HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Bachelor Thesis

Author: Miguel Santamaria

Supervisor: Ph.D. Nastaran Fatemi

Date: July 29, 2016

Acknowledgement

I would first like to thank my thesis supervisor, Ph.D. Nastaran Fatemi, who steered me in the right the direction whenever I needed it and for providing me with all the necessary facilities for the research.

I would also like to acknowledge Jaspreet Sandhu as the second reader of this thesis, and I am gratefully indebted to her for her very valuable comments on this paper.

I also place on record, my sense of gratitude to one and all, who directly or indirectly, have lent their hand in this venture.

Abstract

The purpose of this project is to enable analysis of varied subjects using the social platform of *Twitter*, static or in real-time, using an application that provides an expanded geographic filtering of tweets and generates maps and charts on the analysis, *GeoTwit*. This paper thus contains the analysis, development and implementation of this web tool, as well as numerous and varied subjects that have already been analyzed over the duration of a semester, at the level of individual topics as well as current events.
In preparation for the development of this tool, market studies were realized in addition to a thorough analysis of different Twitter's APIs, in order to glean the most optimal approach for the project. Furthermore, several prototype applications were developed and tested in advance, to understand and resolve the potential roadblocks and issues that could and did arise in the course of development. Finally, various schemas and diagrams like the mock-up and UML schemas were prepared, in order to visualize and discretize the steps involved. The final implementation integrates and streamlines all these tools to allow the end user to analyze a maximum of two simultaneous subjects in real-time. Static analyses are also possible, but are not as expanded as the real-time functionality. Requiring the user to connect to his Twitter account, the application allows him/her to import and export their analyses (maps, charts, speed reception and various other useful information) as external files.

Contents

Chapter 1

Introduction

1.1 Summary of the Problem

Within the span of a few years, social networks have rapidly mushroomed across the globe to become all-encompassing and omnipresent in our lives. Across such networks, a massive quantity of data is shared at every moment in time, and each time a larger volume from the preceding day. In addition to a widespread use by lambda people, the scientific community has been finding novel ways to analyze and make sense of this cascade of available social information: *data mining*. Indeed, what better source there is than a social network, continuously and increasingly reporting collective human opinion and behavior in real-time, to analyze human-oriented subjects?

The goal of this project is thus to set up a web application that allows users to enter one or two subjects of their choice, to visualize a real-time activity graph/map for these subjects both on Twitter and on a geographic map and finally to import and export the results as external files. Static analyses are also possible, but are not as expanded as the real-time functionality.

In case the reader would not well know Twitter[1], or as a reminder, this social networking service enables to send and read short 140-character messages called "tweets". Tweets can be posted in a public way or in a private one (in this case, only the users that follow the poster can access and read these data). If a user decided to enable the geolocation features in the parameters of his account, he can choose to either add or not his current location as a metadata for each posted tweet, identified by a geolocation tag. A word, phrase or topic that is mentioned at a greater rate than others is said to be a "trending topic".

[1] https://en.wikipedia.org/wiki/Twitter

1.2 Generalities

This project - *GeoTwit* - is carried out as the final Bachelor thesis within the TIC department of the Computer Engineering sector of the Yverdon-les-Bains's HEIG-VD school, for the IICT institute. The student - *Miguel Santamaria* - supported by his supervisor - *Ph.D. Nastaran Fatemi*, professor and teacher - worked alone on his project, at the level of both development and documentation. Note that this project is not confidential.

The project - selected in *December 2015* and started on *February 22, 2016* - covers several different aspects of web development as well as intelligent analysis (data mining) and data visualization. A progress report was asked for the *June 17, 2016*, the final thesis was delivered on *July 29, 2016*, and the oral defense is planned for the *September 5, 2016*. The basic duration of the project is *450 hours*, even if the work was done in less time than this value.

1.3 Specifications

Functionalities

The application provides the following features:

- The reading of keywords and the selection of geographic areas on the map by the user.
- The retrieval of tweets, using the Twitter's APIs. Note that only a certain percentage of these tweets include geographic information, necessary for the future operations; a first filtering is thus operated here.
- The analysis and the filtering of tweets via calculation of the number of tweets by areas and by subjects.
- The visualization of the results on maps.
- The interaction (zoom-in, zoom-out, etc.) with the maps. The development of this feature involved the use of appropriate algorithms (like tweet grouping) and libraries.
- The generation of data charts and the possibility to import/export one's analysis.

Used Technologies

The suggested tools for this project were the *Scala* language coupled with *Play Framework*, while noting other technological choices were also possible. Due to both the personal interests of the student and a certain curiosity for unknown technologies in general, it was decided that these choices will be retained. Lacking proficiency in either of these technologies, the student was initially required to learn the Scala language and the use of Play Framework before starting the development on the application. Note that the *JavaScript* web language is used to complement the client side of the application.

Chapter 2

Analyses

This chapter contains all the analyses done before the implementation of the application itself.

2.1 Existing Applications / Market Studies

There are several existing applications that allow a user to visualize tweets on a map, the most visible being:

- **One Million tweet Map**[1]: the One Million tweet Map application lays out, to some extent, what has been implemented in this project, i.e. displaying tweets in real time (filtered or not) on a geographical map with grouping methods.

  However, unlike GeoTwit, the application does not offer a static search and view of tweets nor the possibility to export and import the analyzed data and to access and analyze charts. In addition, it also does not offer the possibility to analyze and compare two subjects at the same time or to filter tweets by language or location of the tweet. Nevertheless, this application does not require users to connect with their Twitter's accounts, for using the application.

- **TrendsMap**[2]: this application shows a real-time mapping of Twitter trends across the world. If you click on any word you will see a real-time stream of relevant tweets. One must register and/or pay for further access to features/content.

  The interesting feature of this application is the display of current trends across the world, which is not currently present in GeoTwit. Users are also not required to connect to their Twitter accounts. However, users have to pay to access more detailed content, and there is no possibility to access static content nor to filter tweets with more complex filters or to access data analysis and visualization tools like charts. There is also not the possibility of importing and exporting one's analysis report.

[1] http://onemilliontweetmap.com/

[2] http://trendsmap.com/ - found via mashable.com/[3]

- **tweetping**[4]: shows real-time reception of tweets all around the world on a map.

  Since the application is still in a beta phase, it is not possible to access most features, but the application intends to offer data analysis tools like charts as well as a complex filtering process, nearly matching the GeoTwit's features. However as with the previous two apps, users do not seem to be able to access static features or import and export analysis. Users also need to pay in order to access more interesting functionalities, but don't have to connect with their Twitter accounts. A feature of note was that the GUI of this application is well designed and user friendly.

- **tweetMap**[5]: This tool allows one to visualize tweets written in a certain period, and create interesting charts and visualizations.

  This application provides interesting tools to analyze static content with a map (complex filters and charts), perhaps more so than GeoTwit. Users also do not need to connect. In this case as well, there was not the possibility to analyze real-time data or export analysis.

- **Stweet**[6]: his service combines Twitter and Google Street View data to provide us a location-based breakdown of trending topics in a specific neighborhood.

  This application, despite its simplicity, is innovative given the functionality of allowing the user to display tweets directly onto 3D street view.

- **GeoChirp**[7]: this app allows the user to search for tweets in a given location, but only in a static way (no real time updating).

  This application is a basic one, only returning tweets in a given location, with the upper limit specified by the user. Like GeoTwit, users must connect to their Twitter's accounts to use the application. However, there is no way to analyze real-time data or visualize the data.

- **tweet To Map**[8]: this web plug-in allows one to put markers on a map, based on tweeted topics.

  The idea of making a plug-in for use in other application is an innovative idea, which make this application more interesting than the others. However, the application only supports static content and solely displays markers on the geographical map. Subsequently, it has not been used by GeoTwit.

[4] https://tweetping.net/
[5] https://worldmap.harvard.edu/tweetmap/
[6] http://tweet.we-love-the.net/Stweet/ found via mashable.com/
[7] http://tweet.geochirp.com/
[8] http://tweettomap.com/

GeoTwit is thus not a standalone innovation, but can attest to some new and interesting features, markedly the possibility to collect tweets in both dynamic and static ways and the access to data analysis, visualization tools and import/export features. Here is a brief summary of GeoTwit's strengths and drawbacks:

- ✔ **Data export**: None of the analyzed applications seem to allow users to export the real-time analyzed data. GeoTwit allows the user to fully export the queried data and its analysis.
- ✔ **Charts**: Charts are a good way to easily analyze data, but is a functionality not often provided by applications. In fact, very few of the analyzed applications provided a charts tool.
- ✔ **More complex filters**: In most of the analyzed applications, users can only type a word or a phrase to filter tweets. In GeoTwit, users can filter tweets by complex keywords sets (a maximum of 2) with the possibility to add Boolean operators like "AND" and "OR" and compare results in real-time. It is further possible to filter tweets by location, language and dates (only for the static view).
- ✔ **Static and streaming search**: Other similar applications seem to offer either static or streaming analysis, but rarely both of them at the same time. This is one of the strong points of GeoTwit.
- ✔ **Free**: Lastly, GeoTwit is entirely free, unlike several other applications, including access to all of its features. In return, users have to connect to their Twitter's accounts and are limited to a certain number of simultaneous searches and static searches per period.
- ✖ **Mandatory connection to Twitter account**: Users need to connect with their Twitter accounts in order to use GeoTwit. As the application is free, this requirement arises out of Twitter imposed limitations.
- ✖ **Limitations**: Users cannot not follow more than two simultaneous streaming processes per account, and have an upper limit to possible static search per period of time. This disadvantage arises from the fact that GeoTwit has free and thereby a limited access to Twitter API.
- ✖ **Restricted Map display**: Unlike other applications that provide displays like Google Street View or other original views, GeoTwit only offers Roadmap view.
- ✖ **No functionality of current trends**: This was an often seen and useful functionality in the previously discussed applications, and can be useful to guide the users in their choice of filters. Unfortunately, there is currently no way to access current trends on GeoTwit.
- ✖ **GUI**: Despite the functional and reasonably user friendly GUI of GeoTwit, it can clearly be improved especially in regard to the previously analyzed applications, some of which clearly had enviable interfaces.

2.2 Technologies

This chapter contains a description of the main technologies used throughout this project. Note that many links have been provided for further reference, but are not necessary to get a reasonable understanding of the technology and its relevance to the project.

2.2.1 Scala

Scala is a multi-paradigm (object-oriented, imperative, concurrent and functional) programming language designed at the EPFL in 2003 by Martin Odersky. It has a strong and static typing discipline and runs on a JVM (Java Virtual Machine). As Scala heavily draws from Java, Java libraries may be used directly in Scala code and vice-versa.

Motivations

Given the generally high regard for Scala as a language and the student's penchant for functional (like Haskell or Java8) and object-oriented languages during his formation, Scala has been used in this project.

2.2.2 Play Framework

Play Framework is an open-source web framework created in 2007 by Guillaume Bort that allows the developer to write web applications with Java or Scala. With the decision to use Scala in this project, this choice follows logically, as the framework is compatible with the language, is well-documented and well regarded by the community.

In the following paragraphs, one can find a brief overview of the framework, in order to understand and follow what was done during the project.

Installation

Since the installation of the framework is quite simple, it will not be rewritten in this documentation, and refers the reader to its official documentation[9].

Creation of a new Project

To create a new Play's Scala project, the developer has to type the following command, where [APPLICATION_NAME] is the name of the application to create:

```
activator new [APPLICATION\_NAME] play-scala
```

Once done, one simply enters the new created folder and types "*activator*" to enter the Play console, where it is now possible to compile the code and start the server with a "*run*" command. Once the server has been started, the code will compile automatically on the loading of the webpage.

[9] https://tweet.playframework.com/documentation/2.5.x/Installing.

2.2.3 Twitter's APIs

Before starting, it must be noted that all content of this chapter comes from the documentation of the official Twitter development web site[5]. Subsequently, many diagrams contained in this document have been taken from this documentation.

Please also note that Twitter offers a "Best Practices" page, which have been helpful during the conception of the application, both for the security and optimization parts[10].

Twitter's Technologies

Twitter offers many tools to developers to facilitate the use of their platform within applications, which include:

- Fabric: mobile development tools to enable reliable interfacing with the social features of Twitter (sharing, connection, etc.).
- Twitter for Websites: widgets collection, used to streamline integration of Twitter widgets ("Post tweet" and "Follow" buttons, display of tweets, etc.) into a web site.
- **OAuth**: an authentication protocol used by Twitter APIs for securing and authorizing different executed requests. It should be noted that this is not a platform developed by Twitter but rather a free protocol. You can find more details about this protocol in the following chapter.
- **REST API**: These provide read and write access to Twitter data in JSON format or via OAuth in order to enable their use in development. They work like most of the APIs provided by services of this kind and are listed below.
- **Streaming API**: These APIs serve the purpose of sending a response to the requests of the REST APIs through a sustained HTTP connection. They can be useful to continually display tweets linked to a given subject and thus to automatically fetch new tweets matching with the filter. This technology is further analyzed below.

The various technologies listed above are the ones that can potentially be interesting within the project. After a quick analysis, it turns out that the tools that will be most useful and relevant are the REST API and the Streaming API, coupled with the OAuth protocol for security. For the first version, it has been planned to not use mobile technologies or the widgets option, which for the moment has been found not sufficiently wide and complete.

[10] https://dev.twitter.com/overview/general

OAuth

The current version (v1.1) of Twitter APIs uses version 1.0A of the OAuth protocol. The developer can choose from among two ways of authentication within their application:

1. **Application-User authentication**: this is the most common form of resource authentication for APIs of this kind (Instagram, Facebook, Twitter, etc.). Signed requests both identify the identity of the application and the end-user it makes API calls on behalf of. Specifically, the application user must have a Twitter account, be able to connect and then authorize the application to access data via his account (for actions requested by the developer: reading, posting tweets among other things) in order to get an access token. Later, all executed requests will be signed by this token linked to the current application and user. In order to use this method, the user must therefore have a Twitter account in addition to trusting the given application so as to grant it reading and/or writing access on Twitter's data.

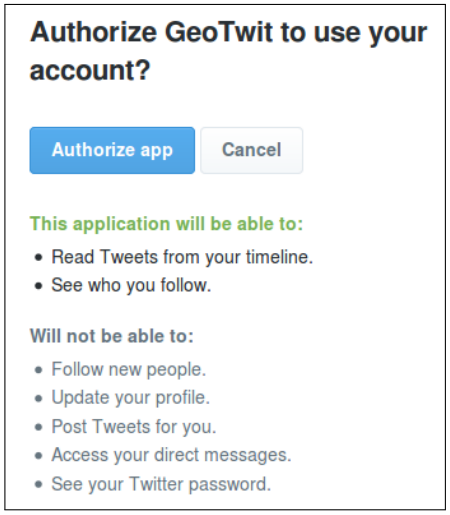


Figure 2.1: example of Twitter application's authorization

2. **Application authentication only**: the application executes the requests of the APIs on its own behalf irrespective of the user. This method could at the first sight seem promising but is unfortunately limited in its list and number of possible calls to the API. While it enables the user to search for tweets and get user information, one does not have access to geographic data or the possibility of streaming.

In spite of the ergonomic issues that the first method (application-user authentication) could give rise to, it is the better choice for our application. This is especially evident given that the second method does not allow the developer to use either geolocation or streaming, which are essential to the real-time visualization of Twitter's activities and are employed heavily in our application.

Introduction to REST and Streaming APIs

The REST API identifies the Twitter applications and its users by using OAuth and returns JSON-formatted responses. It allows one - among other things and according to the specific user's rights - to read and post tweets, access to Twitter user profiles, etc. The last released version (namely v1.1) works in the following manner:

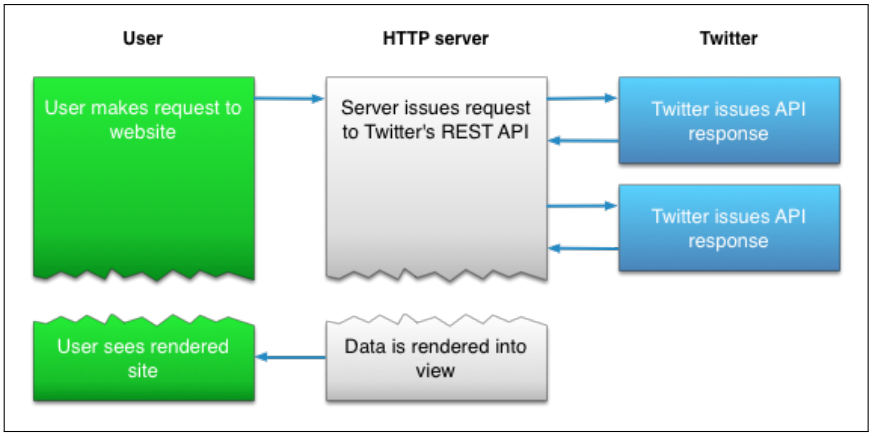


Figure 2.2: use-case of the REST API within a web application, taken from the documentation of the Twitter's APIs

In a nutshell, the user makes requests that are executed by the server, which then receive response from Twitter and forwards them to the client.

Streaming API's which dynamically update results of the search in real-time works a bit differently, but also uses OAuth and JSON:

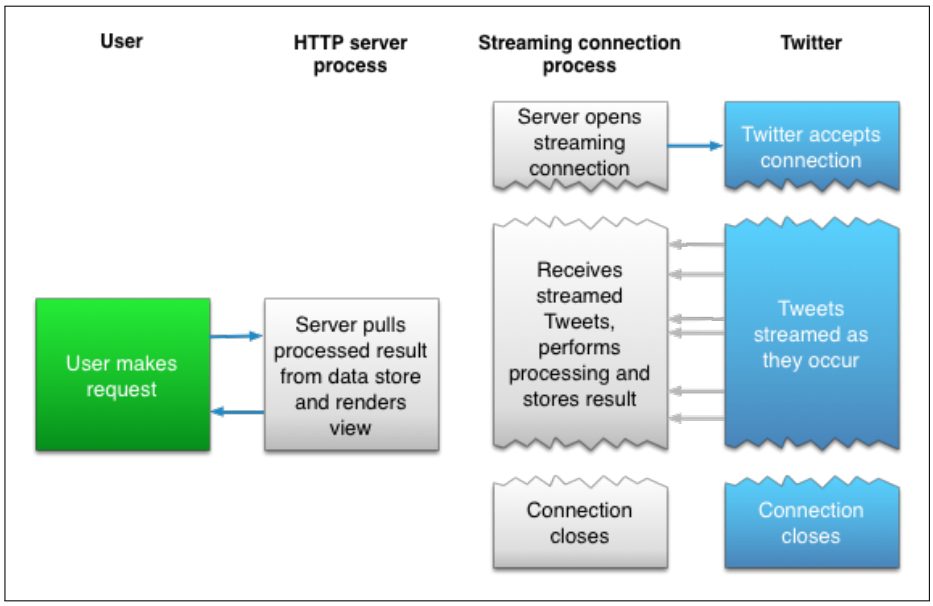


Figure 2.3: use-case of the Streaming API within a web application, also taken from the documentation of the Twitter's APIs

The operation is here a little more complex: the part of the code that maintains the streaming is executed in a separate process from the one that processes the HTTP requests.

There are 3 types of endpoints:

1. **Public streams**: streaming of public data flowing through Twitter, which can be filtered by keywords, users, etc.
2. **User streams**: single-user stream, roughly containing all of the data corresponding to a single user's twitter feed.
3. **Site streams**: a multi-user version of user streams.

For obvious reasons the public stream is the endpoint used within GeoTwit.

The Streaming API is the core functionality used in GeoTwit, owing to our desired end-goal, which is to be able to visualize the activities of the Twitter's subjects in real-time. The REST API is not set aside, though, as it will be useful for the comprehension of the streaming which bases its search algorithms on it. It will also enable users of GeoTwit to statically search tweets.

Streaming API

To initialize a stream based on a given filter, the developer must execute a specific HTTP request (as detailed above), indicating the Twitter's servers he wants to connect on. It is also necessary that the connection remain open such that the servers will send him data at regular intervals. To ensure a sustained connection, the Twitter's servers send a message to the application every 30 seconds, indicating that the connection is still active (in case there would be no new actualities). If no confirmation is received for a duration of 90 seconds, the developer needs to initiate a new connection to the stream.

Since there is no other way to test the Streaming API than to develop another application, there is no example in this chapter. It is advised however to refer to the "Prototype Applications" chapter, in which several prototype applications made to test both Twitter APIs and maps libraries have been introduced and explained. Search examples have also been presented below the "REST API" section of this document.

In the aforementioned prototype applications, it was noticed during testing that: Twitter seems to restrict the number of tweets transiting through streaming, which means that the Streaming API does not send all posted tweets. Indeed, when trying to post some relevant tweets when listening to a streaming (in particular during the Euro2016 event), the prototype application did not systematically receive them through the streaming (even if they had geolocation tags, etc.). Thus, the developer only receives limited samples (which still partially represent a view of reality) when listening to a streaming.

Limitations

In order to prevent the Twitter's servers to be overloaded by "churns" (a large number of requests of connections' opening and closing), Twitter restricts the possible number of connection requests[11] in a certain time frame. Once an application exceeds this number, it will receive an HTTP response containing the 420 error code; for unclear reasons, Twitter privately keeps the maximum number of possible connections as well as the time frame value. Note that if the limit is regularly exceeded, Twitter will block the IP address of the client for a while, making this an issue that must be taken into account.

