Stratio streaming documentation

version 0.7.0-SNAPSHOT

Stratio streaming documentation

January 28, 2015

Contents

Contents:

About Stratio Streaming

Nowadays data-intensive processes and organizations of all sorts require the use of real-time data with increasing flexibility and complexity, so we created Stratio Streaming to meet this demand. Stratio Streaming is one of the core modules on which the Stratio platform is based. Stratio Streaming is the union of a real-time messaging bus with a complex event processing engine using Spark Streaming. Thanks to this technology, Stratio Streaming allows the creation of streams and queries on the fly, sending massive data streams, building complex windows over the data or manipulating the streams in a simple way by using an SQL-like language. Stratio Streaming's API masks the complexity of the system so that developers can work with live streams straightaway. The engine Stratio Streaming also offers built-in solution patterns which solve typical use cases and common problems related to streaming. Additionally, we have added global features to the engine such as auditing and statistics.

Introduction

Many big data applications must process batch scenarios combined with large streams of live data and provide results in near-real-time, but the lack of an hybrid framework (batch/real) forces the companies to double the effort to implement new functions.

Existing traditional streaming systems have a event-driven-record-at-a-time processing model, keeping states by record. Making stateful streams processing be fault-tolerant is challenging.

With the advent of Apache Spark, all these issues and challenges have been resolved and now there is a new and powerful way to design distributed, scalable and fault-tolerant systems. For that reason, Stratio Streaming Engine is based on Apache Spark Streaming.

But real-time and big data require a high level of flexibility, performance, powerful features and on-demand queries, so we have included Complex Event Processing capabilities and real-time operations.

This way, you can use Stratio Streaming to define or alter your streams on the fly, launch a complex query with sliding temporal windows, or save your streams to Apache Cassandra… and these are only a few examples of the features available.

Features

Operations on streams:

Streams are conceived are event flows. They have a known definition including typed fields and they should be use as flowing channels of events. In other words, streams have an in-flows and output-flows.

You can see streams as SQL tables in a relational database, but streams are created in an ephemeral environment and no relationship or constraint rules are applied on them.

Stratio Streaming lets you *create, alter or drop streams* on the fly.

Queries on streams:

You can define queries on streams in real-time, including Complex Event Processing operators, such as:

- Filters: filtering events by the fields.
- Windows: restrict the query to time sliding windows or length sliding windows.
- Event sequences: Identifying sequences of events in strict order.
- Event patterns: Identifying events that match a given pattern, not necessarily ordered.
- Stream union: Joining streams to identify common events in isolated streams.

- Projection, group by and built-in functions: group by, having, sum, avg, count operators, among others, are available.

Actions on streams:

There are some built-in operations ready to use on streams:

- Persistence: using Apache Cassandra as long-term storage.
- Statistics: throughput by operation.
- Auditing: using Apache Cassandra as long-term storage.

Stream Query Language

In order to use all these features, Stratio Streaming has a SQL-like language called Stream Query Language.

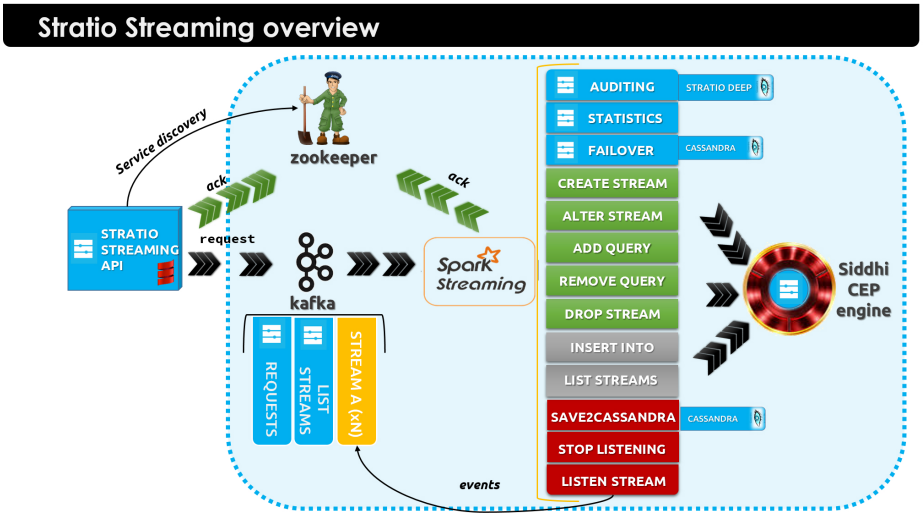
This way, the previous explained features can be organized in a similar way than SQL:

- Stream Definition Language (SDL):
  - Add and remove queries.
  - Create, alter or drop a stream.
- Stream Manipulation Language (SML):
  - Insert events into a stream
  - List existing streams in the engine.
- Stream Action Language (SAL):
  - Manage listeners on stream (start listen, stop listen).
  - Stream persistence, i.e. write stream to Cassandra.
- Built-in functions:
  - Auditing all the request in the streaming engine.
  - Statistics (request per operation, request per stream ...).

Architecture

Stratio Streaming is composed of three main elements:

- A Scala API.
- A publish-subscribe messaging system Apache Kafka.
- A Streaming and CEP engine Apache Spark Streaming and Siddhi CEP.



Where to go from here

To explore and play with Stratio Streaming, we recommend to visit the following:

- Writing and Running a Basic Application for Stratio Streaming: a step by step tutorial to write an application using Stratio Streaming API.
- Using Stratio Streaming API: snippets in Java and Scala.
- Stratio Streaming sandbox and demo

Using Stratio Streaming API

Creating and initializing the API

Scala:

```
scala val stratioStreamingAPI = StratioStreamingAPIFactory.create().initialize
```

Java:

```
IStratioStreamingAPI stratioStreamingAPI = StratioStreamingAPIFactory.create().initialize();
```

Creating and initializing the API with server configuration

Scala:

```
val stratioStreamingAPI = StratioStreamingAPIFactory.create()
     .initializeWithServerConfig("stratio.node.com", 9092, "stratio.node.com", 2181)
```

Java:

```
IStratioStreamingAPI stratioStreamingAPI = StratioStreamingAPIFactory.create()
     .initializeWithServerConfig("stratio.node.com", 9092, "stratio.node.com", 2181);
```

Creating a new stream

Scala:

```
val firstStreamColumn = new ColumnNameType("column1", ColumnType.INTEGER)
val secondStreamColumn = new ColumnNameType("column2", ColumnType.STRING)
val streamName = "testStream"
val columnList = Seq(firstStreamColumn, secondStreamColumn)
try {
    stratioStreamingAPI.createStream(streamName, columnList)
} catch {
    case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
ColumnNameType firstStreamColumn= new ColumnNameType("column1", ColumnType.INTEGER);
ColumnNameType secondStreamColumn = new ColumnNameType("column2", ColumnType.STRING);
String streamName = "testStream";
List columnList = Arrays.asList(firstStreamColumn, secondStreamColumn);
try {
      stratioStreamingAPI.createStream(streamName, columnList);
} catch (StratioStreamingException e) {
      e.printStackTrace();
}
```

Adding columns to an existing stream

Scala:

```
val newStreamColumn = new ColumnNameType("column3", ColumnType.DOUBLE)
val streamName = "testStream"
 val columnList = Seq(newStreamColumn)
 try {
        stratioStreamingAPI.alterStream(streamName, columnList)
 } catch {
        case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
 }
```

Java:

```
ColumnNameType thirdStreamColumn= new ColumnNameType("column3", ColumnType.DOUBLE);
String streamName = "testStream";
List columnList = Arrays.asList(thirdStreamColumn);
try {
     stratioStreamingAPI.alterStream(streamName, columnList);
} catch (StratioStreamingException e) {
     e.printStackTrace();
}
```

Inserting data into a stream

Scala:

```
val streamName = "testStream"
val firstColumnValue = new ColumnNameValue("column1", new Integer(1))
val secondColumnValue = new ColumnNameValue("column2", "testValue")
val thirdColumnValue = new ColumnNameValue("column3", new Double(2.0))
val streamData = Seq(firstColumnValue, secondColumnValue, thirdColumnValue)
try {
     stratioStreamingAPI.insertData(streamName, streamData)
} catch {
    case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
String streamName = "testStream";
ColumnNameValue firstColumnValue = new ColumnNameValue("column1", new Integer(1));
ColumnNameValue secondColumnValue = new ColumnNameValue("column2", "testValue");
ColumnNameValue thirdColumnValue = new ColumnNameValue("column3", new Double(2.0));
List<ColumnNameValue> streamData = Arrays.asList(firstColumnValue, secondColumnValue, thirdC
try {
    stratioStreamingAPI.insertData(streamName, streamData);
} catch(StratioStreamingException ssEx) {
    ssEx.printStackTrace();
}
```

