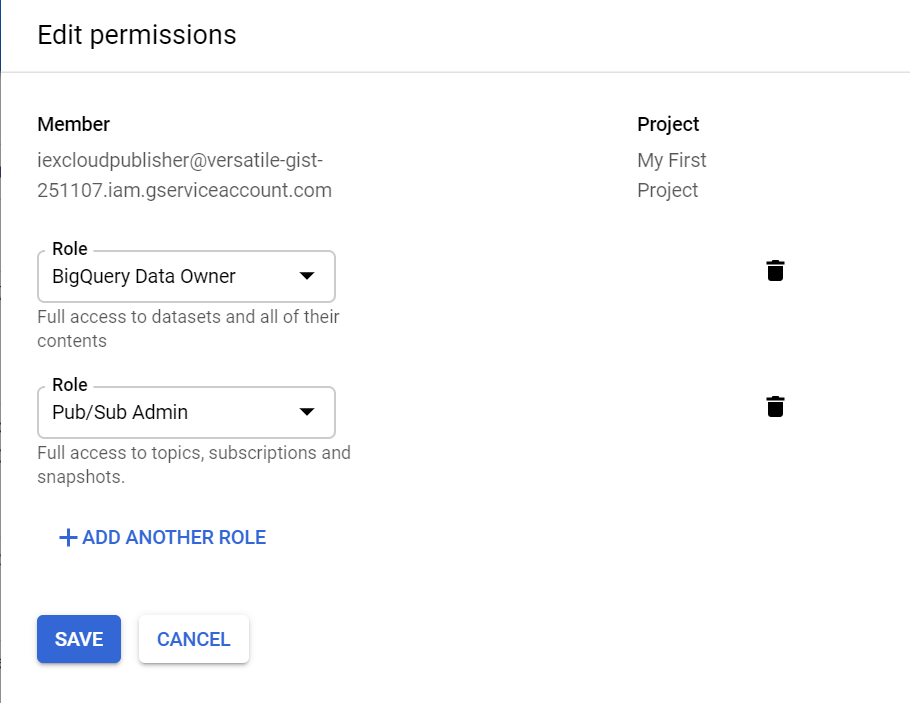
* 1. Let’s create a subscription to consume messages. Call it quotesConsumer and left the rest of parameter as they are.



* 1. Now, let’s create the credentials to be able to send data to pub/sub Topic
     1. Go to Left panel, click on IAM & Admin
     2. Click on Service Account, Click on CREATE SERVICE ACCOUNT
     3. Create a Service account with name iexCloudApp



* + 1. In order to be able to write into Big Query and to consume and publish to pub/sub, you need to add permissions. So on next step add two roles: BigQuery Data Owner and Pub/Sub Admin.



* + 1. Finally click on Create Key , that will download a json file of the credentials to be able to populate messages to Pub/Sub



* 1. Copy the file downloaded to Credentials folder, running following on a terminal:

cp Downloads/credentials\_downloaded.json /home/edem/Credentials/iexCloudApp.json

* 1. Now create a data set called Stocks on BigQuery. On Google Console left panel click on BigQuery section.
  2. Click on CREATE DATASET and call it Stocks, left rest of parameters as they are.
  3. Now let’s create an account on IexCloud, accessing to this link → <https://iexcloud.io/>
     1. Click on Sandbox Testing to have it on.