[11] https://dev.twitter.com/streaming/overview/connecting

Please also note that the streaming is discouraged with mobiles using cellular network, because of the "*disconnection* → *connection*" loop's high risk to create "churns".

Finally, note that it is only possible to initiate two instances of the same application following a Twitter streaming with the same account at the same time.

Subscription to Twitter's subjects

As stated above, it is necessary to make an HTTP request to indicate to the servers of Twitter that the developer wants to subscribe to a particular stream. This request must/can have several parameters[12], which notably allow one to give various parameters to the server, like the desired language of the tweets, the filters (keywords, hashtags, etc.), the location, or other minor parameters.

For location, the Streaming API uses bounding boxes. In other words, it is a rectangular area defined by a pair of geographic coordinates (a longitude and latitude pair; be aware of the fact that we are more used to see coordinates in the "latitude, longitude" format), with the southwest corner of the bounding box given first, followed by the northeast coordinate. It is possible to have many bounding boxes in one request, separated by commas.

For example, if one wants to search tweets in Switzerland, one has to give the following coordinates: *5.956085,45.817851* and *10.489254,47.808454*.

Figure 2.4: graphical representation of the coordinates pair bounding Switzerland, taken from Google Maps

[12]https://dev.twitter.com/streaming/overview/request-parameters.

Beware however: the location does not act as a filter for the other filters, which means the request "*track=twitter&locations=5.956085,45.817851, 10.489254,47.808454*" will search tweets containing the "twitter" keyword OR being located in Switzerland.

To better understand the operation of the location "filter", several tests were conducted. It is important to note that those tests have been realized with the "twitter4jDesktop" prototype application (detailed on the "Prototype Applications" chapter below) and with the Streaming API. The used keyword was "michel" and the location was a rectangle bounding the U.S.A. (*[-124.411668, 24.957884], [-66.888435, 49.001895]*). This tuple of keyword/location was used to avoid retrieving too many results and to (unsuccessfully) isolate information.

1. **Search with keyword only** → the obtained results are correct and correspond to the keyword:

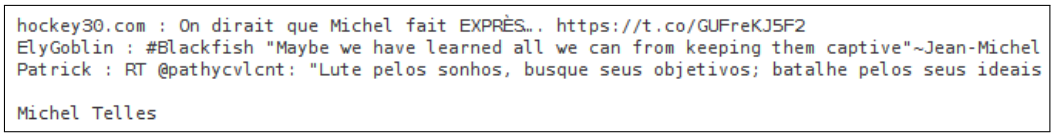


Figure 2.5: search of tweets with keyword only

   However, the tweets returned were from all around the world, which restrict the use cases within the application...

2. **Search with the location only** → anything and everything was obtained here: indeed, most of the returned tweets don't have geographic coordinates. This can be explained because of this "OR" operator applied on the search: the application is going to display all tweets coming from the location area OR the tweets corresponding to the keyword (namely an empty keyword in this case, so any tweet...). In summary, absolutely all tweets are displayed...

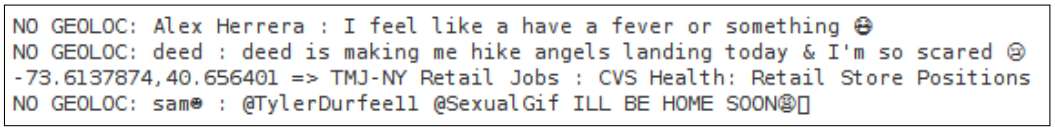


Figure 2.6: search of tweets with location only

3. **Search with keyword AND location** → at first sight one can expect this method to provide good results, but no... It is indeed a simple application of the operator OR: the application is going to display all tweets having the keyword "michel" OR written in the U.S.A. (without taking in account the keyword...), which will give incongruous results again.

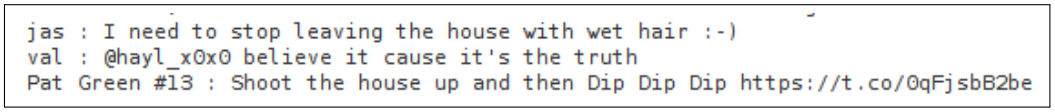


Figure 2.7: search of tweets with both keyword and location

The tweets don't necessary have a geolocation tag (a rate of about 20-30% of tweets having geographic data was observed during tests), so we decided filter the returned results on-the-fly to only accept those which have one.

None of these search methods gave appropriate results... This is a huge problem, because we need to search tweets by a keyword AND a location in the application. Therefore, we have no other option but to code our own location filter, which is applied to the results of a search by keywords.

Two different approaches are possible here:

1. Search tweets by keywords and (effectively OR) geolocation, and thereafter apply a manual filtering by keywords AND by geolocation for countering the logic filter OR. Here are the results obtained in two periods of 5 minutes with the "job" keyword and in the U.S.A. (by around 15:30, Swiss time zone):

   "I received 8061 tweets in 300 seconds, including 314 ones WITH geolocation tags and 284 ones with the wanted geolocation.
   This means 3.9% of the received tweets with the "job" tag(s) owned a geolocation tag and 3.52% contained the desired location.

   I received 6980 tweets in 300 seconds, including 225 ones WITH geolocation tags and 215 ones with the wanted geolocation.
   This means 3.22% of the received tweets with the "job" tag(s) owned a geolocation tag and 3.08% contained the desired location."

   Results here were disappointing with only about 3-4% of tweets having a geolocation tag.

2. Search tweets by keywords, and then apply a manual filtering of the geolocation.

   "I received 2290 tweets in 300 seconds, including 548 ones WITH geolocation tags and 516 ones with the wanted geolocation.
   This means 23.93% of the received tweets with the "job" tag(s) owned a geolocation tag and 22.53% contained the desired location.

   I received 2774 tweets in 300 seconds, including 571 ones WITH geolocation tags and 535 ones with the wanted geolocation.
   This means 20.58% of the received tweets with the "job" tag(s) owned a geolocation tag and 19.29% contained the desired location."

   Results are way more convincing here, with about 20% of tweets having the desired geolocation tag. Even if about 3 to 4 times less tweets were received than before, there was still more relevant tweets. Thus, this methodology was chosen for the project.

Within GeoTwit, it could also be possible to propose default locations (for example by countries). A prototype application ("Leaflet Countries' Borders" - see the "Prototype Applications" chapter) was made. In this application, you can double-click on countries to draw borders all around them. Many libraries/APIs exist and allow one to easily get coordinates by a pointed location for example. The most well-known of them undoubtedly is Google Maps, but other alternatives like OpenStreetMap exists and has been discussed below.

In addition to receiving JSON messages containing tweets' data[13], it is also possible to receive other kind of messages[14] from the servers of Twitter, for example to indicate a deletion of tweets or of geolocation tags, a disconnection, a limit reached, etc.

[13] formatted in the following way: https://dev.twitter.com/overview/api/tweets
[14] all messages' types are available right here: https://dev.twitter.com/streaming/overview/messages-types

Optimizations
In order to reduce the bandwidth, it is possible to request a compressed Gzip data format to the servers with the following HTTP header: "*Accept-Encoding: deflate, gzip*". Twitter also warns the developer about the fact that it is possible to receive a huge amount of data during certain sports or important cultural events. It is thus advisable to run initial tests in order to test if one's application is able to handle this massive flow and take precautionary measures.

Also, for optimization purposes, the servers use the "Chunked transfer encoding" data transfer mechanism, which essentially means sending data in a series of "chunks" without knowing the length of the content before starting transmission of a response to the receiver. The counterpart application thus must be compatible with this type of transfer.

It also must be noted that the received messages are not necessarily ordered and that a message can be received several times.

REST API

We layout some analysis on REST APIs below, which will still be useful in the future of the project, especially for the purposes of the static search functionality.

Limitations
First of all, it should be noted that this API has some limitations to ensure the security of the platform, and to avoid Twitter's servers to be overloaded by requests. Limitations are largely applied on the number of requests that can be made in periods of 15 minutes and by windows. Note that if one's application exceeds limits once too often it may be put on black list, which will consequently forbid the access to the API and thereby the possibility to make further requests. For this reason, the developer needs to carefully optimize the volume of sent requests.

When an application exceeds the rate limit of authorized requests, the Twitter's API send back a "429: Too Many Requests" HTTP response code, which allows one to indicate to the user a limit to the number of requests allowed by Twitter, has been reached, following his request.

The HTTP headers contain, among other things, information concerning the current rates of made and remaining requests:

- X-Rate-Limit-Limit: indicates the rate limit of requests for the type of the given request[15].
- X-Rate-Limit-Remaining: indicates the number of remaining requests for the type of the given request and for the current period of 15 minutes.
- X-Rate-Limit- Reset: the remaining time before the next reset of the 15 minutes' period.

[15] https://dev.twitter.com/rest/public/rate-limits

Tweet search
The search API belongs to the Twitter's REST API. It allows one to search tweets by various criteria[16] (latest or popular tweets containing or not keywords or hashtags, filtered by language and by geolocation, etc.). Given the massive quantity of data to process, for performance purposes, search only takes into account tweets published during the 7 last days preceding the request. After some tests, it was observed that some 9-days old tweets might be received in some cases, but no older.

Most of these parameters are also compatible with the Streaming API. Among them, the most interesting ones are the ones regarding the search of keywords and hashtags. In addition to these filters, there are several other parameters that allow further fine tuning of the returned query, such as the type of results (the latest and/or most popular tweets), the geolocation, the language of the tweet and even iterative or cumulative parameters (*count*, *until*, *since_id*, *max_id*).

Regarding the location, this API directly uses the geolocation. Specifically, it is strictly speaking not possible to search near a specific location: one has to use a specific geocode in the "latitude,longitude,radius" format, where *radius* is the search radius in miles (*mi*) or kilometers (*km*), having for center point the given latitude and longitude.

The API will initially attempt to search for tweets that match the requested latitude, longitude and radius constraints, and then in case of no results, will search for tweets whose users' profiles satisfy the desired geolocation parameters.

For example, if one wants to search for tweets in a radius of 50 kilometers around the main station of Yverdon-les-Bains, the parameters will have the following values: "*46.783177,6.640630,50km*". Ideally for this project, the user should be able to point to a location on a map and indicate the desired limiting radius, while the application automatically acquires the longitude and latitude values associated with the location.

Note that it is also possible to search for tweets by the ID of locations[17]. For example, the tweets emitted from the Twitter's headquarters can be found with the "place%3A247f43d441defc03" parameter. This parameter will certainly be less useful than the search with geolocation, but still deserves to be mentioned in this documentation.

[16] Many search filters exist https://dev.twitter.com/rest/public/search
[17] More information can be found here: https://dev.twitter.com/rest/public/search-by-place

Examples

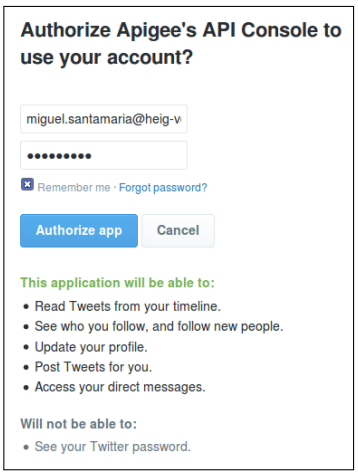


Figure 2.8: authorization of the API console through a Twitter account

To enable facilitated and speedy search requests, the advanced search tool[18] was used for the coming couple of examples. Whenever a search is made, the developer is redirected to a web page containing a URL, built using a particular format. For example, with a search about the @twitterapi user, one will get the following URL of redirection: *https://twitter.com/search?q=%40twitterapi*. Once done, the developer can get the valid URL to use in the API by replacing "*https://twitter.com/search*" by "*https://api.twitter.com/1.1/search/tweets.json*" (in this current example → *https://api.twitter.com/1.1/search/tweets.json?q=%40twitterapi*).

In addition, various tests have been conducted with the API console tool provided by Twitter[19]. Beforehand and in order to be able to make requests, one has to authorize the console to access one's Twitter account. This access is automatically requested as soon as one attempts the first request.

[18] https://twitter.com/search-advanced
[19] https://dev.twitter.com/rest/tools/console

Here is a list containing a few examples, which will turn out to be subsequently useful:

1. Search of the latest French-language tweets having ”yverdon” as keyword and (in reality OR, as mentioned previously) located in a radius of 10 kilometers from the Yverdon-les-Bains’ main station.

    - Request: https://api.twitter.com/1.1/search/tweets.json?q=yverdon&geocode=46.783177,6.640630,10km&lang=fr&result_type=recent
    - Results:

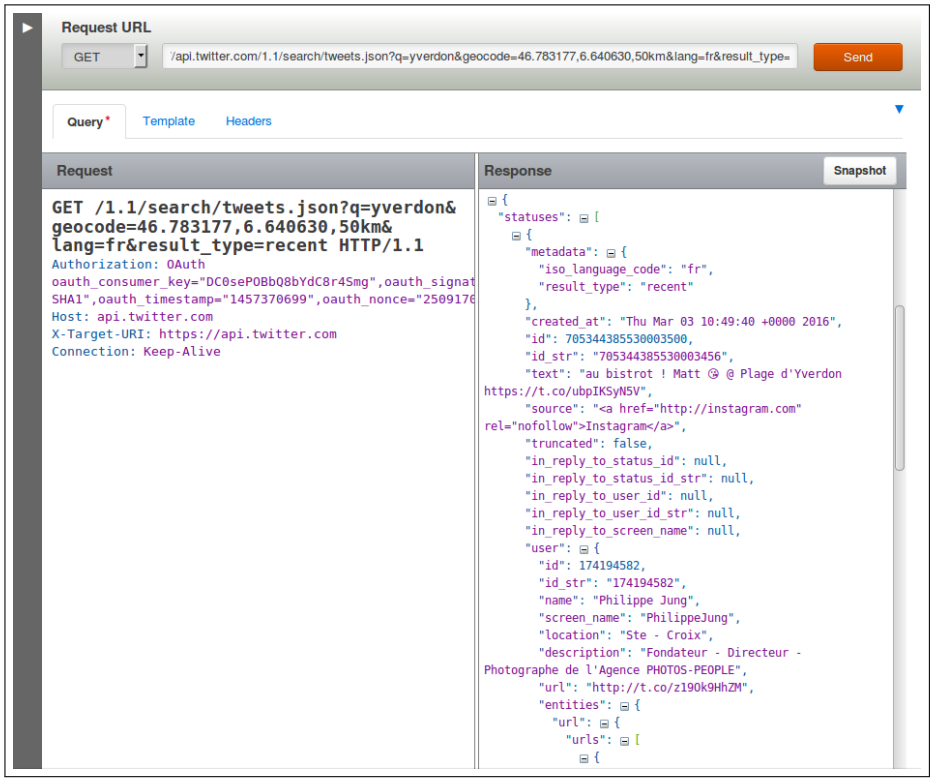


Figure 2.9: results of the latest French tweets containing the ”yverdon” keyword and (OR) being located near Yverdon-Les-Bains in the API console

    - Results on Twitter:

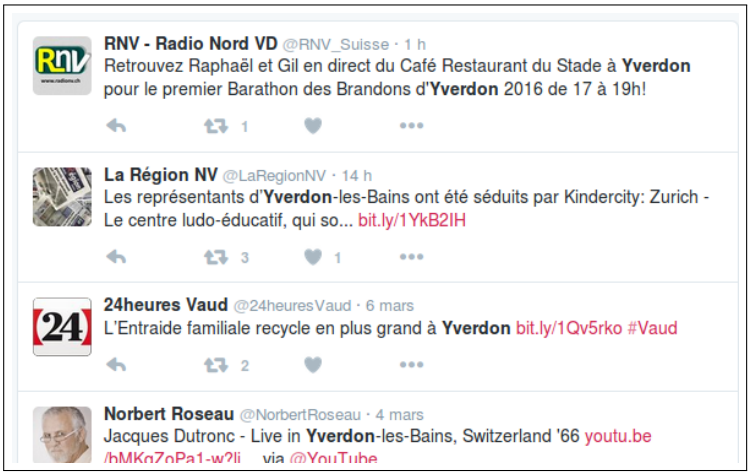


Figure 2.10: results on Twitter of the latest French tweets containing the ”yverdon” keyword and (OR) being located near Yverdon-Les-Bains

2. Search of popular and positive (containing a happy ":)" smiley) tweets, having the "#pizza" hashtag and (OR) being located in a radius of 100 kilometers from the center of Rome (without constraints on tweet language).

   - Request: https://api.twitter.com/1.1/search/tweets.json?q=%23pizza%20%3A%29&geocode=41.894143,12.495479,100km&src=popular
   - Results on Twitter:

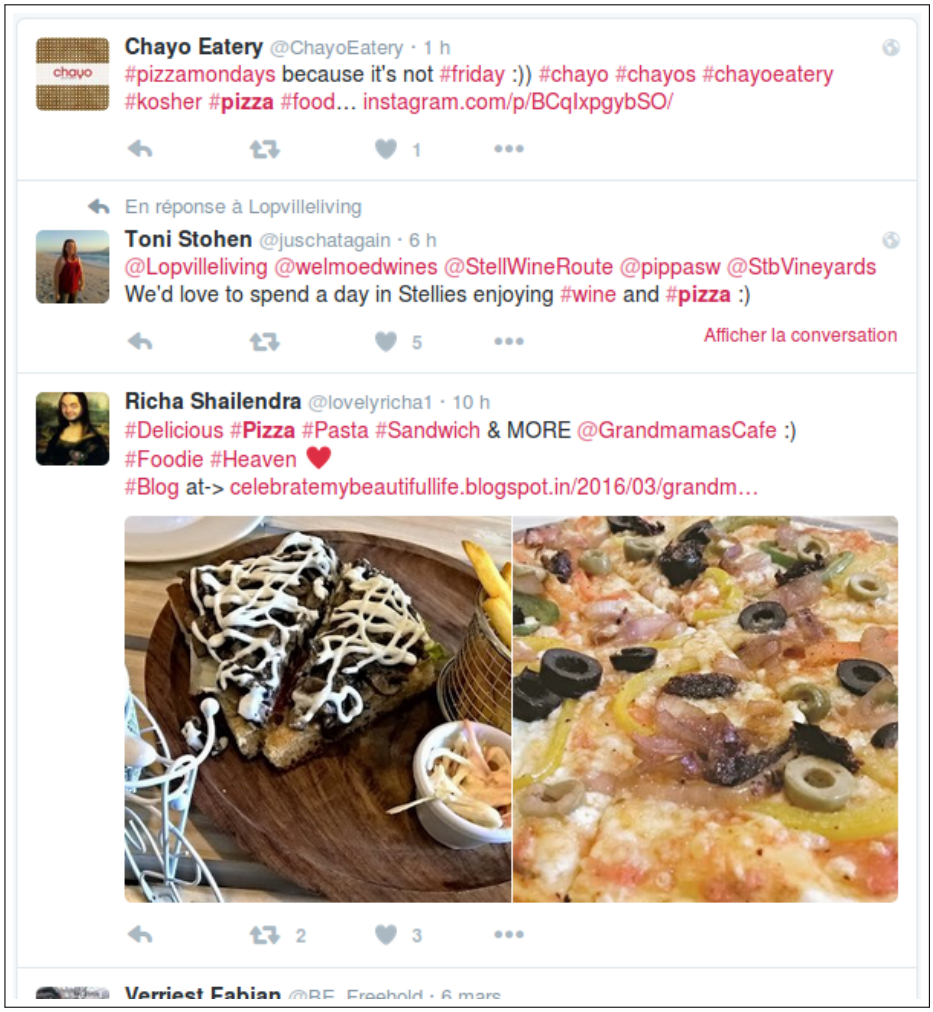


Figure 2.11: results on Twitter of the popular and positive tweets, having the "#pizza" hashtag and (OR) being located near to Rome

2.2.4 Interesting Twitter's Subjects

This section contains various Twitter's subjects, which were deemed interesting for the project. These tests have been conducted with the "twitter4jDesktop" application on 25 April 2016, between 13:00 and 16:30 (CET) for (tweets made in) Europe and the following day from 09:00 to 10:00 (CET) for U.S.A.

Subject	Geolocation (location, coordinates)	Nb. of new tweets in 5 min.
zurich	Switzerland, ([5.956085, 45.817851], [10.489254, 47.808454])	0
fondue		0
logitech		0
nestlé		1
swatch		0
job		1
sport		0
people		0
containing a 'e'		0
bbc	UK, ([-8.306947, 49.696022], [1.801128, 59.258967])	0
london		**15**
paris	France, ([-4.805145263671875, 42.34528267746347], [8.232879638671875, 51.09052797518529])	5
roma	Italy, ([6.6357421875, 47.09805038936004], [18.6328125, 36.577893995157474])	2
pizza		0
pizza	U.S.A., ([-124.411668, 24.957884}, {-66.888435, 49.001895}])	3
job		**618**
sport		0
people		4
vote		1
trump		3
clinton		0
logitech		0 or 1
nestlé		
novartis		
swatch		
mcdonald		0
java		0
scala		0

As for recurring themes for tweets made from Switzerland, the subjects can be separated into two main categories: the Swiss media and Industries/Companies based in Switzerland. Some of the unsuccessful tests included the following keywords: ”coop”, ”glencore”, ”logitech”, ”migros”, ”nestle”, ”novartis” and ”swatch”. These words not included in the table above, because of the insufficient refreshment rate (zero or one tweet per 5 minutes). Concerning the media, we tested some potential keywords that were deemed relevant for use across different parts of Switzerland (French, German and Italian parts) like ”job”, ”sport” or ”people”. These keywords were unfortunately not successful either. A further search for tweets containing the ’e’ letter (which is pretty common) in a last ditch attempt to find tweets with geolocation in Switzerland, resulted in zero results.