Adding queries to streams

Scala:

```
val query = "from testStream select column1, column2, column3 insert into alarms for current
val streamName = "testStream"
try {
    val queryId = stratioStreamingAPI.addQuery(streamName, query)
} catch {
   case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
String streamName = "testStream";
String query = "from testStream select column1, column2, column3 insert into alarms for curr
try {
    String queryId = stratioStreamingAPI.addQuery(streamName, query);
} catch(StratioStreamingException ssEx) {
    ssEx.printStackTrace();
}
```

Removing an existing stream

Scala:

```
val streamName = "testStream"
try {
     stratioStreamingAPI.dropStream(streamName)
} catch {
    case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
String streamName = "testStream";
try {
    stratioStreamingAPI.dropStream(streamName);
} catch(StratioStreamingException ssEx) {
     ssEx.printStackTrace();
}
```

Removing an existing query from a stream

Scala:

```
val streamName = "testStream"
val queryId = "alarms-657c1720-1869-4406-b42a-96b2b8f740b3"
try {
    stratioStreamingAPI.removeQuery(streamName, queryId)
} catch {
   case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
String streamName = "testStream";
String queryId = "alarms-f6bd870f-2cbb-4691-ba2c-ef4392e70a1b";
try {
    stratioStreamingAPI.removeQuery(streamName, queryId);
} catch(StratioStreamingException ssEx) {
     ssEx.printStackTrace();
}
```

Listening to streams

Scala:

```
try {
  val streams = stratioStreamingAPI.listenStream("testStream")
  for(stream  {
         println("Column: "+column.getColumn)
         println("Value:"+column.getValue)
         println("Type: "+column.getType)}
      )
  }
```

```
} catch {
   case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
try {
 KafkaStream<String, StratioStreamingMessage> streams = stratioStreamingAPI.listenStream("te
 for (MessageAndMetadata stream: streams) {
    StratioStreamingMessage theMessage = (StratioStreamingMessage)stream.message();
    for (ColumnNameTypeValue column: theMessage.getColumns()) {
       System.out.println("Column: "+column.getColumn());
       System.out.println("Value: "+column.getValue());
       System.out.println("Type: "+column.getType());
    }
 }
} catch(StratioStreamingException ssEx) {
   ssEx.printStackTrace();
}
```

Stop listening to streams

Scala:

```
try {
    stratioStreamingAPI.stopListenStream("testStream")
} catch {
   case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
try {
   stratioStreamingAPI.stopListenStream("testStream");
} catch(StratioStreamingException ssEx) {
   ssEx.printStackTrace();
}
```

Save the stream to Cassandra

Scala:

```
try {
    stratioStreamingAPI.saveToCassandra("testStream")
} catch {
   case ssEx: StratioStreamingException => println(ssEx.printStackTrace())
}
```

Java:

```
try {
   stratioStreamingAPI.saveToCassandra("testStream");
} catch(StratioStreamingException ssEx) {
   ssEx.printStackTrace();
}
```

Getting the list of all the streams and their queries

Scala:

```
import scala.collection.JavaConversions._

val listOfStreams = stratioStreamingAPI.listStreams().toList
```

```
println("Number of streams: "+listOfStreams.size)
listOfStreams.foreach(stream => {
   println("--> Stream name: "+stream.getStreamName)
   if ( stream.getQueries.size > 0 ) {
     stream.getQueries.foreach(query =>
       println("Query: "+query.getQuery))
   }
})
```

Java:

```
List<StratioStream> streamsList = stratioStreamingAPI.listStreams();
System.out.println("Number of streams: " + streamsList.size());
for (StratioStream stream: streamsList) {
   System.out.println("--> Stream Name: "+stream.getStreamName());
   if ( stream.getQueries().size() > 0 ) {
      for (StreamQuery query: stream.getQueries())
        System.out.println("Query: "+query.getQuery());
      }
}
```

Writing and Running a Basic Application for Stratio Streaming

In this tutorial you will learn how to write a java or scala project for building Stratio Streaming applications and how to run it on a local instance or a standalone cluster. Instructions are based on the Eclipse environment but any equivalent can be used.

