Hands-on 3:
Stream processing with Spark

1. Objective

The objective of this hands on is to let you "touch" the challenges implied in processing streams. In class, we will use Spark for implementing a streaming version of word count and an example using:

- A TCP server.
- Twitter streaming.

2. Material

- Download https://github.com/javieraespinosa/dxlab-spark

3. Getting started with Spark Streaming

Spark streaming is an extension of the core Spark API that enables stream processing of live data streams. Data can be harvested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to file systems, databases, and live dashboards

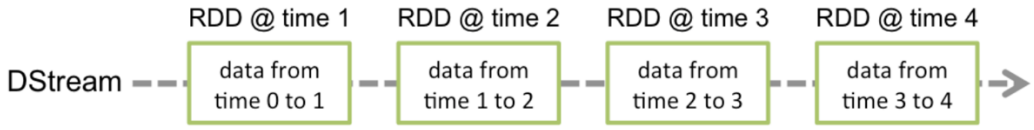
Internally Spark streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

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

This guide shows you how to start writing Spark Streaming programs with *DStreams*. You can write Spark Streaming programs in Scala, Java or Python (see the Spark full guide for details https://spark.apache.org/docs/latest/streaming-programming-guide.html). You will find tabs throughout this guide that let you choose between code snippets of different languages.

4. Basic concepts: DStreams

Discretized Stream is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs, which is Spark's abstraction of an immutable, distributed dataset. Each RDD in a DStream contains data from a certain interval, as shown in the following figure.



Any operation applied on a DStream translates to operations on the underlying RDDs. These underlying RDD transformations are computed by the Spark engine. The DStream operations hide most of these details and provide the developer with a higher-level API for convenience.

4.1.1 Input DStreams and Receivers

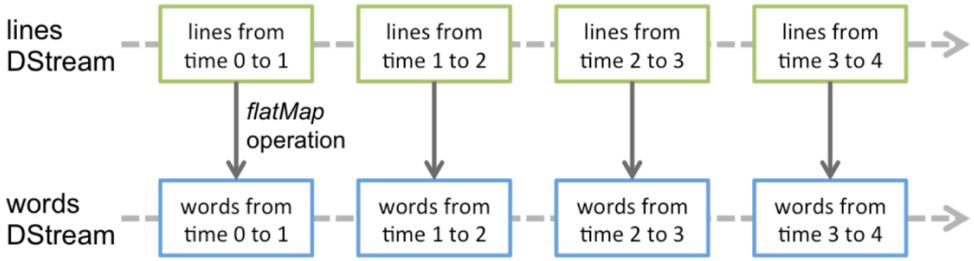
Input DStreams are DStreams representing the stream of input data received from streaming sources. Every input DStream (except file stream) is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing. Spark Streaming provides two categories of built-in streaming sources.

- *Basic sources*: directly available in the StreamingContext API. Examples: file systems, socket connections, etc.
- *Advanced sources*: Sources like Kafka, Flume, Kinesis, Twitter, etc. are available through extra utility classes. These require linking against extra dependencies (see the Twitter exercise next).

Note that, if you want to receive multiple streams of data in parallel in your streaming application, you can create multiple input DStreams. This will create multiple receivers which will simultaneously receive multiple data streams. Yet, note that a Spark worker/executor is a long-running task, hence it occupies one of the cores allocated to the Spark Streaming application. Therefore, it is important to remember that a Spark Streaming application needs to be allocated enough cores (or threads, if running locally) to process the received data, as well as to run the receiver(s).

4.2 Counting words example

Let us take a quick look at what a simple Spark Streaming program looks like. Let us say we want to count the number of words in text data received from a data server. Note that for converting a stream of lines to words, the flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream as shown in the following figure.



We will develop two versions, the first one listening from a TCP socket and the other one listening from Twitter. For this exercise, we will work in Python. Note that the full spark guide has examples using Scala and Java.