As for other major Europeans countries like UK, France or Italy, though most of the tested keywords were still unsuccessful, some of them gave better results, especially from major cities in each country.

We shifted our analysis to the U.S.A., given their affinity for social media usage, especially compared to Europe. During the tests, a holy grail was found: ”job”. More than 600 results in a 5 minutes’ period were received. More typical and currently relevant American subjects - like ”vote”, ”trump”, ”clinton”, ”mcdonald” - and computer subjects like ”java” or ”scala” were tried, but did not yield interesting results.

In conclusion, results are worse than expected. On an average, better results were observed in the U.S.A., especially for the ”job” keyword, which is the only keyword with great tweet frequency. Switzerland was quickly eliminated as a potential source of relevant tweets along with most other European countries, due to the zero rate of tweets having geolocation tags.

After some consideration and discussion with the supervisor, it has been decided that it would be better to try these keywords as well as other ones during hours/days instead of minutes. Those tests have been done once the application was developed, because it was more convenient to do long search and we could have access to graphs as well. You can find these tests in the ”Tests” part of this document.

Another interesting way to find subjects to analyze is to analyze the current trends in the world. Various web sites allow us to watch them, among which ”trend24”[20], was found especially useful to find an hourly listing of worldwide trends.

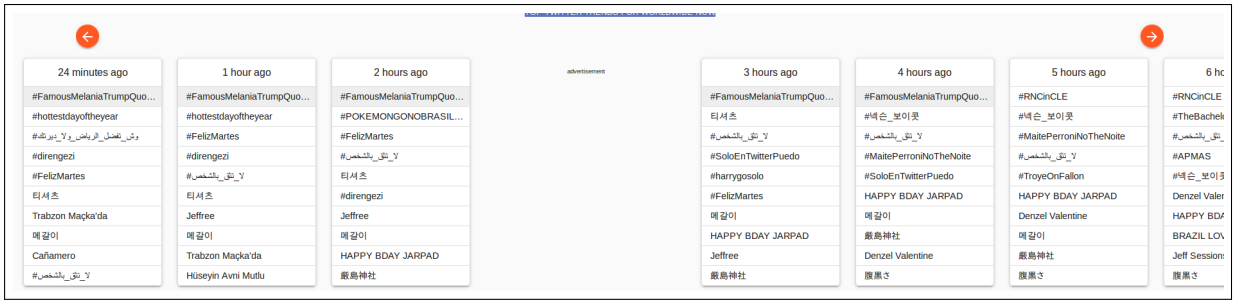


Figure 2.12: example of hot trends in the world within trend24

[20]http://trends24.in/

In addition to these examples, other interesting cases linked to ongoing events are possible: for example, a keyword corresponding to a subject of Swiss votes was analyzed, the name of a candidate in the U.S.A.'s elections during the results day (be careful about the massive flow of data!); the name of a show during its transmission on Television; the name of a sporting event's winner; etc.

Finally, here is a subset of hot and interesting topics that occurred during the semester:

Date	**Subject**
17.03.2016	Hockey: LHC - HC Ambri-Piotta
17.03.2016	St. Patrick
27.03.2016	Easter
21.04.2016	90 years of the Queen Elizabeth
23.04.2016	400 years of Shakespeare
23.04.2016 to 24.04.2006	20KM of Lausanne
08.05.2016	Football: FC Sion - BSC Young Boys
09.05.2016	Transit of Mercury in front of the Sun
11.05.2016 to 22.05.2016	69th Cannes Festival
14.05.2016	Eurovision in Stockholm
16.05.2016 to 05.06.2016	Tennis: Roland-Garros
29.05.2016	Football: Final of the "Coupe de Suisse"
05.06.2016	Swiss votes
10.06.2016 to 10.07.2016	UEFA Euro 2016
01.07.2016 to 16.07.2016	50th Montreux Jazz Festival
02.07.2016 to 24.07.2016	"Tour de France" 2016
19.07.2016 to 24.07.2016	Paléo Festival Nyon
20.07.2016	"Independence Day Resurgence" film release
22.07.2016	"Star Trek Beyond" film release
01.08.2016	National day of Switzerland
04.08.2016 to 14.08.2016	"Fêtes de Genève"
05.08.2016 to 21.08.2016	Olympic Games of Rio de Janeiro
07.09.2016 to 18.09.2016	Paralympic Games of Rio de Janeiro
25.09.2016	Swiss votes
29.10.2016 to 30.10.2016	Lausanne's marathon
08.11.2016	Elections in the U.S.A.

Euro 2016

It was decided to test the prototype application on Twitter during the Euro 2016's games. In order to achieve this, the "leafletAndTwitter4J" prototype application (see the well-named "Prototype Applications" chapter for more information) was used and markers that did not belong to the country being analyzed were manually removed (this could occur given the rectangular shape of the limiting coordinates). For all analyses, the "euro2016" keyword was tested:

- On **11.06.2016**, from **22:00** to **23:00**, during the 2nd half of the **England-Russia** match

Figure 2.13: tweets results for the UK-RU match

Score: **1-1**

47 tweets with geolocation tags

Average of **12'000-14'000** tweets per 10 minutes until the first goal (73'), then **20'000-24'000** after it and until the end

A massive flow of **16'000** tweets was received in a few minutes after the goal of the 92'.

Between **0.01%** and **0.07%** of tweets had a geolocation tag (between 2 and 20 tweets per 10 minutes).

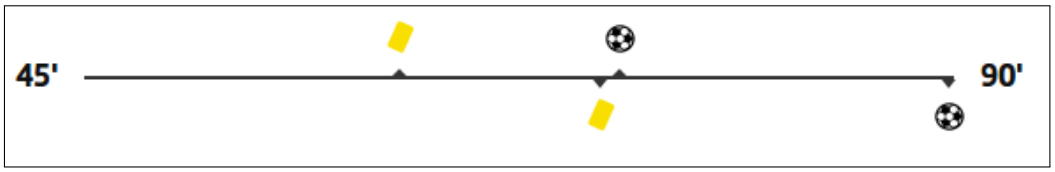


Figure 2.14: progression of the EN-RU match, taken from the RTS web site on the euro2016 section.

- On **15.06.2016**, from **17:55** to **19:55**, during the **Romania-Switzerland** match

Figure 2.15: tweets results for the RO-CH match

Score: **1-1**
5 tweets with geolocation tags
Average of **6'500-8'000** tweets per 10 minutes and **10'000-11'000** at the end of the match
Between **0.01%** and **0.05%** of tweets had a geolocation tag (between 1 and 4 tweets per 10 minutes).

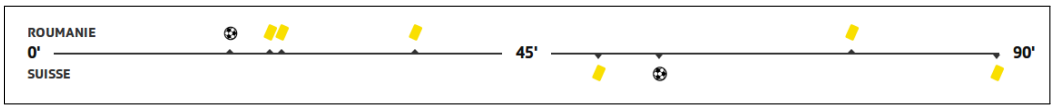


Figure 2.16: progression of the RO-CH match, taken from the RTS web site

- On **16.06.2016**, from **14:55** to **17:05**, during the **England-Wales** match

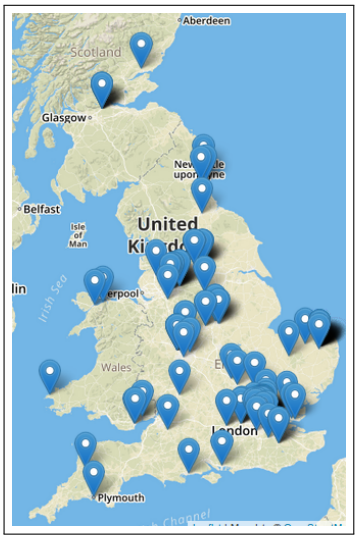


Figure 2.17: tweets results for the EN-WLS match

Score: **2-1** for England

135 tweets with geolocation tags

Average of **18'000** tweets / 10 minutes at the beginning of the match, then **10'000-14'000** until the end and finally **22'000-24'000** each time England scored (56' and 91').

Between **0.02%** and **0.14%** of tweets had a geolocation tag (between 3 and 20 tweets per 10 minutes).

There were several tweets from London, Norwich and Manchester, but just a few in Wales.

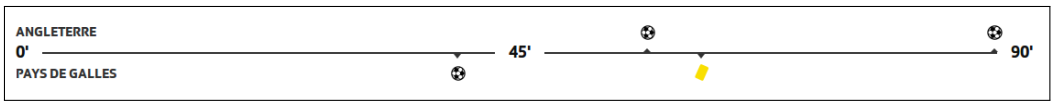


Figure 2.18: progression of the EN-WLS match, taken from the RTS web site

- On **15.06.2016**, from **20:55** to **23:10**, during the **Albania-France** match

Figure 2.19: tweets results for the AL-FR match

Score: **2-0** for France
126 tweets with geolocation tags
Average of **15'000** / 10 minutes at the beginning of the match, **9'000-12'000** during it, and **24'000-28'000** at the end when the France scored twice. Between **0.01%** and **0.09%** of tweets had a geolocation tag (between 5 and 15 tweets per 10 minutes).

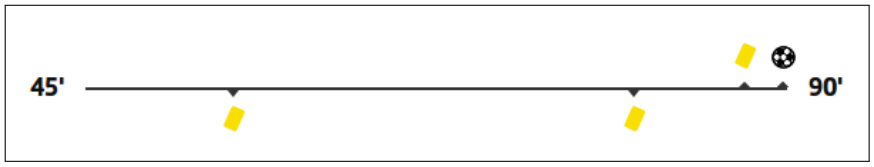


Figure 2.20: progression of the AL-FR match, taken from the RTS web site

There were plenty of tweets made from Lille, Paris and Marseille. Since the match took place in the "Stade Vélodrome" of Marseille, several tweets were posted there.

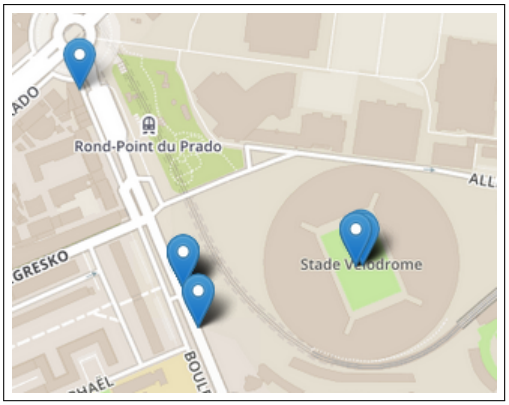


Figure 2.21: tweets results for the AL-FR match in the "Stade Vélodrome" of Marseille

- On **16.06.2016**, from **20:55** to **22:55**, during the **Germany-Poland** match

Figure 2.22: tweets results for the DE-PL match

Score: **0-0**

40 tweets with geolocation tags

Average of **18'000** tweets / 10 minutes at the beginning of the match, **11'000-14'000** until the end.

Between **0.01%** and **0.08%** of tweets had a geolocation tag (between 1 and 6 tweets per 10 minutes).

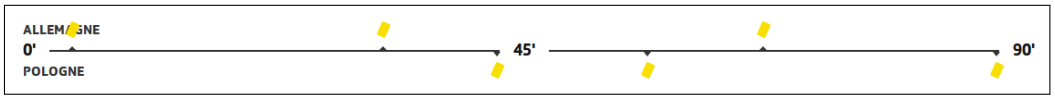


Figure 2.23: progression of the DE-PL match, taken from the RTS web site

- On **19.06.2016**, from **20:55** to **22:55**, during the **France-Switzerland** and **Albania-Romania** matches

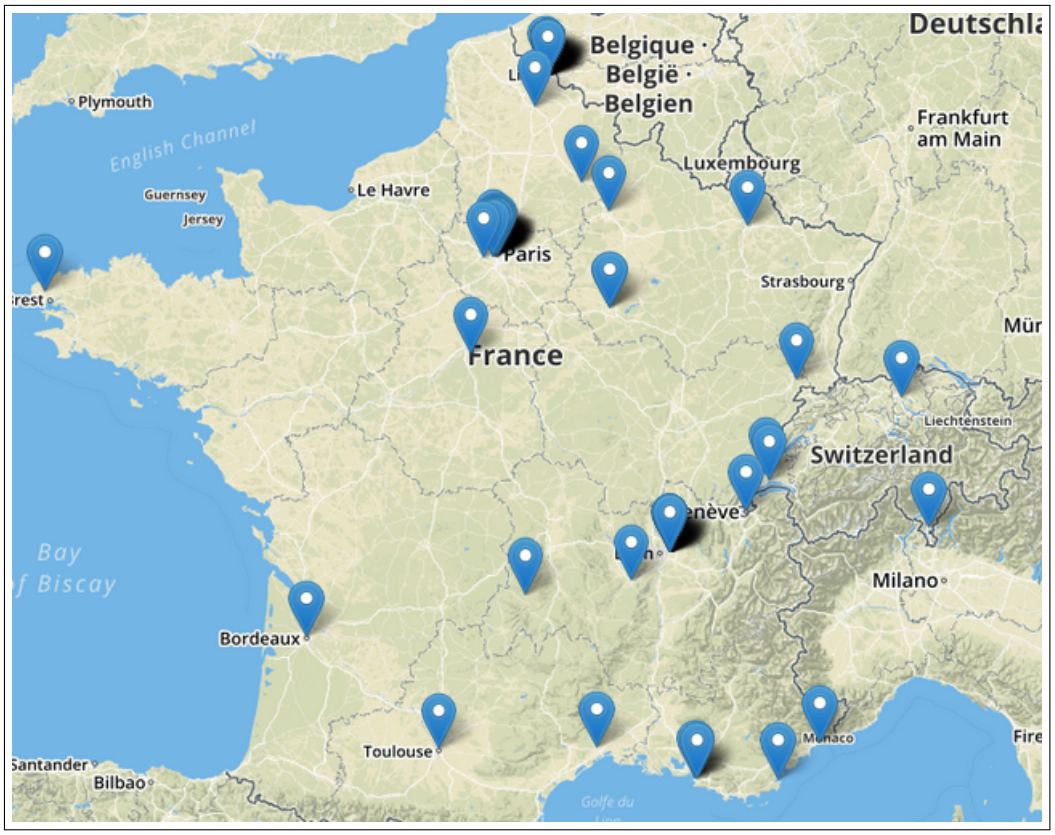


Figure 2.24: tweets results for the FR-CH match

Score (FR-CH): **0-0**
Score (AL-RO): **1-0** for Albania

90 tweets with geolocation tags (85 in France, 5 in Switzerland)

Average of **10'000-13'000** tweets per 10 minutes, and **20'800** at the end of the matches.

Between **0.03%** and **0.18%** of tweets had a geolocation tag (between 5 and 24 tweets per 10 minutes).

There were several tweets in Lyon (the city in where the AL-RO match took place), Paris and Lille (the city in where the FR-CH match took place).

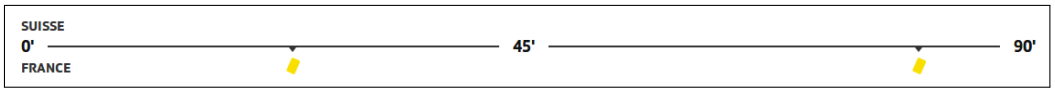


Figure 2.25: progression of the FR-CH match, taken from the RTS web site

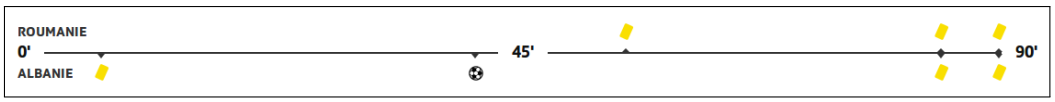


Figure 2.26: progression of the AL-RO match, taken from the RTS web site

- On **27.06.2016**, from **17:55** to **20:00**, during the **Italy-Spain** match

Figure 2.27: tweets results for the IT-ES match

Score: **2-0** for Italy

99 tweets with geolocation tags

Average of **15'000-18'000** tweets / 10 minutes at the beginning of the match, **24'000** during the half-time break, and **26'000** at the end.

Between **0.02%** and **0.19%** of tweets had a geolocation tag (between 4 and 26 tweets per 10 minutes). There were a lot of tweets posted in the North of Italy.

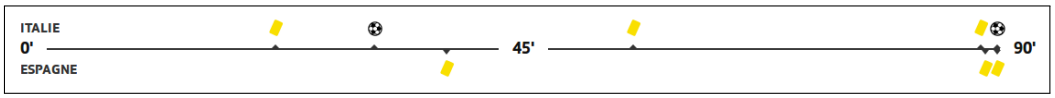


Figure 2.28: progression of the IT-ES match, taken from the RTS web site

- On **27.06.2016**, from **20:55** to **23:05**, during the **England-Iceland** match

Figure 2.29: tweets results for the EN-IS match

Score: **2-1** for Iceland

3 tweets with geolocation tags

Average of **22'000-26'000** tweets / 10 minutes during the whole match.

Between **0.02%** and **0.19%** of tweets had a geolocation tag (between 4 and 26 tweets per 10 minutes).

One tweet was received right after the first goal of Iceland, then two tweets in the same 10 minutes' period after the second one.

Figure 2.30: progression of the EN-IS match, taken from the RTS web site

- On **08.07.2016**, from **21:00** to **22:55**, during the **France-Germany** match

Figure 2.31: tweets results for the FR-DE match

Score: **2-0** for France

98 tweets with geolocation tags

Average of **18'000-21'000** tweets / 10 minutes during the match, and **26'000-27'000** for each France's goal

Between **0%** and **0.08%** of tweets had a geolocation tag (between 0 and 16 tweets per 10 minutes). There were a lot of tweets posted from the stadium of Marseilles (the city in where the match took place).

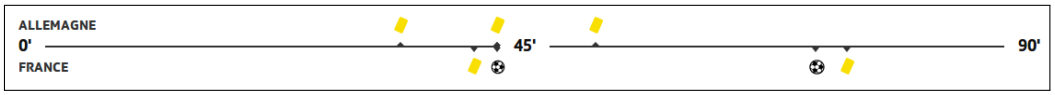


Figure 2.32: progression of the FR-DE match, taken from the RTS web site

In addition, the GeoTwit application was developed enough to realize the first analyses with it. Here are the results:

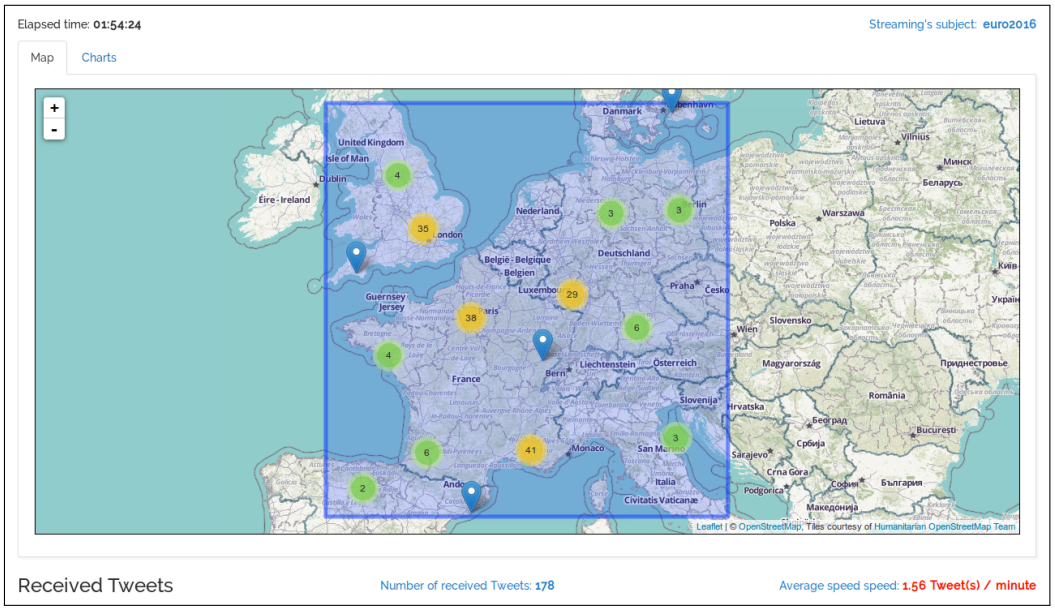


Figure 2.33: tweets results for the FR-DE match with the GeoTwit application

There were *about* **90** tweets with geolocation posted from France ("about", because of certain groupings that grouped tweets from several countries; logically, if we manually count them by zooming on the map, the number must be 98, like the prototype application), about **41** from Germany and a total of **178** in the selected area in about **01:54:24**, which gives an average speed of **1.56 tweets per minute**.

- On **10.07.2016**, from **21:00** to **23:35**, during the **France-Portugal** final match

Figure 2.34: tweets results for the FR-PT match

Score: **1-0** for Portugal

135 tweets with geolocation tags

Average of **27'000-28'000** tweets / 10 minutes during the whole match.