Before you start

Prerequisites

- Eclipse or an equivalent IDE.
- Oracle JDK 7.
- Apache Maven: Stratio Streaming API is available in a Maven repository that will be used in this tutorial.
- Scala >=2.10.3.
- Scala-IDE: follow instructions at Eclipse marketplace to install it from the marketplace (recommended over downloading the plugin from scala-ide.org).
- m2eclipse-scala plugin: follow instructions at scala-ide.org for installation.

Resources

Here is a list of the resources that will be used in this tutorial. You can download them now or as you go through the instructions. Links will be provided later as they will be needed.

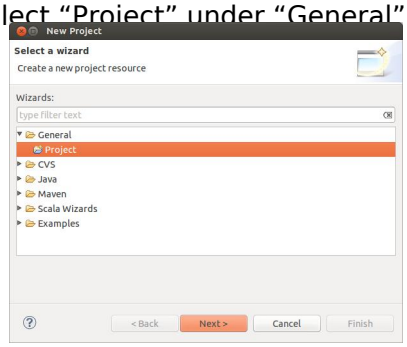
- **Java project example**.
- **Scala project example**.

Creating the project

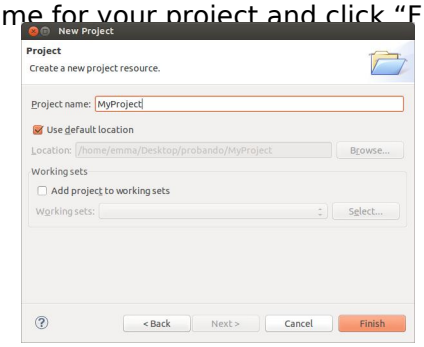
Step 1: Create an empty project

- Launch Eclipse and in the menu choose File -> New -> Project

- In the “New project” window select “Project” under “General” and click “Next”:



- In the next window, enter a name for your project and click “Finish”:



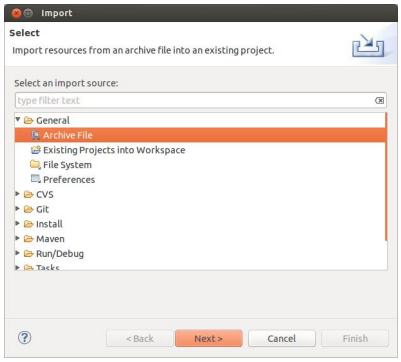
The newly created project now appears in the package explorer.

Step 2: Import the project skeleton

Download the project skeleton of your choice and save it in a convenient location:

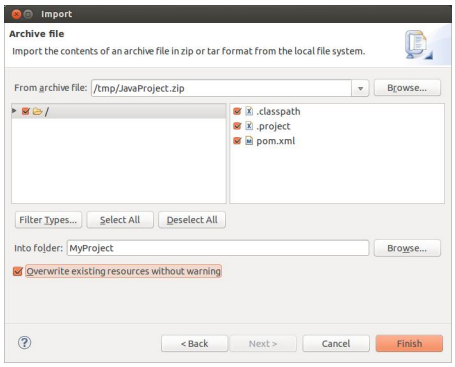
- **`Java project example`**.
- **`Scala project example`**.

In the menu, choose File -> Import. In the “Import” window, select “Archive file” in the section “General”, and click “Next”:



In the next screen:

- Navigate to the zip file you just downloaded using the “Browse...” button.
- Fill in “Into folder” with the name of the project (or use the “Browse...” button to select it from a list).
- Check “Overwrite existing resources without warning”,
- and click “Finish”



The structure of the project will be displayed in the package explorer. Give Maven some time to check and download dependencies. The project should finally appear error-free.

The java project contains an example class (JavaExample.java) and an example test (TestJava.java), the scala one an example object (ScalaExample.app) and an example test (TestScala.scala), the mixed project contains all the formers.

Navigate through your project to get familiar with it. You can add your own code and optionally alter the stream, insert data, add queries and listeners.

Running the application

You can run the example provided in the project directly on your IDE. To do so, right click on the JavaExample.java file -> Run As -> Java Application.

For the Java or Scala project, the result should be similar to the following:

```
shell-session Streams in the Stratio Streaming Engine: 3
    -- Stream Name: stratio_stats_base
    -- Stream Name: stratio_stats_global_by_operation
    -- Stream Name: testStream
```

Congratulations! You successfully completed this tutorial.