5. Counting words produced by a TCP socket

Terminal 1. Netcat server

Go to you dxlab-spark-master file.

```
$ docker-build
```

Run the TCP server.

```
$ docker-compose run netcat nc -l -p 9999
```

Terminal 2. Spark interpreter for python (pyspark)

```
$ docker-compose run pyspark
```

First, we import `StreamingContext`, which is the main entry point for all streaming functionality. We create a local `StreamingContext` with two execution threads, and batch interval of 1 second.

```
from pyspark import SparkContext

from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 s
econd

ssc = StreamingContext(sc, 1)
```

Using this context, we can create a *DStream* that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. `9999`).

```
# Create a DStream that will connect to hostname:port, like localhost:9999

lines = ssc.socketTextStream("11.0.0.10", 9999)
```

This `lines` *DStream* represents the stream of data that will be received from the data server. Each record in this *DStream* is a line of text. Next, we want to split the lines by space into words.

```
# Split each line into words

words = lines.flatMap(lambda line: line.split(" "))
```

`flatMap` is a one-to-many *DStream* operation that creates a new *DStream* by generating multiple new records from each record in the source *DStream*. In this case, each line will be split into multiple words and the stream of words is represented as the `words` *DStream*. Next, we want to count these words.

```
# Count each word in each batch

pairs = words.map(lambda word: (word, 1))

wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console

wordCounts.pprint()

ssc.start()             # Start the computation
```

```
ssc.awaitTermination()  # Wait for the computation to terminate
```

The `words` *DStream* is further mapped (one-to-one transformation) to a *DStream* of `(word, 1)` pairs, which is then reduced to get the frequency of words in each batch of data. Finally, `wordCounts.pprint()` will print a few of the counts generated every second.

Test your exercise by providing words in the server side, what happens in the client side?

Look for a representative set of texts that can let you see the word count in action.

6. Counting words from Twitter posts

These exercises are designed as standalone Scala programs which will receive and process Twitter's real sample tweet streams. This section will first introduce you to the basic system setup of the standalone Spark Streaming programs, and then guide you through the steps necessary to create Twitter authentication tokens necessary for processing Twitter's real time sample stream.

6.1 Twitter credentials setup

Our hand-on is based on Twitter's sample tweet stream, so we need to configure authentication with a Twitter account. To do this, you need to setup a consumer `key+secret` pair and an access `token+secret` pair using a Twitter account.

6.2 Creating a temporary Twitter access keys

Follow the instructions below to setup these temporary access keys with your Twitter account. These instructions will not require you to provide your Twitter username/password. You will only be required to provide the consumer key and access token pairs that you will generate, which you can easily destroy once you have finished the tutorial. So, your Twitter account will not be compromised in any way.

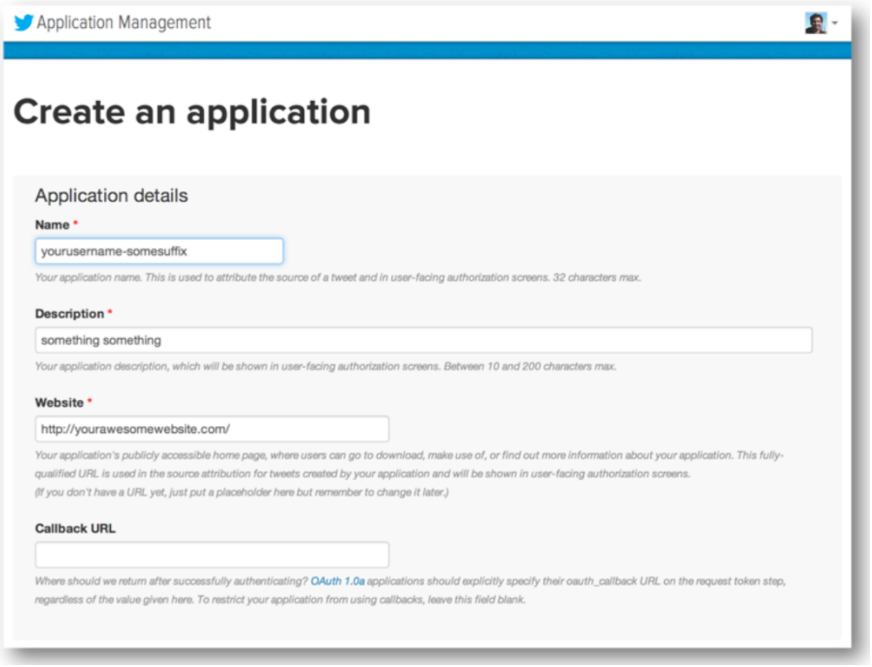
Open Twitter's Application Settings page[1]. This page lists the set of Twitter-based applications that you own and have already created consumer keys and access tokens for. This list will be empty if you have never created any applications. For this tutorial, create a new temporary application. To do this, click on the blue "*Create a new application*" button. The new application page should look the page shown below.