Between **0.01%** and **0.08%** of tweets had a geolocation tag (between 3 and 22 tweets per 10 minutes).

Figure 2.35: progression of the FR-PT match, taken from the RTS web site

This match was also totally analyzed with the GeoTwit application (see the "Final Match of the Euro2016 Event" chapter).

Discussions

After these results, we can conclude that England and France are the countries whose people posted the most tweets during the Euro2016 event, from among the analyzed countries. The France case can easily be explained: as this was the hosting country, several people were tweeting from the live matches. The English just seem to like Twitter and thus generally use it in high volumes; also, it is the country with the highest percentage level of tweets having geolocation tags. Concerning Switzerland, there were still too few tweets for it to be interesting: the application indeed received only 5 tweets from Switzerland in a 2 hours' period. Tweets are generally clustered around big cities and in stadiums during matches. We can obviously see a certain correlation between the goals moments and the peak of tweets. These analyses were much more rewarding compared to the previous analyses realized in Europe, because of the considerably higher tweet rate.

2.2.5 Twitter4J

Twitter4J[21] is a non-official library, which - as its name implies - allows one to easily use the features of Twitter's APIs with Java and which was proposed by the supervisor, Mrs. Fatemi. Given that Java classes are fully compatibles with Scala classes and vice-versa, it turns out that this library is easily usable with the Scala programming language.

It works both with the OAuth technology and the Gzip compression and is 100% compatible with the version 1.1 of the Twitter's API. The advantages of this library lies in the fact that it significantly simplifies the APIs uses (for both REST and Streaming APIs), at the level of both data reading and writing. The documentation of this library has examples for each use-case, as well as many useful explanations.

You can find brief relevant instructions about the use of this library in the following paragraphs, allowing you to better understanding what I did in order to develop the prototype applications, and finally the GeoTwit application.

Installation

If the developer is using a Java or Scala desktop application, he can download the latest stable version of Twitter4J, then add "lib/twitter4j-core-4.0.4.jar" to his application class path. If he wants to use other features (like streaming or asynchronicity), he has to add other libraries (like "lib/twitter4j-stream-4.0.4.jar" for the streaming), too. When using Play Framework, the developer just has to add **"org.twitter4j" % "twitter4j-core" % "4.0.4"** in the library dependencies in the */build.sbt* configuration file of his application.

Creation of a New Application

In order to use the Twitter's APIs on a new application the developer firstly has to create and register a new application on Twitter[22].

After this, he will be given access to a web page, which contains all the parameters of his application.

Once he has ensured that the access level of the application is the one he wants, he can easily get all keys and access tokens in the well-named tab, in order to use them in his application.

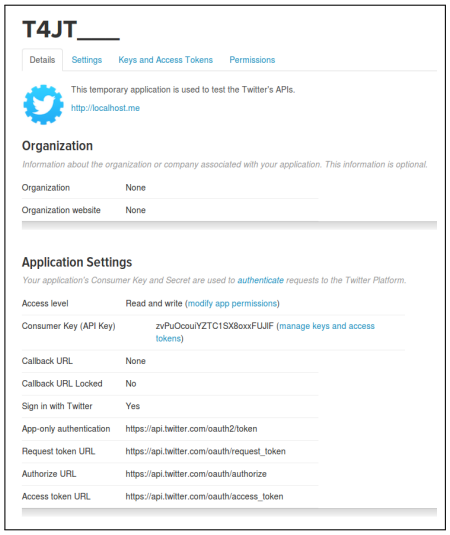


Figure 2.36: Twitter application's parameters

[21] http://twitter4j.org/en/index.html
[22] https://apps.twitter.com/app/new

Desktop Applications

Desktop applications are a little bit trickier to use with the Twitter's APIs, because the developer has to manually get access and must indicate to the API that he does not want it to redirect the user on a web page after the connection phase. You can find an example in the *GeoTwit/prototypes/twitter4jDesktop* folder.

For afore-mentioned reasons, the developer has to use the PIN-based OAuth[23][24] (the user has to give a PIN number generated by Twitter in order to use the API); here is the procedure used in the prototype applications:

1. Create a new app on the "Twitter for developers" web site (see "Creation of a New Application" chapter above).
2. Use the Twitter4J library to make a token request and get an access token[25].

   If done correctly, the user will then be asked to grant access to the application through a web page's link. Through clicking on this page, he will be redirected to the authorization page:

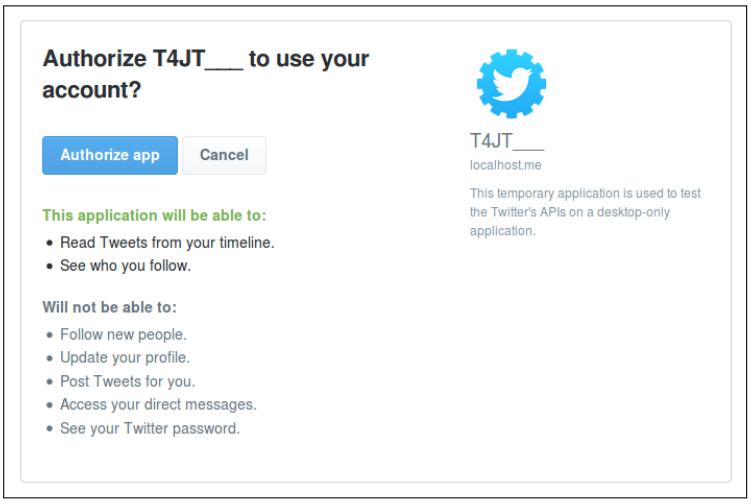


Figure 2.37: Twitter application's authorization page

3. Once the user has authorized the app, he will get the PIN to input into the application:

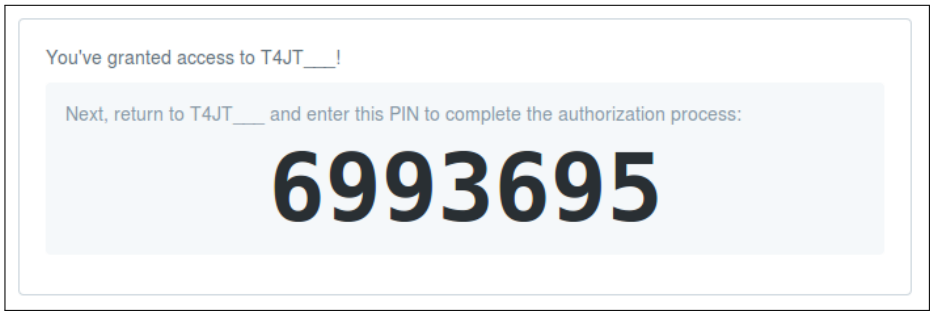


Figure 2.38: PIN code generated from Twitter

[23] https://dev.twitter.com/oauth/pin-based
[24] https://dev.twitter.com/web/sign-in/implementing
[25] http://twitter4j.org/en/code-examples.html, on the "OAuth support" section

Here is an example of a right output:

```
debug:
Open the following URL and grant access to your account:
https://api.twitter.com/oauth/authorize?oauth_token=v6inWQAAAAAAuPXSAAABU5no9GU
Enter the PIN(if aviailable) or just hit enter.[PIN]:6993695
Yay! Token successfully got: AccessToken{screenName='Migwelsh28', userId=533045969}
BUILD SUCCESSFUL (total time: 29 seconds)
```

Figure 2.39: output example of a Twitter's desktop application

4. On having received the token, you can simply use it for your next requests.

2.2.6 Geolocation Features

In order to make the web site as ergonomic as possible and thus to be able to display and work with geographic maps, a library is used. Two technologies were analyzed in this section: the all-pervading Google Maps, and one of its main open-sourced rival, OpenStreetMap.

Google Maps

The Google Maps' API[26] is very intuitive and user friendly (having already used it in the past), but can be problematic in its limitations: if one desires to use the site in a proper way, one needs to pay for the service in one way or another, in order to not be limited at the requests rate level, which is out of scope for a Bachelor thesis. For this reason, further analyses will not be drawn on this library.

OpenStreetMap

OpenStreetMap (currently in version 0.6) - proposed as an alternative by Mrs. Fatemi - is a direct and open-sourced competitor of Google Maps, and also boasts of an API[27]. In fact, there is two different APIs: the live-API[28], and another used for tests[29].

Although this technology is open source, this does not mean that it is lacking or slower: for example, it also uses the OAuth protocol and is used in a lot of applications, like Leaflet and MapQuest[30]. This technology gets especially relevant to this project when coupled to a JavaScript library - also open-sourced and very light - allowing to easily manipulate maps and add effects on them: Leaflet[31]. In our case, the interest in this library - in addition to the main features offered by the map - especially lies in the fact that it allows one to easily communicate with the OpenStreetMap's API. One can also easily draw circles (by a latitude, a longitude and a given radius), rectangles or custom polygons on the map (usable for geolocation features)[32] and add annotations as panels. Also, it is fully compatible with the mobile world, which is interesting for scalability issues.

Subsequently, we decided to use this platform within the project for all its geographic requirements.

[26] https://developers.google.com/maps/?hl=fr
[27] http://wiki.openstreetmap.org/wiki/API_v0.6
[28] http://api.openstreetmap.org/
[29] http://api06.dev.openstreetmap.org/
[30] a short list of these applications: http://wiki.openstreetmap.org/wiki/List_of_OSM-based_services
[31] http://leafletjs.com/index.html
[32] You can find an example here: http://leafletjs.com/examples/quick-start.html

2.2.7 Data Grouping

In course of the project, two major data issues were encountered:

1. A proper method had to be found when sending data from the GeoTwit's server to the client, because of the massive volume of data involved, which can potentially crash the web browser. After some analyses, it turned out that the web sockets technology is one of the best technologies to use in this case, because of the bidirectional connection between the client and the server, and especially given the fact that one can send floods of data through this route without problems other that some latency. This technology is detailed in the chapters related to the GeoTwit application (see the "GeoTwit - Technical Documentation" chapter) further in the document.

2. When displaying data on the map: since there can perhaps be thousands of tweets to display on the map, we have to group them with a grouping algorithm. Some existing solutions were analyzed, and one of the most interesting one was without a doubt the marker-cluster plug-in proposed by Leaflet again[33], which allows one to group data by circles on the map. This library provides a lot of interesting functions, is fully compatible with the Leaflet library (because Leaflet wouldn't develop libraries incompatible with their main library), and can handle up to 50'000 markers in one map[34] [35].

[33] https://github.com/Leaflet/Leaflet.markercluster
[34] http://leaflet.github.io/Leaflet.markercluster/example/marker-clustering-realworld.10000.html
[35] http://leaflet.github.io/Leaflet.markercluster/example/marker-clustering-realworld.50000.html

2.3 Prototype Applications

In order to test various libraries that were used within the project and as discussed before in this documentation, some prototype applications containing interesting tests were developed (in the *GeoTwit/prototypes* folder). Please note that there are no inputs controls and the GUIs are very poor, because these are only test applications.

2.3.1 Leaflet

This application contains a simple map drawn with **Mapbox**'s imagery and **OpenStreetMap**'s data all together through the **Leaflet** JavaScript library. In this basic application, you can pin markers and draw rectangles and circles on a map. The *js/map.js* file contains all the map's code.

Just open *index.html* in your web browser to initiate the application.

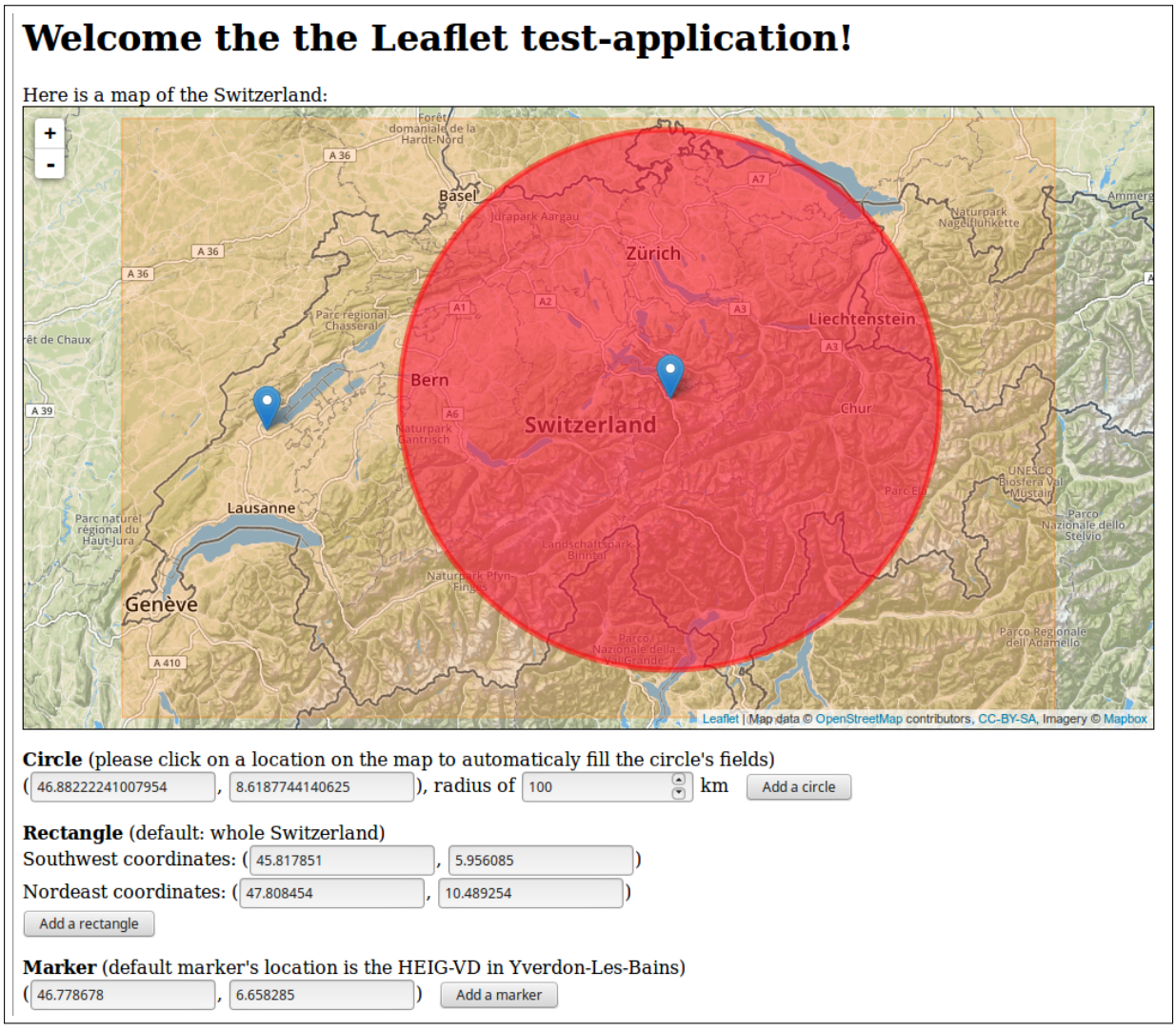


Figure 2.40: "Leaflet" prototype application

In this application, the Leaflet library uses both OpenStreetMap for the map's data and Mapbox for the map's imagery. The Leaflet library acts like a wrapper on the OpenStreetMap's API.

Mapbox[36] is a platform, which provides map's design as blocks to easily integrate location into web applications. Note that this platform has a limited requests rate, which was noticed too late. In GeoTwit, the OpenStreetMap imagery is directly used.

Before interacting with all the Leaflet's components, the developer is required to create an account on the Mapbox platform[37] and then he has to:

1. Get the default API's public token[38] and write it somewhere.
2. Create a MapBox project[39], which is used by the Leaflet library to provides imagery. Once done, the developer has to get the project's ID and write it somewhere.

Figure 2.41: a MapBox project

Finally, this simple web-application was created and the downloaded Leaflet library was extracted into it, followed by well detailed steps from the quick start tutorial. In order to call the map's layer with MapBox; the code looks like this:

```
L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="http://openstreetmap.org">OpenStreetMap</a> contribut
    maxZoom: 18,
    id: 'edri.pjdcfni6',
    accessToken: '[MY-PUBLIC-TOKEN-KEY]'
}).addTo(mymap);
```

Figure 2.42: MapBox integration into Leaflet

2.3.2 Leaflet - Countries' Borders

This application provides the same functionalities as **Leaflet**, with the possibility to double-click on a country to draw a polygon all around it. In order to make the application work, please do the following:

1. Ensure the JavaScript's packages manager "NPM"[40] in installed on your computer.
2. Run "npm install -g browserify" to install *browserify*, which is a library used to use npm's packages within an application without having a Node.js server[41].
3. "npm install" to install npm's packages.
4. Open the *index.html* file in your web browser.

[36] https://tweet.mapbox.com/
[37] https://tweet.mapbox.com/studio/signup/
[38] https://tweet.mapbox.com/studio/account/tokens/
[39] https://tweet.mapbox.com/studio/classic/projects
[40] https://tweet.npmjs.com/
[41] https://tweet.npmjs.com/package/browserify

5. Double-Click on any country on the map in order to draw a polygon all around it. Notice that some small countries are not present in the data set so you won't be able to draw a polygon around them.

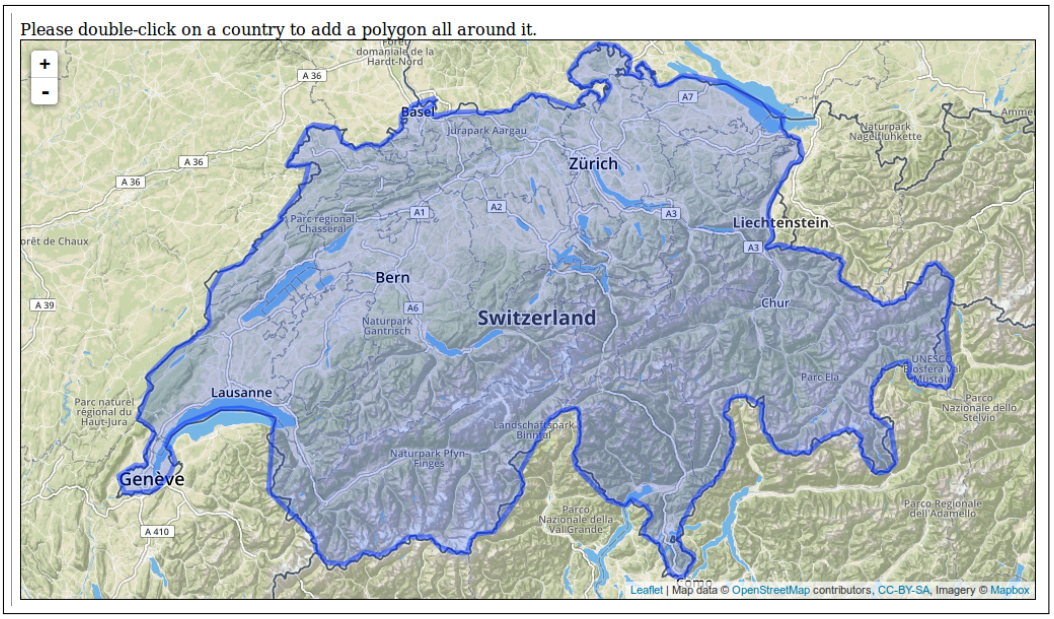


Figure 2.43: "LeafletCountriesBorders" prototype application

All implementation's details are well commented in the code, but here are the main things to know:

- The data of all the countries' borders are contained in the */data* folder in a binary format (Shapefile or ".shp"[42]).
- Since the content of this file is in binary, JavaScript cannot read it directly and thus has to parse it and convert it to JSON (specifically to GeoJson[43], which is an agreed format used for encoding a variety of geographic data structures in JSON).

  In order to do this, the *calvinmetcalf*'s shapefile-js library[44] was used, which was randomly found on Google and which perfectly works by writing "shp('data/TM_WORLD_BORDERS-0.3').then(function(geojson) ...".
- Two npm's packages were also used to respectively get a country's ISO (e.g. CH or FR) by a longitude and latitude pair and to get a country's name by its ISO (e.g. CH => Switzerland): which-country[45] and world-countries[46].

  Since these packages are npm packages (which must normally be used in a Node.js server), the amazing "browserify" npm package is used. This one allows the developer to use npm's packages within a simple JavaScript application.

  As soon as you updated the */js/map.js* file (which manages the double-click event on the map), you have to type "browserify map.js -o bundle.js" in a console to generate the */js/bundle.js* file that contains a working version of the map.js with all the code of the npm's packages included.

[42] https://en.wikipedia.org/wiki/Shapefile
[43] http://geojson.org/
[44] https://github.com/calvinmetcalf/shapefile-js
[45] https://tweet.npmjs.com/package/which-country
[46] https://tweet.npmjs.com/package/world-countries

- When the user double-clicks on a map, the application first gets the clicked point's coordinates, then search for the clicked country's ISO and name with the coordinates, and finally get the borders' coordinates in the GeoJson data by the country's name in order to draw the polygon with Leaflet.