Where to go from here

If you are planning to write your own Stratio Streaming application, Using Stratio Streaming API may be useful. Those are snippets written in both Java and Scala.

Stratio Streaming sandbox and demo

Vagrant Setup

To get an operating virtual machine with stratio streaming distribution up and running, we use Vagrant.

- Download and install Vagrant.
- Download and install VirtualBox.
- If you are in a windows machine, we will install Cygwin.

Running the sandbox

Create any system folder and using the command line, type vagrant init stratio/streaming.

To facilitate the reading of the document , we will refer to this directory as /install-folder.

Please, be patient the first time it runs.

What you will find in the sandbox

- OS: CentOS 6.5
- 3GB RAM - 2 CPU
- Two ethernet interfaces.

Name|Version|Service name|Other Stratio Streaming|{{site.projects`2].version}}|stratio-streaming|service streaming start Stratio Streaming Shell|{{site.projects`2].version}}|-|/opt/sds/streaming-shell/bin Apache Kafka|0.8.1.1|kafka|service kafka start Apache Zookeeper|3.4.6|zookeeper|service zookeeper start Stratio Cassandra|2.1.05|cassandra|service cassandra start Elasticsearch|1.3.2|elasticsearch|service elasticsearch start Kibana|3.1.0|-|http://10.10.10.10/kibana Mongodb|2.6.5|mongod|service mongod start Apache Web Server|2|httpd|service httpd start

Access to the sandbox and other useful commands

Useful commands

- Start the sandbox: ` vagrant up `
- Shut down the sandbox: ` vagrant halt `
- In the sandbox, to exit to the host: ` exit `

Accessing the sandbox

- Located in /install-folder
- ` vagrant ssh `

Starting the Stratio Streaming Shell and other useful commands

From the sandbox (vagrant ssh):

- Starting the Stratio Streaming Shell: /opt/sds/streaming-shell/bin/shell
- List all available commands: help
- Exit the Stratio Stratio Streaming Shell: exit

F.A.Q about the sandbox

I am in the same directory that I copy the Vagrant file but I have this error::

```
A Vagrant environment or target machine is required to run this
command. Run vagrant init to create a new Vagrant environment. Or,
get an ID of a target machine from vagrant global-status to run
this command on. A final option is to change to a directory with a
Vagrantfile and to try again.
```

Make sure your file name is Vagrantfile instead of Vagrantfile.txt or VagrantFile.

When I execute vagrant ssh I have this error::

```
ssh executable not found in any directories in the %PATH% variable. Is an
SSH client installed? Try installing Cygwin, MinGW or Git, all of which
contain an SSH client. Or use your favorite SSH client with the following
authentication information shown below:
```

We need to install Cygwin or Git for Windows.

Stratio Streaming Demos

Demo #1: Sensor Monitoring

This demo will show up some of the features of Stratio Streaming, an interactive CEP engine built with Apache Spark and Apache Siddhi, such as:

- Use time-based and event-length sliding windows (with aggregation functions)
- Launch some queries in order to control thresholds and insert events in other streams
- The use of derived streams (streams whose definition is inferred by the engine, because it is implicit in the queries)
- Start actions on streams

This CEP use case is oriented to track measures in a sensor monitoring environment, but Complex Event Processing could be succesfully applied to other scenarios and use cases, such as fraud detection, real-time applications and systems monitoring, stock-quote analysis or ciber-security, among others.

In this demo, you will use the following components and features:

- The Stratio Streaming Engine, taking care of all the real-time processing and all the CEP operations.
- The Stratio Streaming Shell, in order to interact with the engine in real-time.
- The Stratio Streaming API, in order to send simulated sensor measures to the engine.
- The INDEX action over several streams, in order to send all the events in a stream to a data storage, in this case Elastic Search.
- Kibana web application as a real-time monitor of the entire system.

To put all these pieces to work, you need to:

- Write some commands in the Stratio Streaming Shell to create all the streams, queries and actions.
- Simulate some random sensor measures related to basic signals of a system (cpu, memory, processes...)
- And lastly, visualize all the indexed data in real-time.