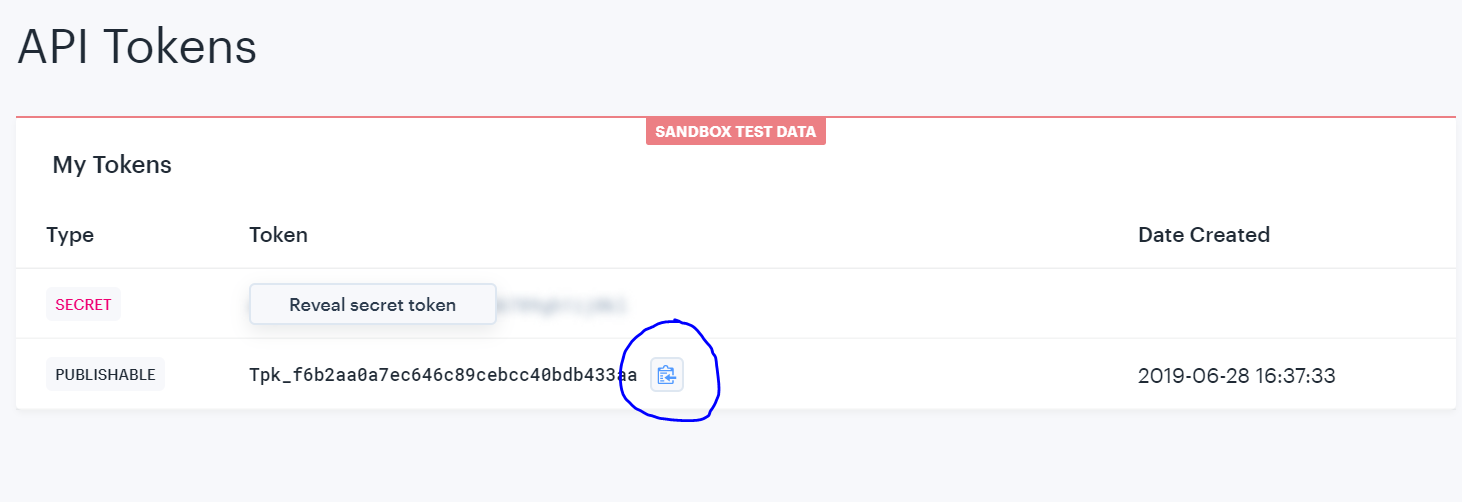
1. Setup NiFi ingestion.
   1. There is an already environment variable pointing to the downloaded credentials. To check this open an terminal and run following:

echo $GOOGLE\_APPLICATION\_CREDENTIALS

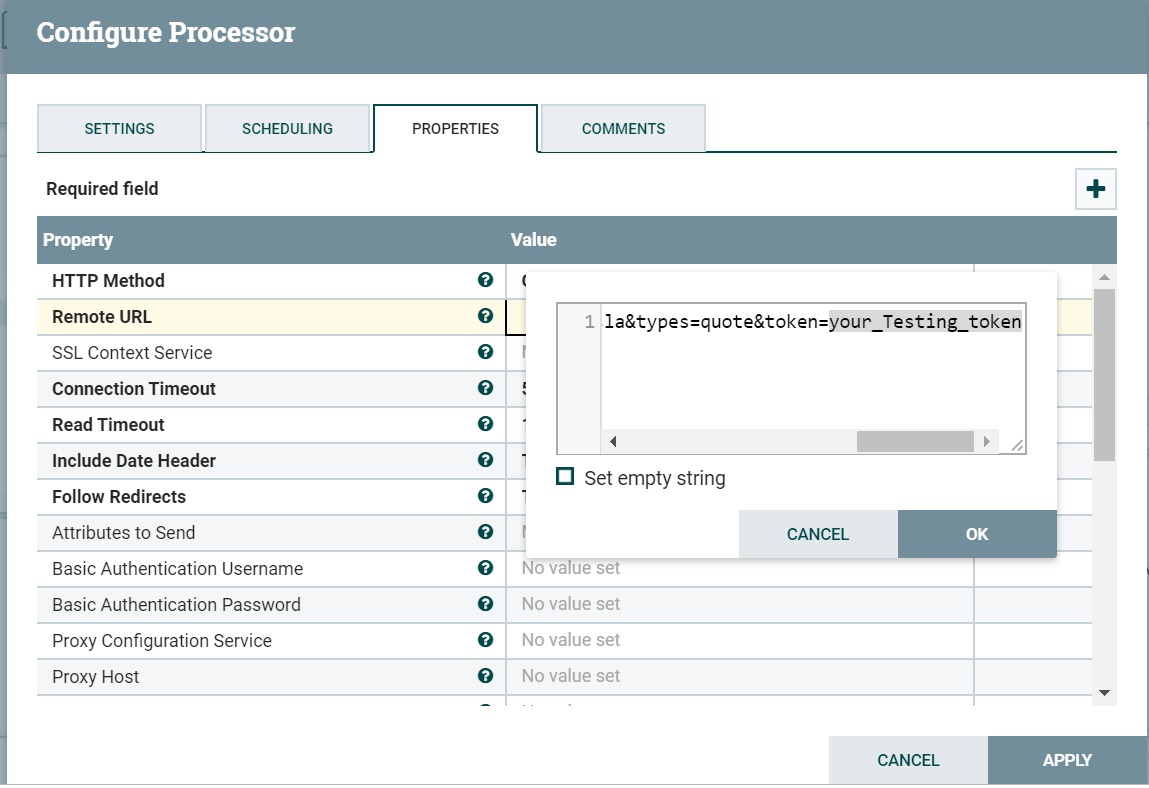
You should get this:

/home/edem/Credentials/iexCloudApp.json

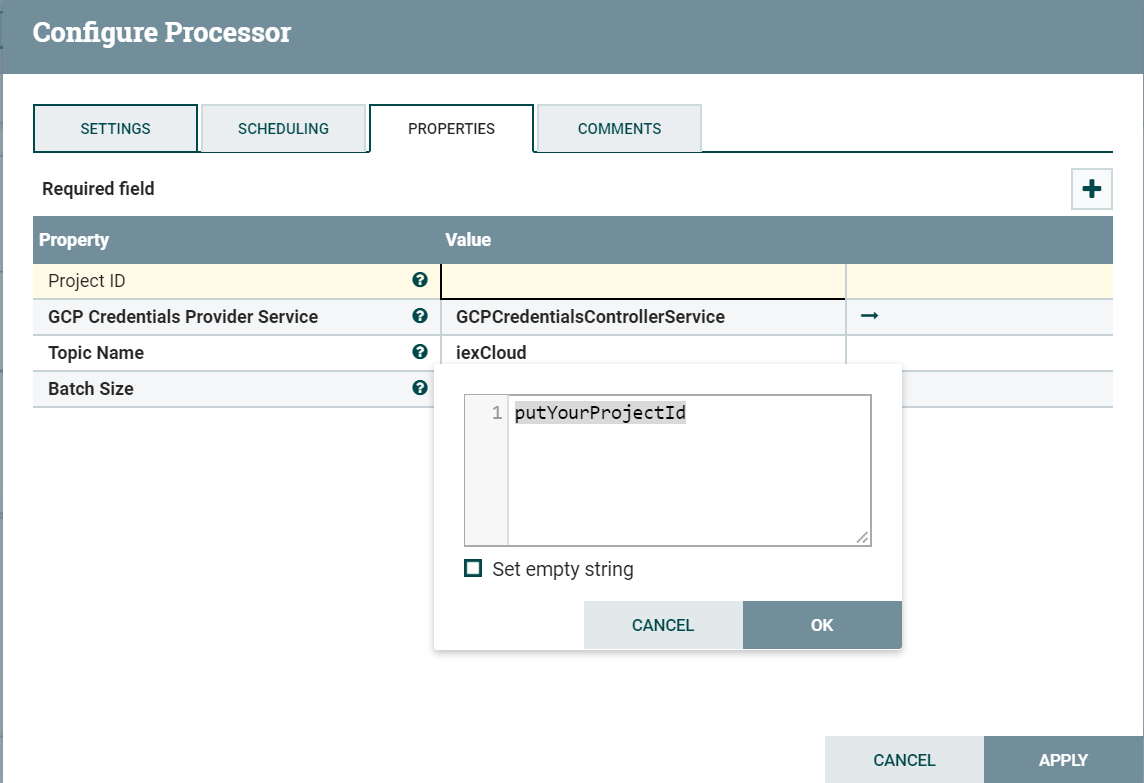
* 1. On same terminal launch NiFi executing the following: “Software/nifi-1.9.2/bin/nifi.sh run”
  2. Now, create an account to consume stock prices on real time via http API REST. For doing that go to your <http://iexcloud.com> account click on API Tokens and copy your publishable token.