2.3.3 Twitter4JDesktop

This is a simple Java application in which you can search for tweets with the REST API or subscribe to the Twitter's Streaming API (by default). In order to use this application please do the following: uncomment the country in which you want to search tweets in the "main" of the *Twitter4j.java*, then compile the application, launch *Twitter4j.java* and follow the output instructions.

```
run:
Trying to get the persisted token...
Persisted token successfully got!
Please enter some tags to search: job
How long do you want the streaming to run (in seconds)? 10
[Sun Jun 12 17:09:52 CEST 2016]Establishing connection.
[Sun Jun 12 17:09:53 CEST 2016]Connection established.
[Sun Jun 12 17:09:53 CEST 2016]Receiving status stream.
I received 79 Tweets in 10 seconds, including 14 ones WITH geolocalion tags and 13 ones with the wanted geolocation.
This means 17.72 % of the received Tweets with the "job" tag(s) owned a geolocation tag and 16.46 % contained the desired location.
```

Figure 2.44: "Twitter4JDesktop" prototype application

Note that you may have to provide a PIN for the application if this is the first time you opened it on your computer (see the above "Twitter4J" chapter).

All the Twitter4J documentation was used in order to develop this application and everything is well-commented so no other explanations were put here.

2.3.4 Leaflet and Twitter4J

This application receives streams of tweets and displays them on a map.

It contains two applications: **twitter4JWeb**, which is a Java application containing the **twitter4jDesktop** code and a web sockets server used to communicate with the second application **leafletAndTwitter**. This application is a JavaScript application displaying the received tweets from the web socket server in a map. The well-named "js/websocket.js" file contains the web socket client part.

Is order to correctly use them, please do the following:

1. Uncomment the country in which you want to search tweets in the beginning of the "readStreaming" method in the *Streaming.java* file of the "twitter4jWeb" application. You can also change the duration of the analysis below.

2. Include the keyword you want to search at the end of the "initializeConfiguration" method in the *Streaming.java* file. Try with the "job" keyword in the U.S.A. to get good results.

3. Check that NetBeans and the GlassFish server are properly configured on your computer.

4. Run the Java server "twitter4jWeb" in order to deploy and run the GlassFish server.

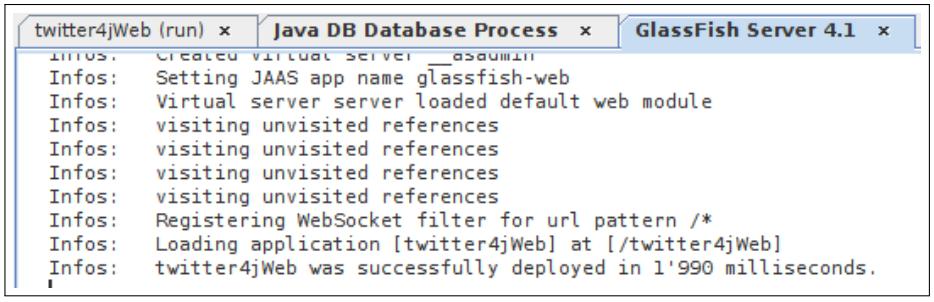


Figure 2.45: deployment of the GlassFish server in the "twitter4JWeb" prototype application

5. Open the *leafletAndTwitter/index.html* file in your web browser when the server is correctly started.

6. Then click on the "Start Streaming" button in the web application to start the streaming.

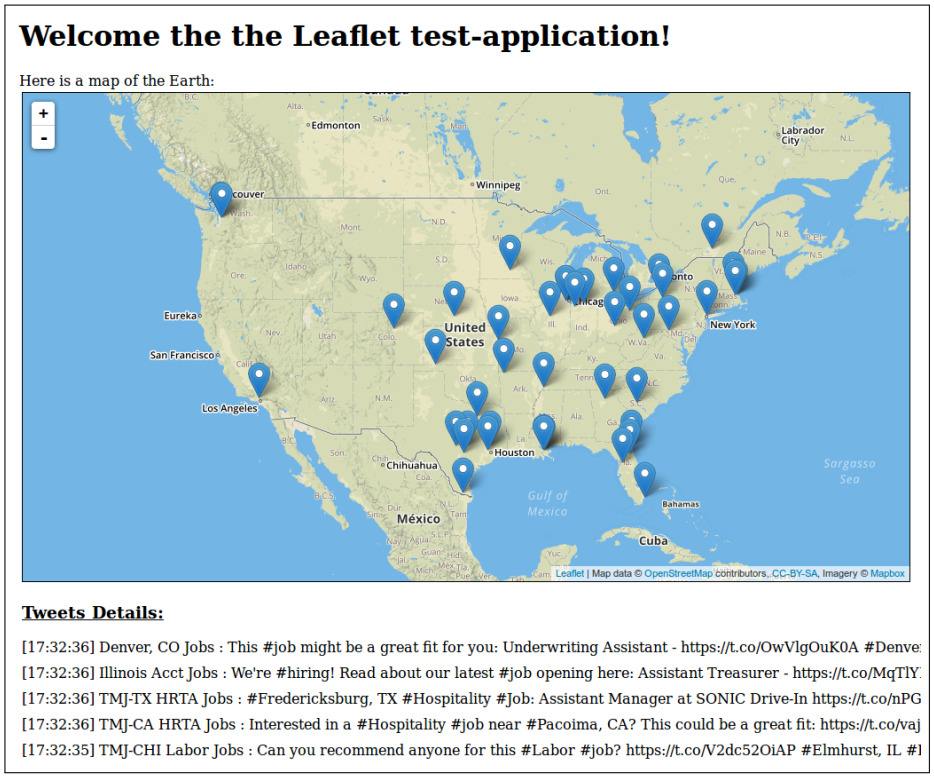


Figure 2.46: "LeafletAndTwitter" prototype application, using the "job" keyword in the U.S.A.

Like before, the Twitter4J documentation was used in order to develop this application and everything is well-commented so no further explanations deemed necessary.

2.3.5 Discussions

These applications were used to properly test all libraries used within the project, especially the Leaflet and the Twitter4J libraries. By developing them, a substantial part of the project's work was accomplished in addition to allowing us to better understand and document these libraries.

Note that the **"Leaflet - Countries' Borders"** application will perhaps be put on a project apart on GitHub, because some people could need to use it.

Chapter 3

GeoTwit - Technical Documentation

This chapter contains all documentation related to the *GeoTwit* application's code.
See the "*Instruction Manual*" appendix in order to properly install the application.

3.1 Mock-Up

The following mock-up was produced with the open-sourced *Evolus Pencil* application. In order to make the mock-up more comprehensible, it was separated into distinct sections; in order to have a global view, you can find the whole mock-up schema in the *doc/analysis/GUI* file. When the user first accesses the web site, he is redirected to the home page, where he is prompted to log into his Twitter account.

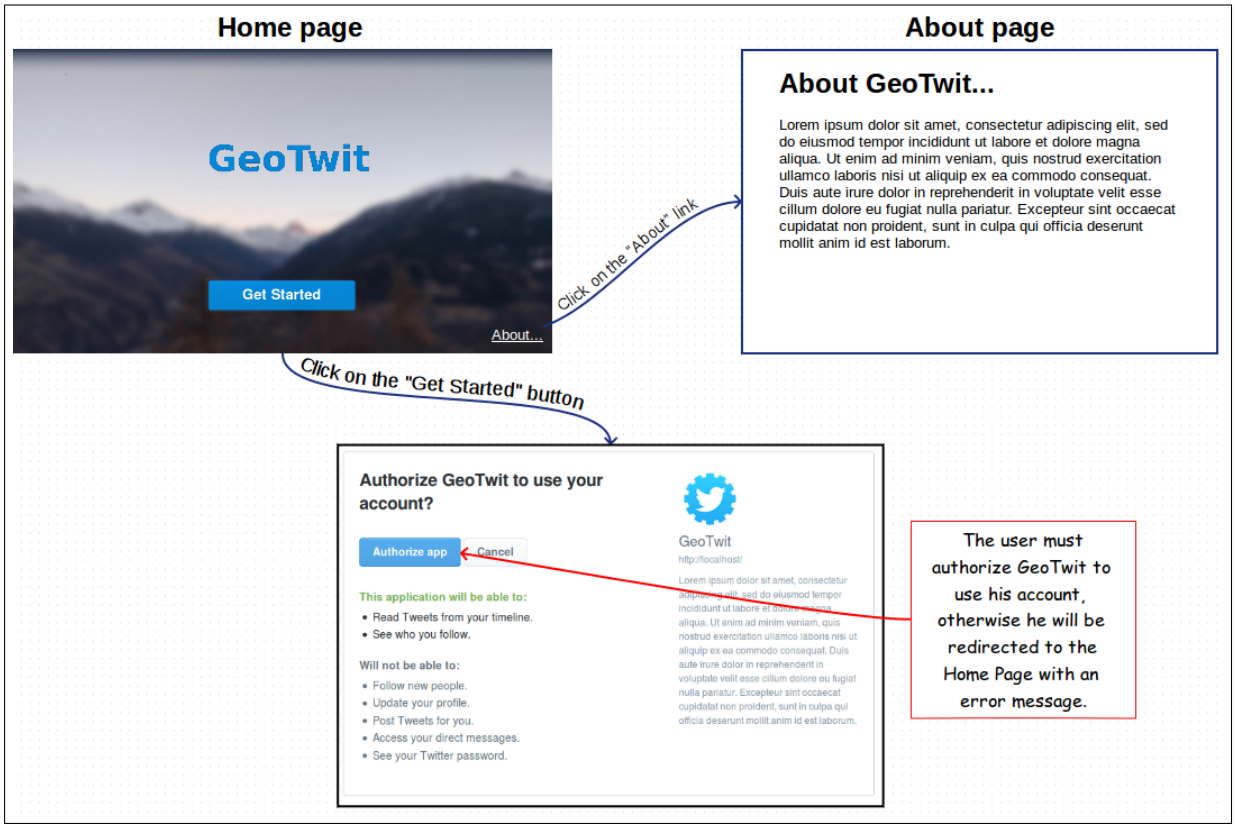


Figure 3.1: mock-up - connection process

Once the connection is established, the user is redirected to the search page where he can search for tweet keywords in dynamic or static mode. First have a look at the **dynamic mode** (sometimes also called "streaming mode" in the application), which streams from Twitter's API.

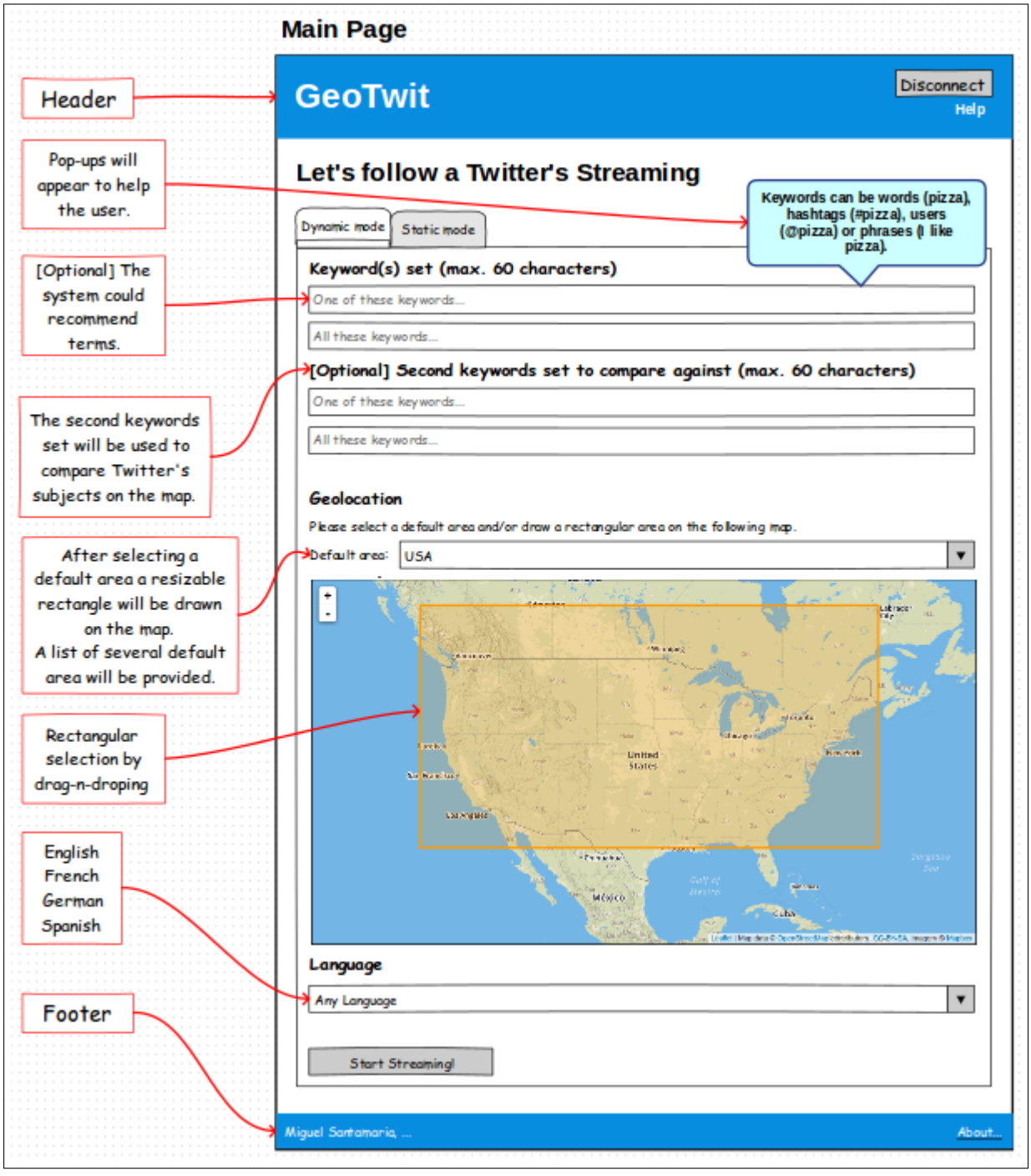


Figure 3.2: mock-up - main search page (dynamic mode)

The user fills in the fields, and clicks on ”Start Streaming!” to begin the streaming process.

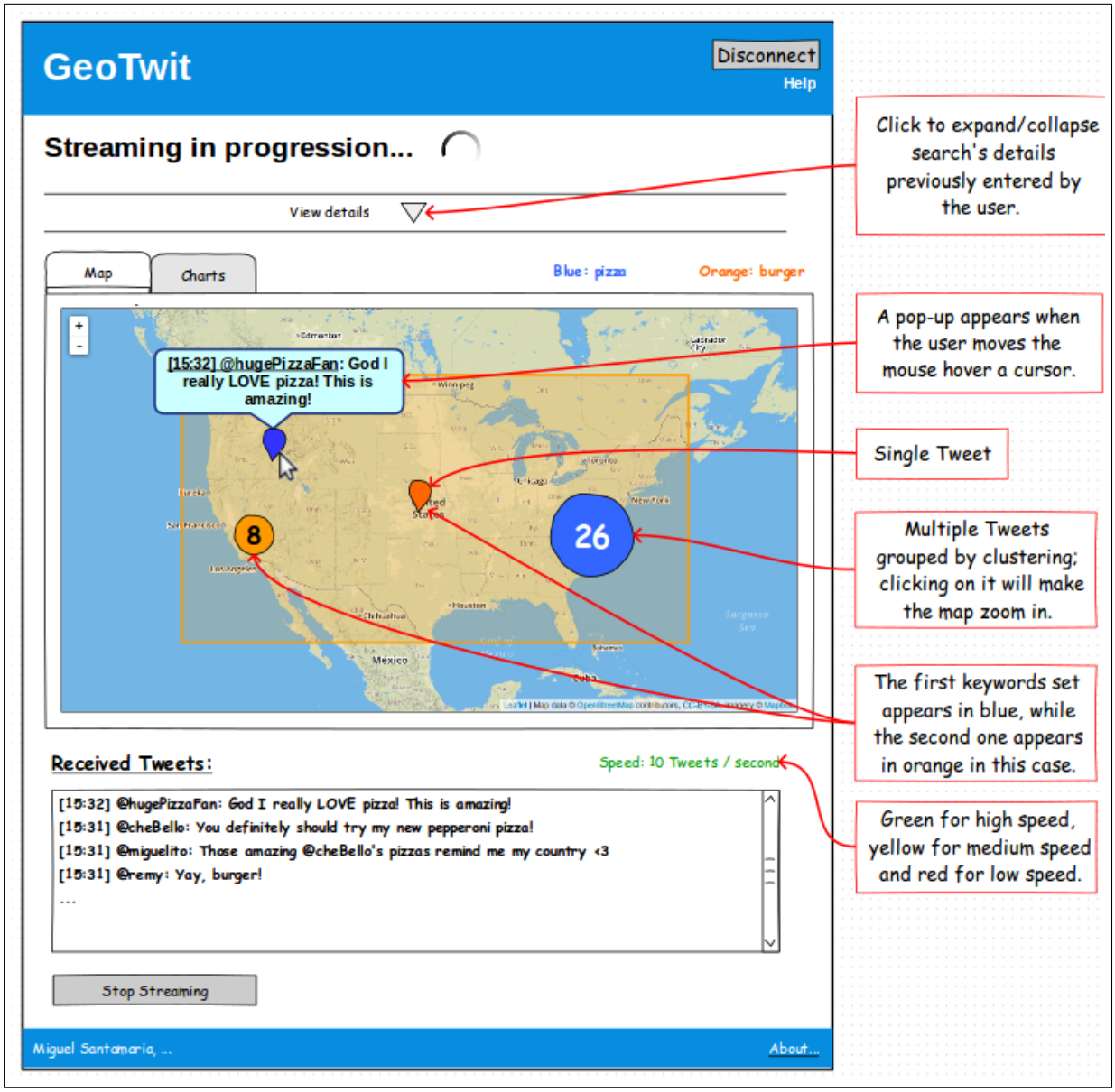


Figure 3.3: mock-up - streaming process

When observing the results, the user has access the "Charts" tab, which can provide interesting charts on the current streaming data.

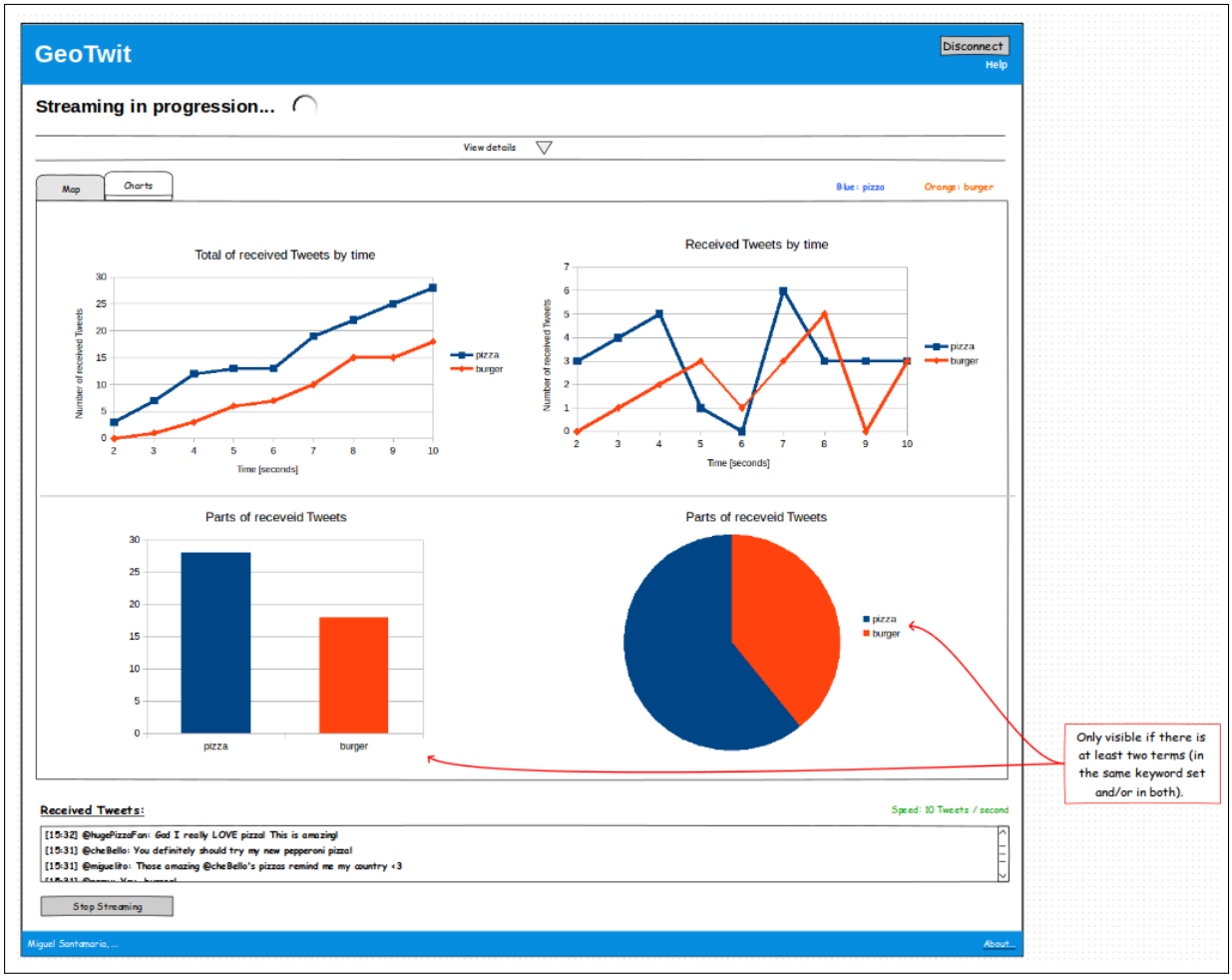


Figure 3.4: mock-up - charts of a streaming process

And now let's have a look at the **static mode**, which uses Twitter's REST API.