Provide the required fields:

- The **`Name`** of the application must be globally unique, so using your Twitter username as a prefix to the name should ensure that. For example, set it as [your-twitter-handle]-test.
- For the **`Description`**, anything longer than 10 characters is fine.
- For the **`Website`**, similarly, any website is fine, but ensure that it is a fully-formed URL with the prefix *`http://`*.

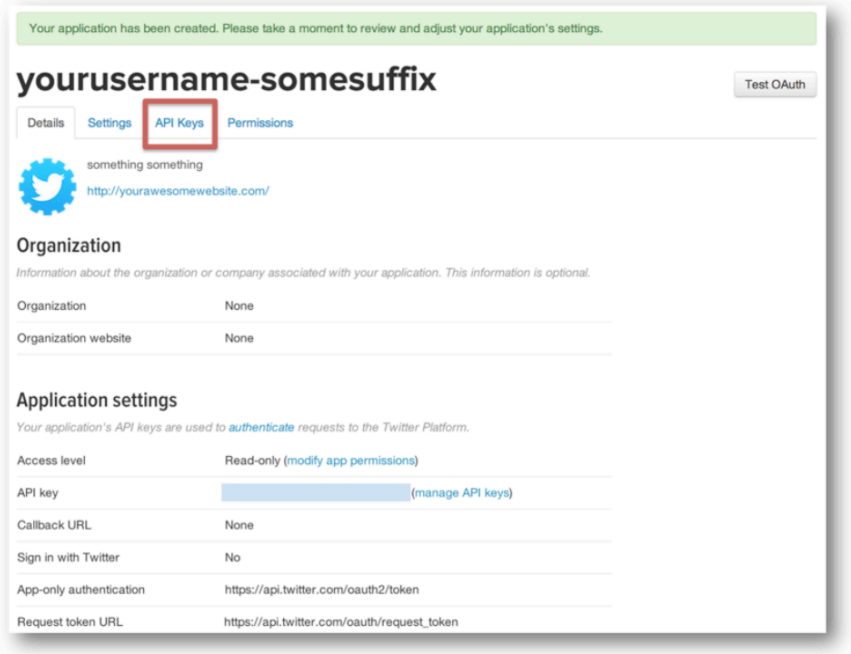
Then, click on the "*Yes, I agree*" checkbox below the **`Developer Rules of the Road`**. Finally, fill in the `CAPTCHA` and click on the blue "*Create your Twitter application*" button.