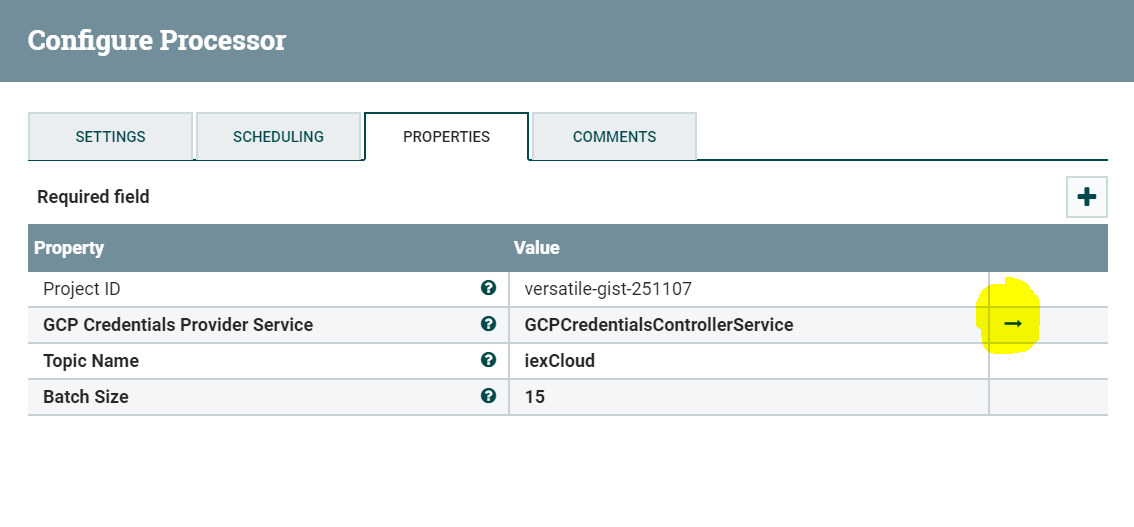
* 1. Open NiFi application <http://localhost:8080/nifi>
  2. Load available Template, do a right click and select the template that is stored on /home/edem/Exercises/Streaming/Nifi/iexCloudTemplate.xml.
  3. Select the template clicking on this top icon and selecting iexCloudTemplate 
  4. Update API REST call to and put your token at the end of the URL. Click on first Box (InvokeHTTP), and click on Remote URL (that’s where you put API Rest Address), and replace “your\_Testing\_token” with the copied token:



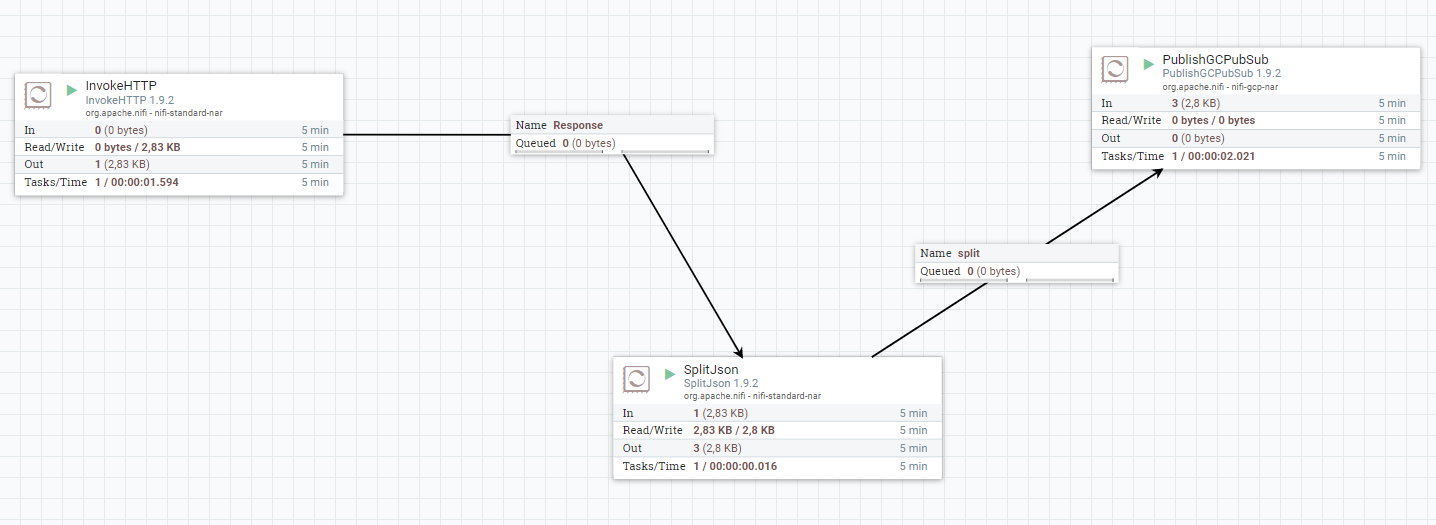
* 1. Configure Pub/Sub publisher. For doing this click on last Box (PublishGCPPubSub), and put the project name that you have created on Google Cloud (Project ID).