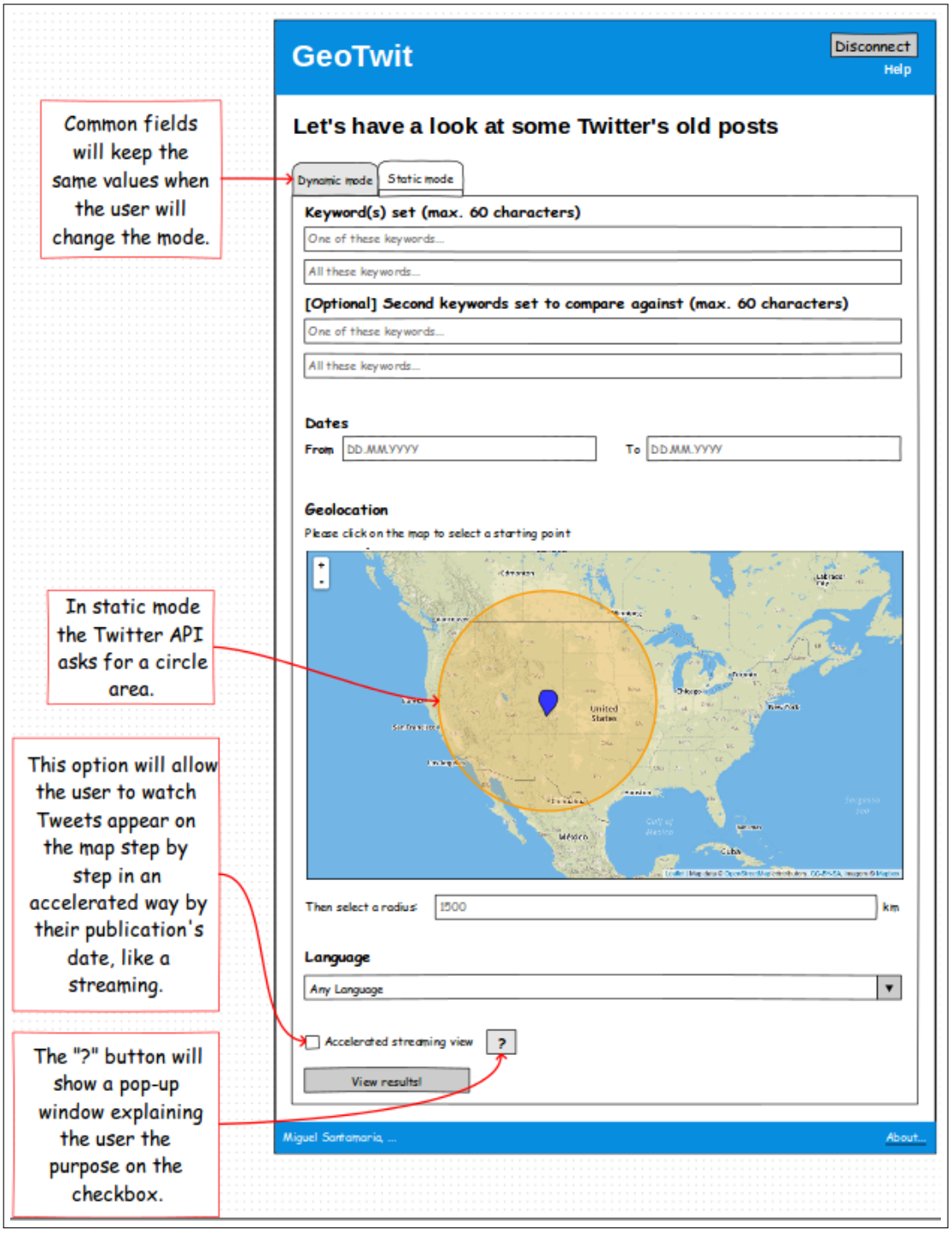


Figure 3.5: mock-up - main search page (static mode)

Here again the user must fill in the fields and click on the "View results!" button in order to access the results.

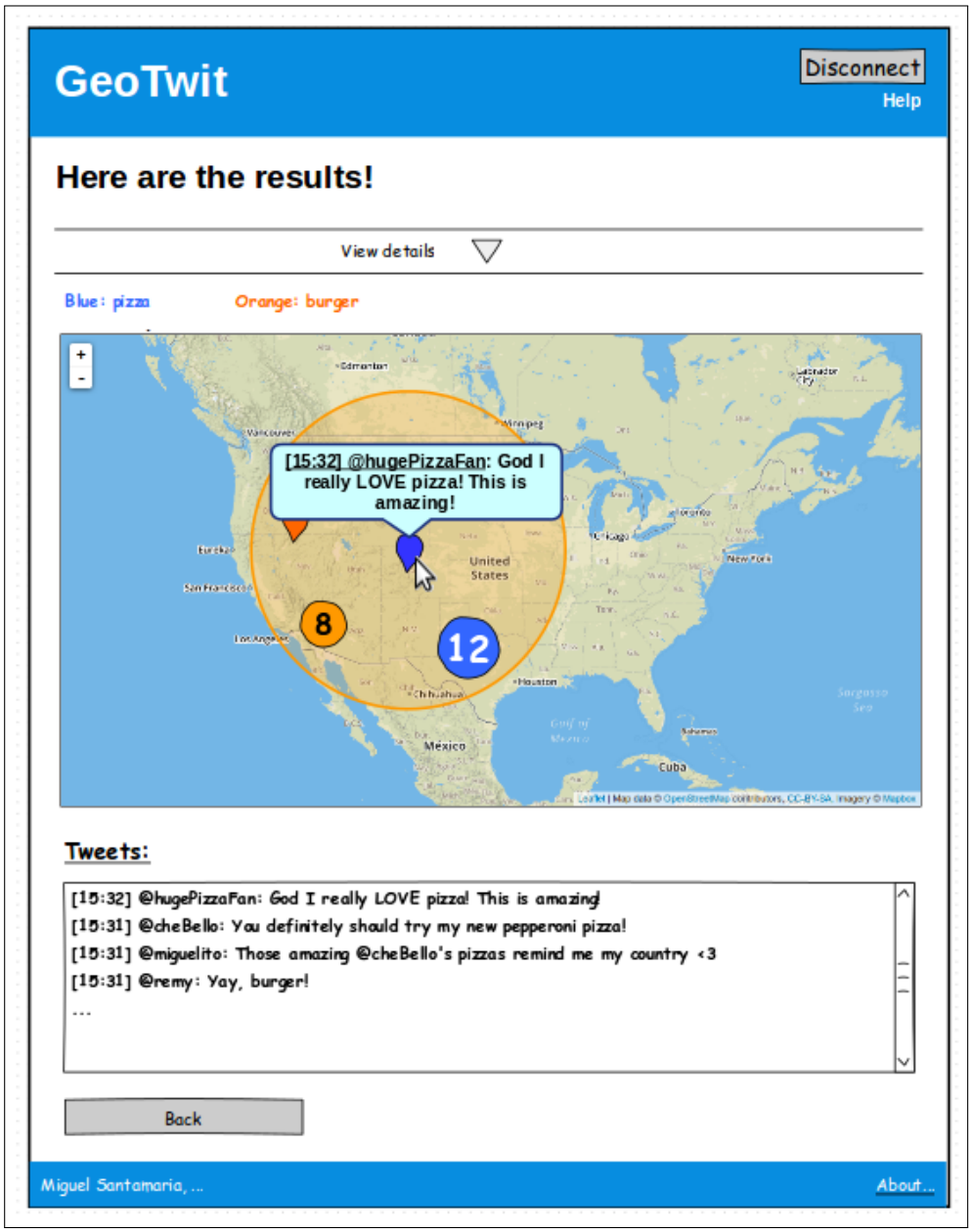


Figure 3.6: mock-up - view of the results in the static mode

There is no chart functionality in Static mode.

3.2 Anatomy of the Application

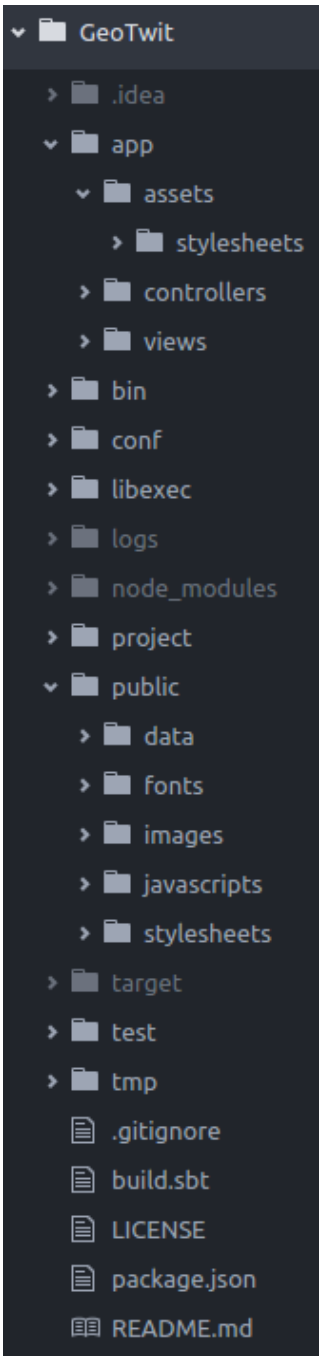


Figure 3.7: anatomy of GeoTwit

Here is the anatomy[1] of the application:

- **app**: contains all components related to the server, like controllers, views and assets.
  - **assets**: contains Less[2] files, which are converted to CSS when the server compiles the code.
  - **controllers**: contains all the application's controllers and actions, which act like standard API endpoints. It also contains action composition (see the "Actions Compositions" chapter below for more information).
  - **views**: contains all views and sub-views of the application.
- **conf**: contains all the application's configurations, in particular the routes' configuration file.
- **logs**: contains the application's log files.
- **node_modules**: contains all the client's NPM[3] packages; you can find more information about them in the "Used JavaScript Libraries" chapter below.
- **public**: contains files related to the build properties of the project.
  - **data**: contains json and shapefile data files, that respectively contain all the possible search languages and territorial data of nearly all of the world's countries.
  - **javascript**: contains the Search page's JavaScript file as well as various JavaScript libraries (see the "Used JavaScript Libraries" chapter below for more details).
  - **stylesheets**: contains the various JavaScript libraries' CSS files.
- **build.sbt**: this file is used to build the application and contains the application's description, library dependencies, resolvers and a filter used to automatically compile the Less code.

Other files and folders are not important and are largely used by Activator and SBT to start and compile the application.

[1] More information here: https://www.playframework.com/documentation/2.5.x/Anatomy
[2] http://lesscss.org/
[3] https://www.npmjs.com/

3.3 Implementation Details - Server Side

3.3.1 UML Diagram of the Server

Here is the UML diagram of the application's server, made with an old Bachelor project of HEIG-VD, *Slyum*[4]:

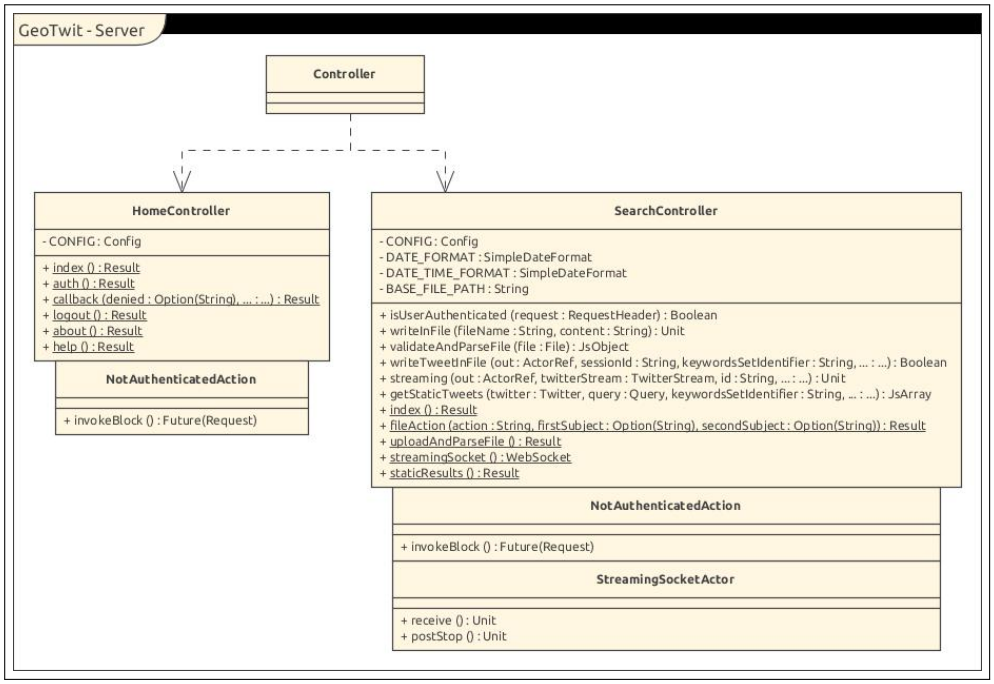


Figure 3.8: UML diagram of the application's server, made with Slyum

The controllers' actions are laid out in this schema; the blocks attached at the bottom of the controllers are subclasses: "NotAuthenticatedAction" are action composition classes used to check if the user is correctly connected before accessing an action (see the "Actions Compositions" chapter below); "StreamingSocketActor" is the class that represents a WebSocket actor / thread (also see the "Web Sockets" chapter below). Certain parameters of the "callback" method of the **HomeController** as well as the "writetweetInFile", "streaming" and "getStatictweets" methods of the **SearchController** were explicitly removed from the schema due to too much volume and space taken up by them. Here are the three methods' complete signatures:

- *callback(denied: Option[String], oauthToken: Option[String], oauthVerifier: Option[String]): Result*
- *writetweetInFile(out: ActorRef, sessionId: String, keywordsSetIdentifier: String, internalId: Int, creationDate: String, longitude: Double, latitude: Double, user: String, content: String): Boolean*
- *streaming(out: ActorRef, twitterStream: TwitterStream, id: String, isAreaRectangle: Boolean, keywordsSetIdentifier: String, query: String, southwestCoordinates: Array[Double], northeastCoordinates: Array[Double], language: String): Unit*
- *getStatictweets(twitter: Twitter, query: Query, keywordsSetIdentifier: String, maximumNumberOfRequests: Int): JsArray*

[4]https://github.com/HEIG-GAPS/slyum

The **SearchController** entity does not contain all attributes, as it would have significantly increased the project's complexity; the missing ones are the constant variables that contain the backup file's string values (METADATA_STRING, containing "METATADA:", etc.).
As you can see, there are two controllers: **HomeController** and **SearchController**.

HomeController is related to all actions linked to the Home page and miscellaneous pages, and the connection process:

1. **index** → displays the Home page and an error is the flash scope "error" is set (see the "Sessions and Flash Scopes" chapter).

2. **auth** → occurs when the user clicks on the "Get Started" or "Connect" buttons of the Home page; redirecting the user to Twitter's connection page, giving it the URL of the callback function, which corresponds to the "callback" action below.

3. **callback** → redirects the user either on the Search page if the connection was successful or on the Home page if there was an error or if the user denied the connection process. The status of the connection process depends on the three optional parameters: if "denied" is set, this means that the user denied the connection process and that the two following parameters are null. If "denied" is not set, this means that the two following parameters must have a value, respectively the token's string value (which will be used for the next requests to the APIs) and the "verifier" of the Twitter's OAuth process as a string (which will be asked by Twitter to verify the requests).
   This action is called anyway by the Twitter's API when the user leaves the Twitter's connection page.

4. **logout** → logs the user out and redirects him to the Home page.

5. **about** → displays the About page; the user does not have to be authenticated to access this page.

6. **help** → displays the Help page; the user does not have to be authenticated to access this page.

SearchController is related to the Search page (search and display of the results); all parameters are explained directly in the code so they are not detailed here; all actions require the user to be connected:

1. **isUserAuthenticated** → checks if the user is correctly authenticated and returns a Boolean value; this method is used by Ajax actions in the "AuthenticatedAction" actions composition.

2. **writeInFile** → writes the given string value in the given file's name, and it used to write the data in the file-to-export.

3. **validateAndParseFile** → Validates and parses the given file-to-import in order to export its data within the application; the file must be a well-formatted GeoTwit file (".gt" extension, containing metadata, tweets and results).

4. **writetweetInFile** → writes the tweets whose information are given in parameters in the file to export.

5. **streaming** → starts a new Twitter's streaming process, according to the given parameters.

6. **getStatictweets** → collects the tweets associated to the given query with the given Twitter object, while not exceeding the specified upper limit of requests.

7. **index** → displays the Search page, allowing the user to search for tweets with the Twitter APIs.

8. **fileAction** → gets the file containing the previous streaming's results and either downloads or deletes it, depending on the given "action" parameter; returning a BadRequest result if the file or the given action does not exist.

9. **uploadAndParseFile** → parses and validates the uploaded file to import.

10. **streamingSocket** → this special action opens a WebSocket's connection (through the internal "StreamingSocketActor" class) between a new server's actor (thread) and the client when this one accesses this entity; this connection receives and sends json values.

11. **staticResults** → gets and returns the results of the static mode search, when the user clicks the "View Results" of the "Static Mode" tab in the Search page.

More information about these actions and methods is further detailed in the following chapters.

3.3.2 Behavior of the Views

All action's views inherit from one of these two interface templates:

- *mainIndex.scala.html* if the current page is the home page. It contains only the header and footer partials and calls the page's content.

- *main.scala.html* for all other pages; in addition to containing the page's header and footer, it also contains a page content format, which allows the views to have the same page format (page's header, content, page's footer, etc.) on all pages.

As mentioned above, there is a header (*header.scala.html*) and a footer (*footer.scala.html*) partial templates, which are loaded in all templates. They respectively contain all HTML headers (page title, CSS and libraries loading, etc.) and footers (closing tags).

The *search.scala.html* search page's view is separated in three partial templates: *searchContent.scala.html*, which contains the search fields of both dynamic and static modes, and *searchStreamingResults.scala.html* and *searchStaticResults.scala.html*, which respectively contain the components of the dynamic and static results' pages.

3.3.3 Configuration

Configuration files are located in the "/conf" folder. You can find the routes' configuration file in here, as well as the application's and Twitter's configuration files.

The Twitter file contains the consumer's key and secret code used to make requests to the APIs. They are in a separate file in order to avoid security issues: the developer can only push the application's configuration file (which contains non-confidential data) in the GitHub repository and keep the Twitter file local.

In order to be able to read the configuration in the code, the developer has to inject a Configuration object into the controller of his choice:

```
@Singleton
class HomeController @Inject() (configuration: Configuration) extends Controller { ??? }
```

Figure 3.9: injection of a Configuration object in a controller

3.3.4 Routes

Play Router

Routes[5] are set in the "/conf/routes" file, and are declared with three distinct parameters: the HTTP method (GET, POST, etc.), the URL of access ("/auth", etc.) and the controller and action in which the client will be redirected ("controller.HomeController.auth", etc.) and which can have parameters. They will be used by the server's routing process in order to determine the action required with the given URL. Note that there is also a route configured for the asset files (images, CSS, JavaScript, etc.), so the client can access all these files.

JavaScript Routes

In addition of the Play's main Scala routes and according to the documentation[6], the Play router is able to generate JavaScript code to handle routing from JavaScript running client side back to the application. The JavaScript router aids in refactoring the application; for example, when a URL or a parameter's name is changed, the JavaScript code will automatically get the new structure. In GeoTwit, JavaScript routing is used in the Search page of the application. Specifically, JavaScript routes are declared in the "search.scala.html" view and are used in the "search.js" file:

- **search.scala.html**: JavaScript routes are declared with a helper that generate the JavaScript code of a router named *jsRoutes*. This router contains five routes: the route of the Logout page (in order to redirect the user from the JavaScript code if he is not connected anymore), the one of the "streamingSocket" of the SearchController (in order to initialize a web socket's connection with the server, see the "Web Sockets" chapter below), the "fileAction" action (in order to either delete of download the file-to-export from the JavaScript code), the "staticResults" action (in order to ask the server for the static mode's results), and finally a router for the asset files (in order to access images, fonts and other public assets).

```
@*
* Allows us to use router's links in the JavaScript code.
* The first parameter is the name of the global variable that the router will be placed in.
* The second parameter is the list of Javascript routes that are included in this router.
*@
@helper.javascriptRouter("jsRoutes")(
    routes.javascript.HomeController.logout,
    routes.javascript.SearchController.streamingSocket,
    routes.javascript.SearchController.fileAction,
    routes.javascript.SearchController.staticResults,
    routes.javascript.Assets.versioned
)
```

Figure 3.10: declaration of JavaScript routes in the Search page's view

[5]More information here: https://www.playframework.com/documentation/2.5.x/ScalaRouting
[6]https://www.playframework.com/documentation/2.5.x/ScalaJavascriptRouting

In order to make the helper work, the developer has to include the request's header as a parameter of the view:

```
@* Implicit request is used to generate JavaScript code to handle routing from JavaScript code. *@
@(username: String)(implicit req: play.api.mvc.RequestHeader)
```

Figure 3.11: declaration of JavaScript routes in the Search page's view

- **search.js**: JavaScript routes are used with the *jsRoutes.controllers.[CONTROLLER_NAME].[ACTION_NAME]()* code. If one wants to access an asset entity (like an image or a font), it must replace the controller and action's names by the *Assets.versioned('ASSET_NAME')* code. It is then possible to get the URL of the current entity with the "url" method (which will for example generate the following link: "http://localhost:9000/logout") or also get a web socket's URL with the "webSocketURL" method (which could generate the following link: "ws://localhost:9000/streamingSocket").

```
window.location.replace(jsRoutes.controllers.HomeController.logout().url);
```

Figure 3.12: example of redirection to the Logout page with a JavaScript route

3.3.5 Sessions and Flash Scopes

In Play Framework, sessions do not really act like anyone might guess, because of the stateless status of the framework: according to the documentation[7], data is in fact not stored by the server, but is added to each subsequent HTTP request, using the cookie mechanism. This means that we can only store string values, and not objects. Since this limitation is very annoying, the developer has to use the cache mechanism to store objects (like the Twitter's signed object used to make requests in GeoTwit), by adding a unique ID related to the current user's session for each cached data instance (see the "Cache" chapter below).