Shell steps

- vagrant ssh
- /opt/sds/streaming-shell/bin/shell
- Creation of a base stream, where we are going to insert all the sensor

measures. A stream definition is similar to a table, with field definition and types:

```
create --stream sensor_grid --definition "name.string,data.double"
```

- List command allow us to check out the current state of the CEP

engine. How many streams and queries are already created?, Which actions are enabled on a stream?, What is the definition of a stream?:

```
list
```

- By launching this query we are aggregating the sensor measures in windows based on event length (250 events), so that

we can get an average measure by each sensor type. This is a continuous query, it will be registered from now in the engine, unlike the classic request/response model of the relational databases. In addition, the result of the query will be inserted in another stream, whose definition is inferred from the query's projection. That means that you don't have to explicitly create the output stream. The engine will infer the definition of the stream and create it automatically.:

```
add query --stream sensor_grid --definition "from sensor_grid#window.length(250) select name
```

- We request the engine to start one of the available actions on the

base stream that we have previously created. In particular the one that send all the events in this stream to Elastic Search. Actions can be enabled and disabled in any moment, and there are actions ready to use such us saving the events into Cassandra, MongoDB or ElasticSearch. In addition, there is an special action called LISTEN that send events to an specific topic on Kafka whose name is the same as the stream in which the action has been enabled.:

```
index start --stream sensor_grid_avg
```

- Now, by doing a “list”, we can check out that there are two streams, one query and the

stream called “sensor_grid” has an action enabled, INDEX.:

```
list
```

- Now, working on the aggregated measures, we will launch two queries that use operators to filter and set thresholds on

events. Furthermore, we will use time-based windows to fire alarms if these thresholds are reached only in an specific period of time. The output of these queries is sent to the same new stream, again infered by the engine.

```
name=='cpu' and data > 80]#window.timeBatch(10 seconds)  select name, avg(data) as data, 'Alarm_inte
```

```
e=='memory' and data > 75]#window.timeBatch(5 seconds)  select name, avg(data) as data, 'Alarm_inte
```

```
nd data > 80) or (name=='cpu' and data > 90)]#window.timeBatch(15 seconds) select name , avg(data) a
```

- Let’s start indexing the alarms, too:

```
index start --stream sensor_grid_alarms
```

- If you want, you can start inserting one event by using the shell:

```
insert --stream sensor_grid --values "name.cpu,data.33"
```

- We are done with the shell.:

```
exit
```

Sensor grid simulation steps

- Now, let’s send some bulk data to the engine. All the measures are

fake but we are producing random variations on them, in order to simulate the behaviour of a real system:

```
sudo sh  /opt/sds/streaming-examples/bin/hardware-emulator 2 streaming.stratio.com:9092
```

- You can launch this tool as many times as you want.

Dashboard steps

- Open a browser on your machine and go here: http://[SANDBOX_IP]/kibana/index.html#/dashboard/file/sensor-grid-monitoring.json
- Thanks to this real-time dashboard, you can watch all the things happening inside the engine. All the aggregated events, alarms in some fancy widgets.

Extra: Streaming metrics

Stratio Streaming is the result of combining the power of Spark Streaming as a continuous computing framework and Siddhi CEP engine as complex event processing engine. This dashboard is showing some statistics related to the status of the Stratio Streaming engine, allowing you to inspect commands, events and throughput, in a real-time panel. This way, we took advantage of the engine itself to take care of all the internal events produced by the engine. In order to get this dashboard working, please execute the following commands:

- To start we need change some properties into streaming engine configuration.:

```
sudo vi /etc/sds/streaming/config.conf
```

- Set statsEnabled property to true.
- Now, is necessary to restart streaming service.:

```
sudo service streaming restart
```

- Using the shell, execute this commands:

```
/opt/sds/streaming-shell/bin/shell
```

- You can execute into shell the list command and you should obtain this result:

```
stratio-streaming> list
Stream name                  User defined  Queries  Elements  Active actions
---------------------------  ------------  -------  --------  --------------
streaming-gauge-metrics      false         0        3         []
streaming-counter-metrics    false         0        3         []
streaming-histogram-metrics  false         0        13        []
streaming-meter-metrics      false         0        8         []
streaming-timer-metrics      false         0        19        []
```

- Execute this commands to index all metric streams:

```
index start --stream streaming-gauge-metrics
index start --stream streaming-meter-metrics
index start --stream streaming-counter-metrics
index start --stream streaming-histogram-metrics
index start --stream streaming-timer-metrics
```

- **Now, you can access to metrics kibana dashboard:**
  http://[SANDBOX_IP]/kibana/index.html#/dashboard/file/streaming-status.json