[1] https://apps.twitter.com



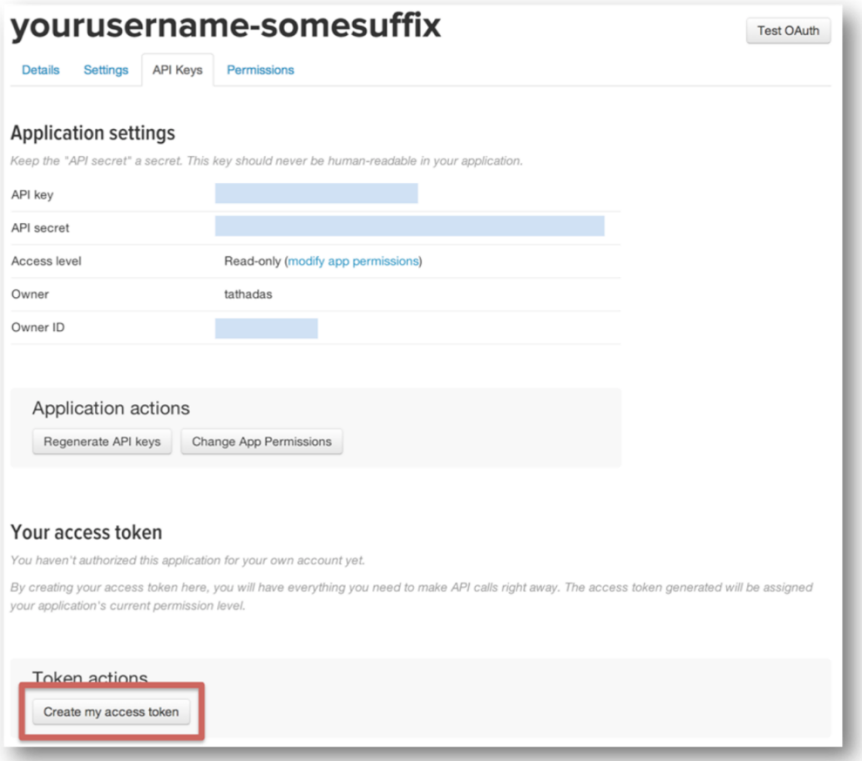
Confirmation

Once you have created the application, you will be presented with a confirmation page like the one shown below. Click on the **API Key** tab.



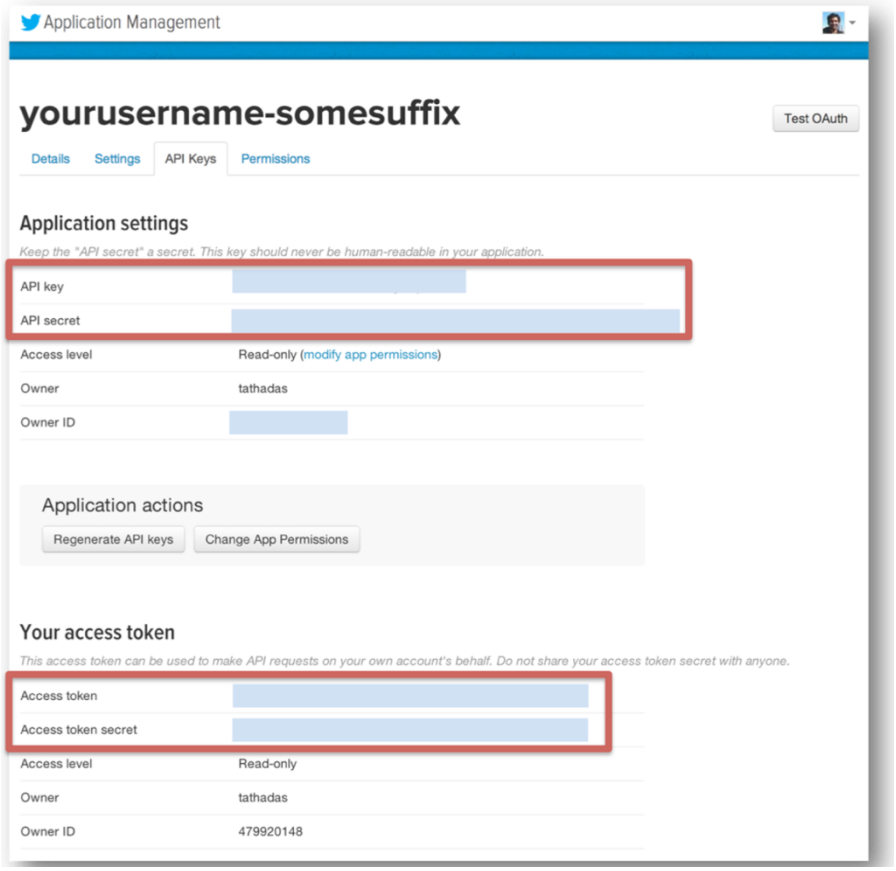
Application settings

You should be able to see the API key and the API secret that have been generated. To generate the access token and the access token secret, click on the "*Create my access token*" button at the bottom of the page (marked in red in the figure below). Note that there will be a small green confirmation at the top of the page saying that the token has been generated.



Final result of the setup process

Finally, the page should look like the following. Notice the API Key, API Secret, Access Token and Access Token Secret. We are going to use these 4 keys in the next section.



6.3 Running the exercise

- Open the `/tweets-tcp.py` in the `dxlab-spark-master/python` folder and substitute the credentials specified there by your Twitter credentials.

Terminal 3. Prepare the tweets producer

- `$ docker-compose run tweets`

Terminal 4. Start receiving tweets in spark

- `$ docker-compose run pyspark`

Once inside `pyspark`, copy/paste the code in `python/spark-streaming.py`.

Note that your current solution monitors the `#Barcelona` hard coded in the **`tweets-tcp.py`** file. **You can modify it if you are willing to listen another #.**