Each value stored in the session is identified by a string key and has a string value (like a "*String* → String" map). In order to add a value to the session, the developer has to use the "withSession" method right after the action's result (*Ok*, *Redirect*, etc.). Since the sessions' data is added in each HTTP request, he has to get the current request's session and put it first in the new request's session in order to keep the older session's data.

```
Redirect(routes.SearchController.index).withSession(
  request.session +
  ("username" -> twitter.showUser(twitter.getId()).getScreenName())
)
```

Figure 3.13: example of addition of a string value to the session

If one wants to set a new session, the app will just ignore the current request's session by removing the "request.session" call. In order to discard the whole session (during a disconnection for example), use the "withNewSession" method right after the action's result.

The session's time-out value is set in the "conf/application.conf" configuration file, specifically in the "play.http.session.maxAge" section. The value currently is 7 days.

[7] https://www.playframework.com/documentation/2.5.x/ScalaSessionFlash

In GeoTwit, there can be two types of information contained in the session:

- **id**: a unique ID used to identify the current user's session and its cache objects. This ID is used to name and identify these cache objects (since the cache is not related to one user initially and is available across users). This ID is generated with the Java's UUID generator and is set in the "auth" action of the **HomeController**.
- **username**: the username of the connected user, which is displayed in the header of the Search pages and is used to know if the user is connected or not (coupled with the cache - see the "Actions Compositions" chapter below). This value is set right after the connection with Twitter, in the "callback" action of the **HomeController**.

In addition of sessions, the framework also offers another tool: the flash scope. This mechanism works like sessions, with the difference that data is kept for only one request. It is used in GeoTwit to forward success/errors codes when an action redirects the user to the Home page.

```
case _ => Redirect(routes.HomeController.index).flashing("error" -> "sessionExpired")
```

Figure 3.14: example of use-case of a flash scope in GeoTwit

The index action of the **HomeController** gets the flash message and sends it to the view, which displays (or not) the right message, according to the status. If there is no error, the controller just sends a "success" message to the view.

3.3.6 Cache

As mentioned before, the cache[8] is used to store objects that cannot be stored in sessions. Though, there is one main problem with this method: each time the server compiles a new code, the cache is cleared so the currently connected user will be disconnected. When the application will be put in production environment, this problem will occur much less frequently.

In order to use the cache, add the "cache" library dependency in the *build.sbt* file (normally already done by default), then inject a *CacheApi* object in the controller in which you want to use the cache functionalities:

```
class HomeController @Inject() (cache: CacheApi) extends Controller { ??? }
```

Figure 3.15: injection of a CacheApi object in a controller

It is then possible to use this "cache" object to read and write the cache.
When an object is added in the cache, the developer can set a time value, which will determine the period of validity of this object; once the limit is reached, the object will be removed from the cache.

```
cache.set(uniqueId + "-tmpTwitter", twitter, 2.minutes)
cache.set(uniqueId + "-requestToken", requestToken, 2.minutes)
```

Figure 3.16: example of addition of objects in the cache, for a 2 minutes' period

[8] https://www.playframework.com/documentation/2.5.x/ScalaCache

In order to read a cache object, this one must be converted to the right type (a RequestToken object in the example below):

```
val getRequestToken = cache.get[RequestToken](id + "-requestToken")
```

Figure 3.17: example of reading a cache object

The developer has to think about testing the collected object's existence, in order to ensure it is still available.

Here are the different objects that can be stored in the cache in GeoTwit ("[id]" represents the unique ID generated for the current session, in order to differentiate the cache objects stored on the server side between each connected client):

- **[id]-tmpTwitter** (2 minutes' validity)
  Contains the **Twitter** object used to get the request token and the Twitter's authentication URL. This object is stored in the "auth" action of the **HomeController** and is collected in the "callback" action right after the Twitter's connection. It has to be stored, because it has already been signed with the request token, and still has to be signed with the access token in the callback, in order to be able to make requests to the APIs. This object is thus reused to build the final **Twitter** object, and since it is not possible to build a new object for security issues, it has to be kept.
  If this object times out before the user reached the callback, he will be redirected to the Home page with an explicit error.

- **[id]-requestToken** (2 minutes' validity)
  Also stored in the "auth" action and used in the "callback" action, this object contains the current user's request token, which will be used to generate the access token. If this object times out before the user has reached the callback, he will be redirected to the Home page with an explicit error.

- **[id]-twitter** (currently has a 7-day validity, which is the same value as the session's time-out and is stored in the application's configuration file)
  Contains the **Twitter** final and signed object, which will be used to make requests to the Twitter's APIs in the **SearchController**. It is set in the "callback" action of the **HomeController**, right after the two cached objects above have been removed. If this object expired for whichever reason, the user will be considered as disconnected and thus will be redirected to the Home page.

3.3.7 Twitter4J

Twitter4J is used in the **HomeController** and is automatically loaded by SBT with the *build.sbt* file. As mentioned before, this file contains the Twitter4J's library dependency:

```
org.twitter4j" % "twitter4j-core" % "4.0.4
```

The first string is the library's location, the second one is the library's name and the last one is the library's version. In order to use the streaming API, the file also has to contain the Twitter4J's stream library:

```
org.twitter4j" % "twitter4j-stream" % "4.0.4
```

3.3.8 Connection Process

In order to properly implement the connection process, the code is based on the *yusuke*'s implementation example of a connection process to Twitter with the Twitter4J library in Java[9] as well as a *Qiita*'s tutorial[10].

When the user clicks on the "Get Started" button located in the Home page, the following steps are operated:

1. The click action calls the "auth" method of the **HomeController**.
2. This action uses the Twitter4J library to make an authentication's request, in order to get the OAuth token and the authentication URL that points to the Twitter's authorization page. When making the request, the application also gives the API a callback URL (*/callback* => action "callback" of the **HomeController**), which will be called at the end of the authentication. This action also stores the temporary Twitter's and request token's objects in the cache for 2 minutes, in order to properly use them in the callback action.
3. Redirection of the user to the Twitter's authorization page.

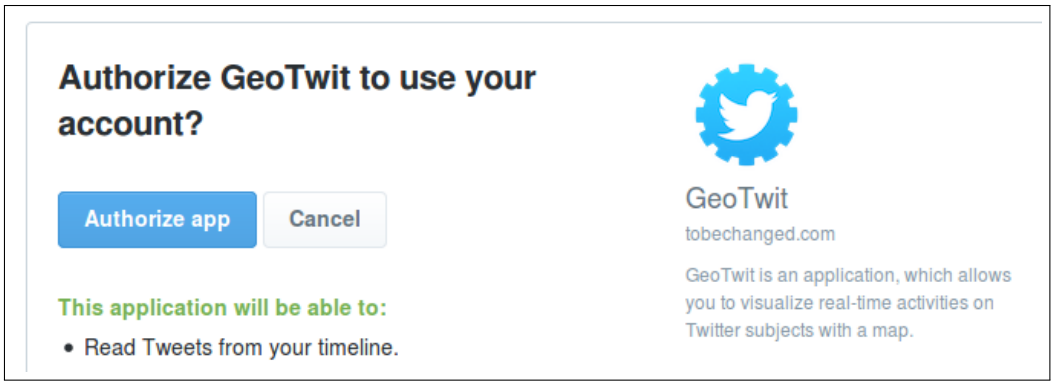


Figure 3.18: Twitter's authorization page in GeoTwit application

4. Whatever the user does, he will be redirected on the callback action:
   (a) If the user successfully authorized the application, the callback receives the OAuth token and verifier from Twitter. It will then get and memorize the OAuth access token (according to the temporary Twitter and request token's objects received from the "auth" action, as well as the received verifier from Twitter), add the new Twitter object in the cache for 7 days (same time as the session's time-out) in order to use it in the **SearchController** to make requests to the API, and then finally redirect the user to the "search" action, passing the username in the session (in order to display it on the pages' header).
   (b) If something went wrong, the user is redirected to the Home page with an error.

[9] Available on GitHub here: https://github.com/yusuke/sign-in-with-twitter
[10] http://qiita.com/omiend/items/90163d29b465fb7ab8f0

3.3.9 Actions Compositions

Actions compositions[11] can be interpreted as generic action functionalities, allowing the developer to execute generic code in several actions without writing it again. In summary, this code is executed before the execution of the action itself; compositions thus act like filters (which contain code executed before any action when a request is made to the Play server), but for specific actions, and can be chained by forwarding the HTTP request through a pipeline-like process.

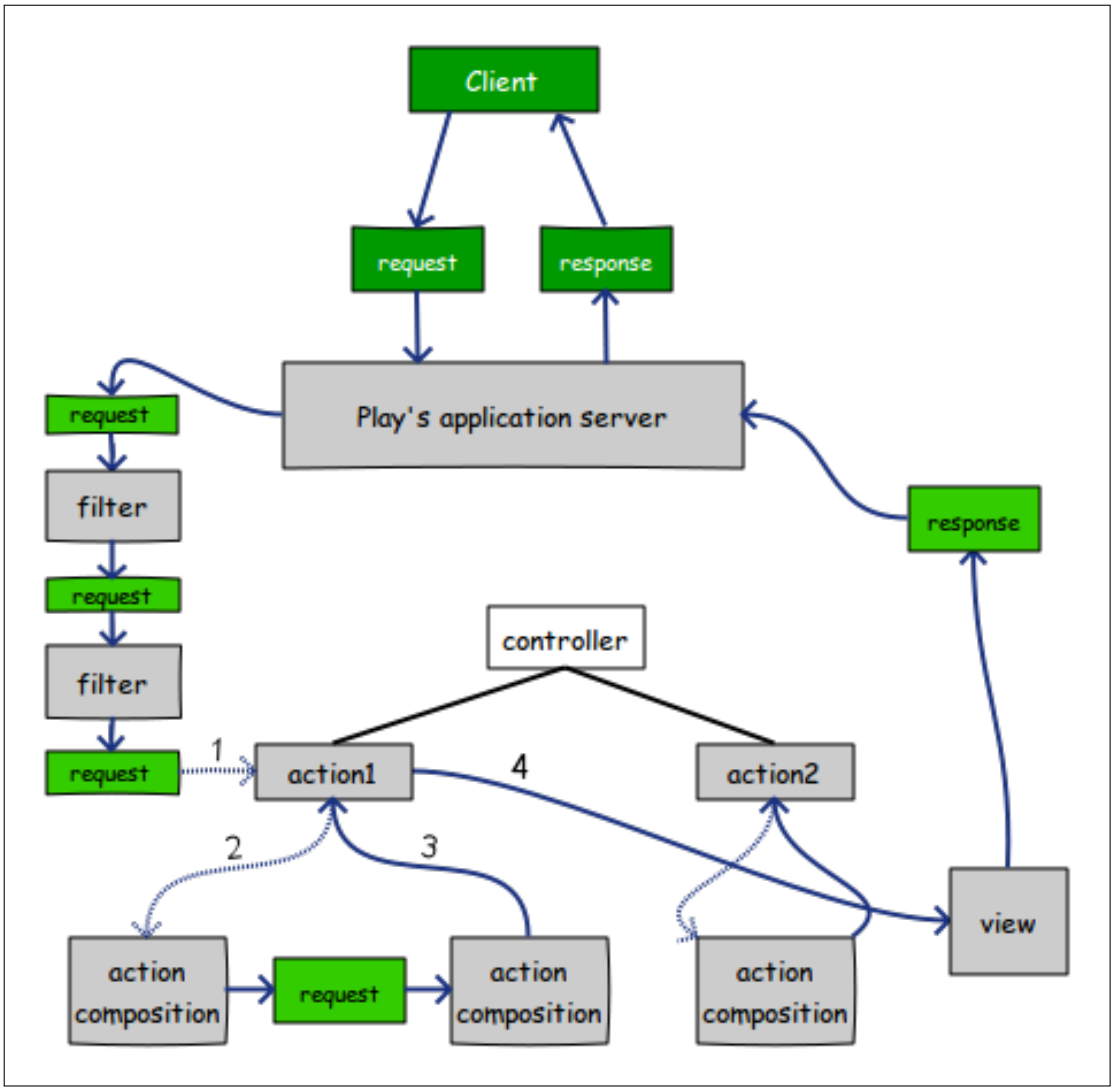


Figure 3.19: pipeline-like schema of filters and actions compositions, made with Evolus Pencil

In the above schema, the client first makes a request to the Play's application server, which sends it to the first filter. This filter executes its code and can perhaps modify the request's content; once done, he forwards the updated request to the next filter, and so on. When the request is finally forwarded to the right controller's action (through the routing process), this one calls the actions compositions related to itself, which alternately execute their code and can also update the request. Once the last action composition finishes its process, the request is finally forwarded to the action, whose body's code is executed and in which a result (for example the call of the view with the *Ok* keyword) will be provided.

[11] More information here: https://www.playframework.com/documentation/2.5.x/ScalaActionsComposition

In GeoTwit, actions compositions are used to check if the user is connected or not before accessing pages: for all the **HomeController** actions (except the "logout", "about" and "help" ones), the user is redirected to the Search page if he is already connected; on the opposite, if the user is not connected and tries to access a **SearchController**'s action, he will be redirected to the Home page with an error message. In order to determine if a user is disconnected or not, the actions compositions check if the "username" value of the session is set and if a *[id]-twitter* object exists in the cache for the current session (where "[id]" represents the unique ID of the current session). If one of these two values does not exist (or if the unique ID is not set), the user is considered disconnected.

These actions compositions are objects extending the ***ActionBuilder*** trait (the ***NotAuthenticatedAction*** object in the **HomeController**, and the ***AuthenticatedAction*** object in the **SearchController**), and implementing a "invokeBlock" method, which is called for every action built by the ***ActionBuilder***.

```
object NotAuthenticatedAction extends ActionBuilder[Request] {
  def invokeBlock[A](request: Request[A], block: (Request[A]) => Future[Result]) = {
    ???
    block(request)
  }
}
```

Figure 3.20: an example of an actions composition

The "request" parameter of the "invokeBlock" method corresponds to the current HTTP request object. The "block" parameter is a function used to wrap the request, by giving a future result to the HTTP request and thus allowing the request to be chained until the last actor (the action itself).

Actions are then defined with these objects, in order to execute the "invokeBlock" method code before the action itself.

```
def index = NotAuthenticatedAction {
  ???
}
```

Figure 3.21: an action using an actions composition

3.3.10 Web Sockets

According to documentation[12], web sockets are sockets that can be used from a web browser based on a protocol that allow a two-way full duplex communication. Both the client and the server can send and receive messages at any time, as long as there is an active web socket connection between them. In summary, one can use this technology to forward real-time messages from a client to a server and vice-versa; in GeoTwit, web sockets are used to forward data during the streaming process.

[12] https://www.playframework.com/documentation/2.5.x/ScalaWebSockets

In order to make the web socket's communication work, the application thus must have a web socket's client and server entities. In GeoTwit, the client is a JavaScript part located in the "search.js" file (modern HTML5 compliant web browsers natively support web sockets via JavaScript), and the server is located in the **SearchController**. Play provides two different mechanisms for handling web sockets: by using Akka Streams and by using iteratees. Since this second method is deprecated, it will not be explained below and Akka is the one which is used in the application.

Since it would be annoying to split the server and the client parts' explanations, they are both explained here.

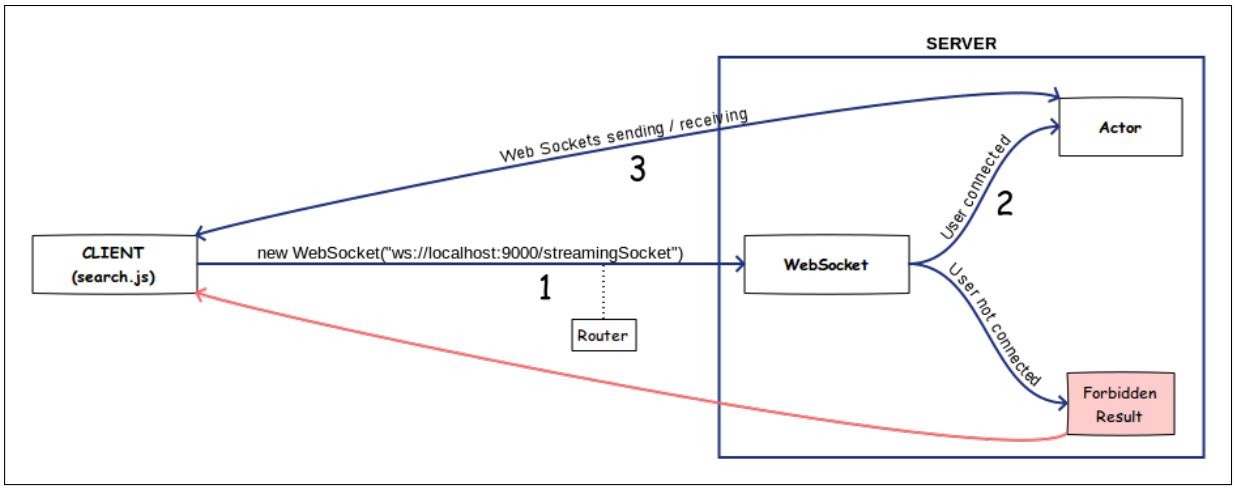


Figure 3.22: diagram of the web sockets' process is Play Framework, made with Evolus Pencil

In order to be able to send web sockets between a client and the server, first a connection has to be established. In GeoTwit, the following process is executed:

1. The client has to open a new web socket connection to the server, with the web socket protocol (*ws://[URL_OF_SERVER]*). A route is declared in the server's router in order to redirect the client's request to the right entity (which is not exactly an action in this case, but a *WebSocket* object).

```
// Gets the URL of the WebSocket's entity, by reversing the routing process
// and by using the jsRoutes object declared in the "search.scala.html" view.
socketConnection = new WebSocket(jsRoutes.controllers.SearchController.streamingSocket().webSocketURL());
```

Figure 3.23: opening of a new web socket connection from the client to the server

2. The *WebSocket* object of the server acted like an action until now, but no longer. Once it received the client's connection request, it checks that the current user is still authenticated (for security reasons) and if so, it creates a new *Actor* object through an *ActorFlow* object that allows the developer to access a "out" *ActorRef* object that will itself be used to send messages to the client. This *Actor* object acts like a Java Thread, which means it will keep a persistent connection with the client until request to close the connection (web browser's closing, reload of the current page, expressed web socket message indicating the server that the connection has to end, etc.), and which also means that the server can run other processes at the same time. In summary, the *WebSocket* object acts as a thread manager while the *Actor* objects act like threads; there can indeed be several current *Actor* instances running at the same time, but only one connection's *WebSocket* object, which ends as soon as the *Actor* is created.

In GeoTwit, the *WebSocket* object also gives the *Actor* the session's unique ID, so it will be able to access the current user's cache object. Note that the *WebSocket* object (and thus the *Actor* one) is configured to only accept and send Json data (in Play: *JsObject/JsValue* objects[13]), so it will be easier to identify and deal with web sockets' types. If the user is not connected anymore when the server receives a connection request, this will send back a Forbidden result that will make the client disconnect the user.

3. Once the *Actor* threaded object is created, it immediately sends a "successfulInit" message to the client, in order to notify it that the connection was successfully opened. As said before, only Json objects can be sent and received in GeoTwit.

   In order to send sockets to the client, the *Actor* object has to use the given *ActorRef* ("out") object:

```
// Sends a successful initialization's status as soon as the connection has been established.
out ! JsObject(Seq(
  "messageType" -> JsString("successfulInit"),
  "northeastCoordinates"  -> JsArray(Seq(JsNumber(northeastCoordinates(0)), JsNumber(northeastCoordinates(1)))),
  "southwestCoordinates"  -> JsArray(Seq(JsNumber(southwestCoordinates(0)), JsNumber(southwestCoordinates(1))))
))
```

Figure 3.24: example of a web socket's path from the server to the client

   When a message is sent from the client, the *Actor* receives it with the "receive" method. This method gets the received json message's type and process actions depending on it. Finally, the "postStop" method is triggered when the web socket's connection ends, and is used to properly close the ongoing streaming process. In order to manually ends the *Actor* threaded object, the developer has to send a *PoisonPill* object to the *ActorRef* ("out") object with the following command:

```
out ! PoisonPill
```

4. On the client side, everything is managed with the *socketConnection* object, initialized when the client first sent a connection request to the server. This object contains several methods, among which the "onmessage" one that is triggered when the client receives a message from the socket. Like in the server side, when a message is received, the client first gets the received json message's type and process actions depending on it.