* 1. On same panel click on GCPCredentialsControl arrow



* 1. On Services panel enable this Service service.
  2. Just click on play icon and you should see that there are not error messages icon.



* 1. Now let’s check that there are quote messages being populated on pub/sub. To do that open a new terminal and run the following:

*gcloud auth login*

That will open a browser window, use your google account edem and password. Once done, run the following on same terminal, that will consume one message form pubsub and show it on the screen:

*gcloud pubsub subscriptions pull --auto-ack quotesConsumer*

You should get a result like this:

| {"quote":{"symbol":"FB","companyName":"Facebook, Inc.","primaryExchange":"AQDNSA","calculationPrice":"close","open":193.18,"openTime":1631785537228,"close":192.71,"closeTime":1647837998683,"high":197.23,"low":195,"latestPrice":193.67,"latestSource":"Close","latestTime":"November 8, 2019","latestUpdate":1641609105625,"latestVolume":11185807,"iexRealtimePrice":null,"iexRealtimeSize":null,"iexLastUpdated":null,"delayedPrice":198.17,"delayedPriceTime":1574551322030,"extendedPrice":196.8,"extendedChange":-0.04,"extendedChangePercent":-2.2E-4,"extendedPriceTime":1580409283095,"previousClose":192.83,"previousVolume":13590151,"change":0.42,"changePercent":0.0023,"volume":10952382,"iexMarketPercent":null,"iexVolume":null,"avgTotalVolume":14384967,"iexBidPrice":null,"iexBidSize":null,"iexAskPrice":null,"iexAskSize":null,"marketCap":546422219808,"peRatio":30.84,"week52High":209.29,"week52Low":125.26,"ytdChange":0.42207470437950917,"lastTradeTime":1621199497878,"isUSMarketOpen":false}} | 847529191768259 | |

* 1. Finally stop NiFi messages publisher clicking on Stop icon.

1. Now let’s create a Apache Beam Streaming application to consume quote messages and store them on a Big Query table on real time.
   1. Create a Google Storage Bucket on <http://console.cloud.google.com> → Left panel chose Storage → Storage.
      1. Click on Create bucket
      2. Put your bucket name, example edem-roberto.
      3. Leave rest of the options with default values.
   2. Enable Dataflow api
      1. On google console, go to Api & Services, click on ENABLE APIS AND SERVICES, look for Dataflow and enable it.
   3. Open a terminal and Download Beam exercise code
      1. Move to Exercise folder → run cd Exercises
      2. Clone Git repository → run git clone https://github.com/rlopezherrero/GFT-EDEM-MasterData.git
   4. Open Spyder → run spyder.
   5. On Spyder open file beam.py that is present on Exercises/GFT-EDEM-MasterData/AlmacenamientoProcesamiento/Streaming/BamExercise
   6. Follow the instructions on the file
   7. Once completed, click run on Spyder, that will run beam job.
   8. Validate that your data is flowing to Big Query
      1. On google console <http://console.cloud.google.com> , left panned look for Big Query console
      2. Run a query over Quote table on Stock dataset to validate that the quotes records are stored there.
   9. Now let’s deploy it on cloud Dataflow platform
      1. On Beam.py file change the runner to use DataflowRunner instead of DirectRunner.
      2. Click on Play button to deploy it on Google Cloud.
      3. Now go to google console and open Dataflow. You will see your workflow running there.
2. Now we want also to consume intra-day stock prices and store them also on real time on Big Query. For doing that mimic that has been done on previous exercise with following details:
   1. Use following Rest API to get intra-day stock prices → [https://sandbox.iexapis.com/stable/stock/amzn/intraday-prices?token=](https://sandbox.iexapis.com/stable/stock/amzn/intraday-prices?token=Tpk_f6b2aa0a7ec646c89cebcc40bdb433aa)YOUR\_TOKEN
   2. Think on the setup you should do to consume this new data.
   3. Implement Beam job to do that.