Here are the messages that can be sent from the server to the client in GeoTwit (the first Json object's attribute is the message type, sometimes followed by various parameters):

- **{"successfulInit"}**: sent when the client's connection request has been successfully received by the server; when the client receives it, it shows the results page, draws the selected area (rectangle or complex polygon) on the results map and sends back a "readyToStream" message to the server.

[13] https://www.playframework.com/documentation/2.5.x/ScalaJson

- **{"newTweet", *keywordsSet*, *internalId*, *creationDate*, *longitude*, *latitude*, *user*, *content*, *nbReceivedtweets*}**: sent when a new tweet has been received through the Twitter's streaming process; the client adds the tweet to the map, displays it in the results panel, and sends a confirmation to the server, indicating that the tweet belongs to the country's territories (only if the user selected the country in the drop-down menu ; see the client's "readyToStream" and "tweetLocationConfirmation" web sockets for more information).
  The "keywordsSet" parameter indicates the keywords set identifier (either "first" or "second") to which the tweet belongs (so the client can display tweets with different colors). "internalId" contains the internal ID of the tweet, used to identify tweets in the file-to-export. "creationDate" contains the creation date of the tweet, "longitude" and "latitude" are the tweet's geographic coordinates, "user" contains the name of the user who posted the tweet and "content" contains the tweet's content. Finally, "nbReceivedtweets" contains the current number of received tweets (with or without geolocation tags), used by the graphs.

- **{"errorFile"}**: sent when an error occurred during the creation/writing of the file-to-export by the server. An alert is displayed in order to inform the user, but the streaming process continues.

- **{"stopStreaming", *reason*}**: sent when the server stops the streaming process (mainly in cases of exception); triggers the client to indicate to the user that the streaming has ended. The "reason" is an optional parameter that indicates the reason of this interruption, which can be:

  - *sessionExpired*: when the server cannot access the Twitter's cached object anymore, which means that the current user's session has expired; the client redirects the user to the Logout page in order to properly disconnect him. This error can only happen at the beginning of the streaming process, when the server tries to initialize the streaming connection with the Twitter's API; once done, the cached object is not used anymore and the streaming can continue (in order to avoid unwanted interruptions during the process).
  - *tooManyStreamingProcesses*: when the user ran too many (more than two) copies of the same application that are authenticated with the same account.
  - *queryTooLong*: when the user typed too many characters in one of the keywords sets (more than 60 in one OR phrase - see the "Building of the Twitter's Query String" chapter of the client side below); the search page is reloaded in order to make a new search.
  - *exception*: when a non-handled exception occurred; displays an alert window.

Here are now the messages than can be sent from the client to the server:

- **{"readyToStream", *isAreaRectangle*, *firstKeywords*, *secondKeywords*, *coordinates*, *language*}**: sent as soon as the client received a "successfulInit" message from the server; when the server receives this message, he starts the Twitter's streaming process(es) with the given user's input parameters. "isAreaRectangle" indicates whether the user manually drew a rectangle on the map (*true*) or selected a country in the drop-down menu (*false*); if *false*, this value indicates that the server must wait for the client to send a confirmation that indicates that the received tweet is in the geographical area that represents the country's territories (since the client only gave the bounding box to the server, in order to avoid to overload the connection - see the figure below), in order to only write tweets belonging to the area in the file to export; if the value is *true*, the server will automatically write tweets in the file, without asking the client for a confirmation. "secondKeywords" can be empty, since it is an optional input. "language" contains a BCP 47 language identifier[14] ("en", "fr", etc.) that corresponds to the value of the language the user selected to filter tweets (if empty, the system will simply not filter tweets by language). The "coordinates" parameter is an array that contains each coordinate of the rectangle's corners; this rectangle can either be the rectangle the user manually drew, or the rectangle bounding all the selected country's territories (since a country can have many of them) if the user selected a country in the drop-down menu. Here is an example of a bounding rectangle for French territories (aka France and Corsica):

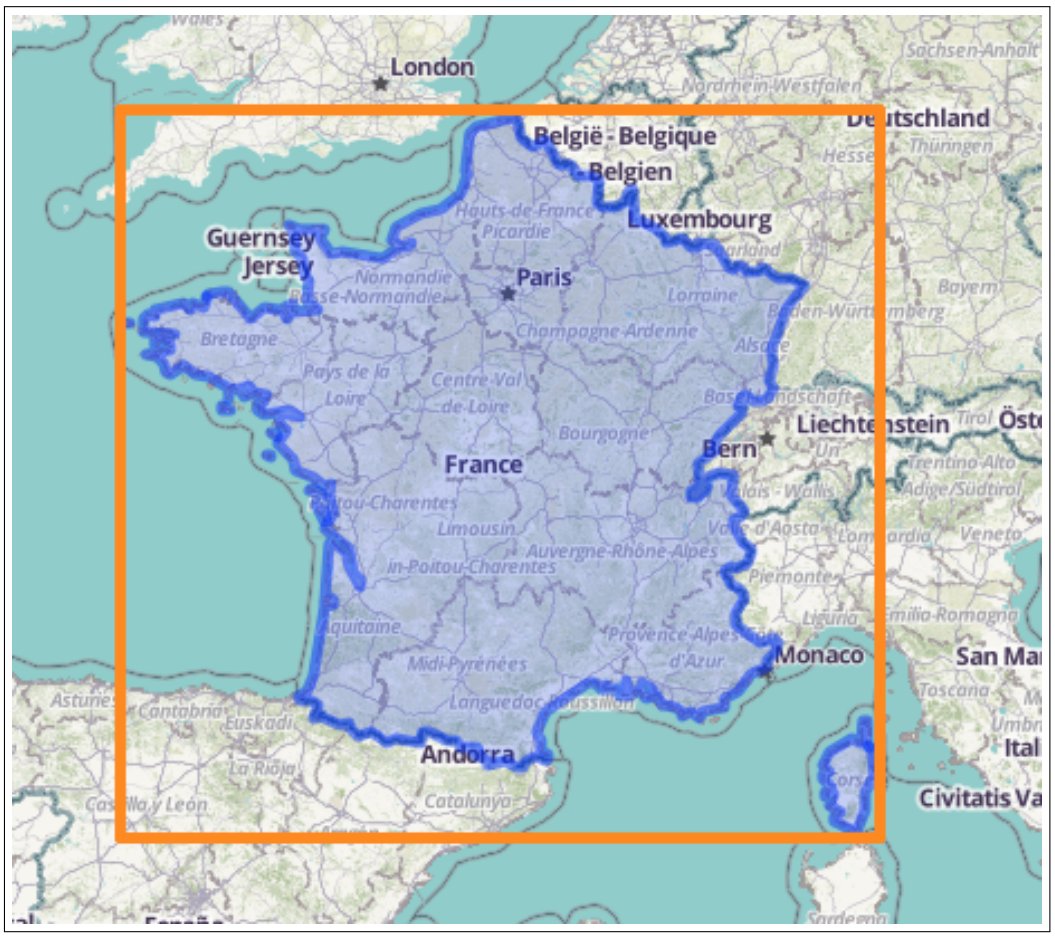


Figure 3.25: example of a rectangle bounding French territories

[14] https://en.wikipedia.org/wiki/IETF_language_tag

Since it is not possible to send huge amounts of data like the coordinates of all the country's borders through the web sockets system (and even if potentially possible, it is not optimal), the client only sends this bounding rectangle. Once received, the server will search for tweets contained within this rectangle, and will send them to the client, which will finally check that the tweet belongs to one of the country's territories (see the client side's "Reception of tweets from the Server" chapter below for more information).

- **{"currentResults", *elapsedTime*, *gtrt*, *grt*, *gprt*, *atrt*, *art*, *agvw*}**: sent each second by the client; gives the server the current results (the elapsed time, and the data related to each of the 6 charts: **gtrt** → Total of received geolocated tweets; **grt** → Reception of geolocated tweets; **gprt** → Parts of the received geolocated tweets; **atrt** → Total of all (with and without geolocation) received tweets; **art** → Reception of all tweets; **agvw** → tweets with geolocation tags vs. tweets without; more information about charts in the "Charts" chapter of the client side's section). The server internally saves and overwrites these results, in order to write them at the end of the backup file once the streaming process has ended.

- **{"tweetLocationConfirmation", *keywordsSet*, *internalId*, *creationDate*, *longitude*, *latitude*, *user*, *content*}**: sent by the client in order to confirm that the received tweet (received through the "newtweet" web socket) belongs to the selected country's territories, in order for the server to write it in the file to export. This socket is only sent if the user selected a country in the drop-down menu; indeed, only the bounding box is sent to the server at the beginning of the streaming process, in order to avoid overloading it (see the "readyToStream" web socket). If the tweet does not belong to the territories, the client simply ignores it. The "keywordsSet" parameters indicates which subject the tweet-to-confirm belongs to (either "first" or "second"), "internalId" is used to internally differentiate the tweets to write and results from an internal server's counter, and "creationDate" is the tweet's creation date, as a *YYYY-MM-DD'T'hh:mm:ss* format. Other parameters are the basic tweet's parameters.

- **{"stopStreaming"}**: sent when the user clicks on the "Stop Streaming" button; indicates to the server that he can kill the current *Actor* and stop the ongoing streaming process.

Here is a summary of the communication that occurs between the server and the client:

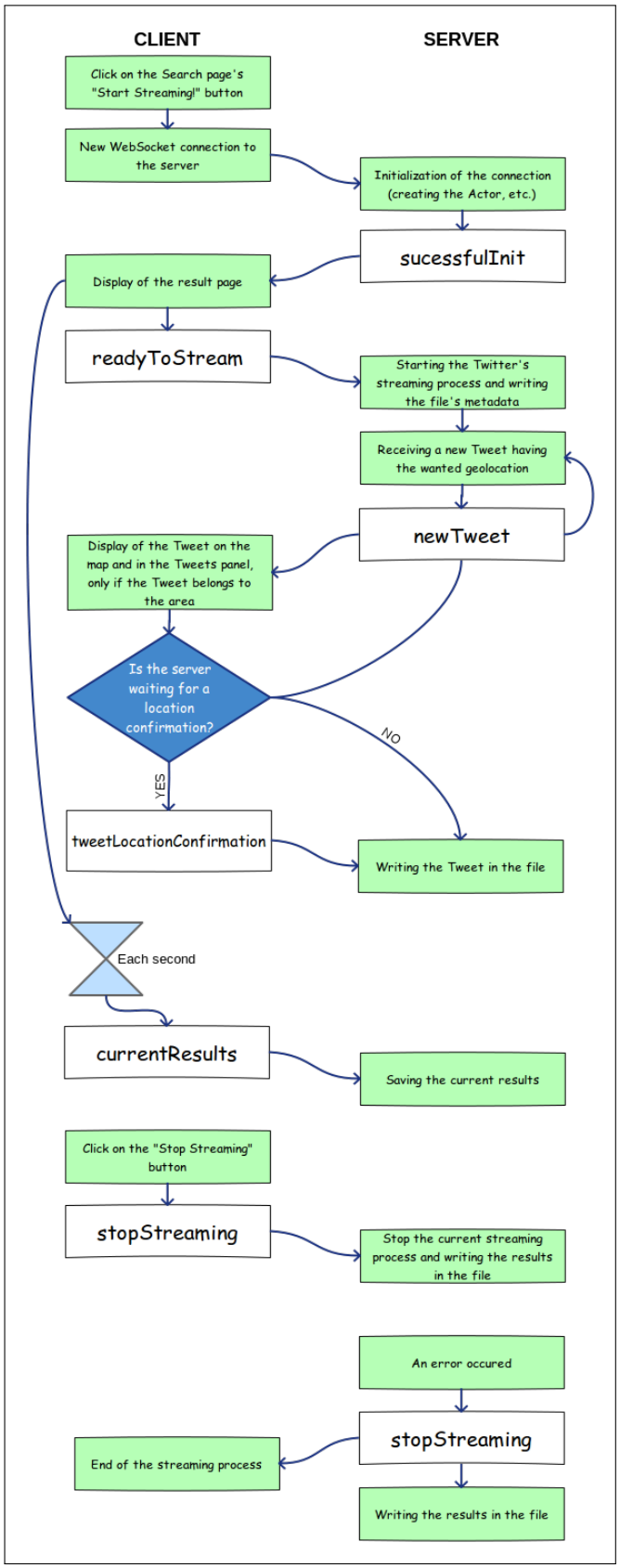


Figure 3.26: web sockets' operation diagram in GeoTwit, made with Evolus Pencil

3.3.11 Streaming Processes

Once the client sends all the data related to the streaming process(es) through the "readyToStream" web socket, the current web socket's *Actor* of the server validates the user parameters and checks that the user is still connected and only then, it starts the first streaming process through the Twitter API with the Twitter4J library. If the user filled the second keyword set/subject, the *Actor* starts a second streaming process request. As a reminder, the Twitter API only authorizes two simultaneous streaming processes per user, so it is not possible to add new keywords sets for now. The streaming process is located in the "streaming" method of the **SearchController**, and initializes a listener subscribed to a server of the Twitter's Streaming API with the both the right query parameters and a *TwitterStream* object initialized from the cached Twitter object. Once a tweet is received, it will trigger the "onStatus" method of the listener object, which will check if the tweet has a geolocation tag and secondly, if it is located in the desired rectangle. If valid, the *Actor* will send a "newtweet" web socket to the client.

If a user deletes a tweet previously posted, the information will be received in the "onDeletionNotice" method of the listener object; Though deletions are ignored in GeoTwit, the tweet still deserves to appear in the results despite deletion. If an exception occurs during the streaming process, the "onException" method of the listener object will be triggered. This one tries to convert the *Exception* object into a *TwitterException* object, in order to get the exception's status code and thus to properly inform the client about what happened through the "stopStreaming" web socket.

Further details on each step can be found in the extensively commented code, as it would be wasteful to go into them here.

3.3.12 Generation and Export of Files in Streaming Mode

At the end of a streaming process, the user is prompted to export the backup file of the analyzed subject(s) generated during the process. Both the client and the server take action during this process, but everything is explained here, in order to avoid confusion.

To begin with, here is the format of the generated text file:

```
METADATA:
        FIRST_SUBJECT: job nursing
        SECOND_SUBJECT: beach
        LANGUAGE: ANY
        COORDINATES: [[-128.6367166042328,21.109098973260398],[-128.6367166042328,51.057968857986694],[-65.00390410423279,51.05
TWEETS:
        first-subject#1;2016-07-22T17:42:38;-95.992775;36.1539816;"TMJ-TUL Nursing Jobs";"Want to work in #Tulsa, OK? View our
        first-subject#2;2016-07-22T17:42:38;-93.2922989;37.2089572;"Missouri Nursing Job";"See our latest #Springfield, MO #job
        first-subject#3;2016-07-22T17:42:39;-73.9209701;41.7003713;"TMJ-NY-USA Nursing";"This #job might be a great fit for you
        second-subject#1;2016-07-22T17:42:41;-74.43198772;39.35509086;"Ms.Choya";"About to hit the beach....have an amazing day
        first-subject#4;2016-07-22T17:42:48;-83.7430378;42.2808256;"TMJ-MIN Nursing Jobs";"See our latest #AnnArbor, MI #job ar
RESULTS:
        ELAPSED_TIME: 00:00:24
        GTRT: [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,3,3,3,3,3,3,3,3,3,3],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1]]
        GRT: [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0]]
        GPRT: [3,1]
        ATRT: [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,3,3,3,3,3,3,3,3,3,3],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,43,43,43,43,43,43,43]]
        ART: [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,43,0,0,0,0,0,0]]
        AGVW: [[100,0],[2.33,97.67]]
```

Figure 3.27: format of the generated text file that contains streaming process(es) data

Spaces and tabulations were added in this example in order to make the file more readable; in real files, these are not included, in order to gain space.

And here are the explanations:

- **METATADA**: this section contains the metadata of the current streaming process and is located at the beginning of the file. The metadata contains:
  - the **first subject's search-string**;
  - the **second subject's search-string**, only if the user inputs a second keyword set; if not, this line does not exist;
  - the **language** used to filter the tweets ("ANY" if the user did not select a specific language);
  - the **coordinates** used to filter the tweets, as a string-representation of an array of arrays of double (see the illustration above for an example).
- **TWEETS**: this section contains the data of all received geolocated tweets, and is located after the metadata. There is strictly one tweet per line, and each tweet contains the following information, separated by semicolons:
  - the **identifier** of the current tweet, which is formatted as the following: "*[SUBJECT_IDENTIFIER]-subject-#[TWEET_ID]*", where *[SUBJECT_IDENTIFIER]* corresponds to the string identifier of the subject with which the tweet has been found (either "first" or "second"), and where *[TWEET_ID]* corresponds to the local counter value of tweets received for the current subject until now (in order to identify tweets of a same subject between them);
  - the **date and time** that represent the creation of the current tweet, following a *YYYY-MM-DD'T'hh:mm:ss* format, where '*T*' is a simple character used to separate the date from the time (and which is used by the Scala parser in order to convert this string into a date);
  - the **longitude** of the tweet;
  - the **longitude** of the tweet;
  - the **name of the user** who posted the tweet, bounded by quotation marks (in case the user's name contains a semicolon);
  - the **text content** of the tweet, also bounded by quotation marks for the same reason; if a tweet's content contains line breaks, they are automatically removed.
- **RESULTS**: this section contains the results of the streaming process, and is located at the end of the file, after the tweets. The results contain:
  - the **time** the process lasted, formatted in *hh:mm:ss* format;
  - the **graphs' data** (one line per graph: **GTRT** → Total of received geolocated tweets; **GRT** → Reception of geolocated tweets; **GPRT** → Parts of the received geolocated tweets; **ATRT** → Total of all (with and without geolocation) received tweets; **ART** → Reception of all tweets; **AGVW** → tweets with geolocation tags vs. tweets without). All graphs have a two-dimensions array of two arrays of numbers, except the GPRT one, which only have a simple array of numbers. If a chart does not exist (if there is only one subject the GPRT one will not exist, for instance) or is empty, an empty array is written (for example: "[[],[]]", or "[]" for the GPRT).

  Since all the collected tweets are public, there is no confidentiality issues about the storage of this data in an external file.

During the process, the file is initially created in the "writeInFile" method of the **SearchController**, if it does not already on the server side. This method writes the given string value in the file whose name corresponds to the given string. For the streaming process, the files are located in the "/tmp" file and are named as the following: *streaming-[ID].gt*, where *[ID]* corresponds to the unique ID of the current session of the connected user. Every connected user thus has a file having a unique name for his current session. This also means that there can only be one file per session, which allows the server to not be overloaded by files. *gt* is the self-assigned official extension of GeoTwit files, but is just a text file in reality. Note that the file is deleted once it has been downloaded or rejected, as explained later in this chapter.

The first time the "writeInFile" method is called (and thus the file is created), it occurs in the "receive" method of the *StreamingSocketActor* class of the **SearchController**, when the "readyToStream" web socket type is received and when the user's inputs have been validated by the server. This first call allows the server to write the metadata of the current streaming process at the beginning of the file.

Then, this method is called each time the server receives a new tweet from the Twitter's server, either in the "onStatus" method of the listener located in the "streaming" method of the **SearchController**, or in the "receive" method of the *StreamingSocketActor* class, when the actor received a "tweetLocationConfirmation" web socket type (depending on whether user manually draws a rectangle on the map or selects a country from the list - more information in the "Web Sockets" chapter above). This call writes the received/confirmed tweet in the file.

Each second, the client sends the current elapsed time and the chart results to the server (with the "currentResults" web socket), which overwrites them each time. As soon as the streaming process is stopped (either because an exception was thrown or because the user clicked the stop button), the server writes the last received results at the end of the file (still with the same "writeInFile" method). Simultaneously, the client asks the user if he wants to export the generated file, in the "stopStreaming(...)" function of the *search.js* file. If the user says *yes*, he will be able to download the file; if he says *no*, the file will be removed from the server, in order to keep it clean at all times. Note that the user is given a prior warning about the results of his decision.

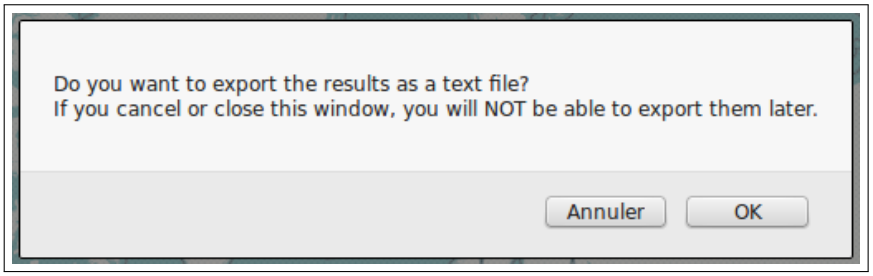


Figure 3.28: pop-up asking the user if he wants to export the data as a text file (French buttons are due to the browser's default language)

In order to easily download the file and to get a feedback about the download process (success/failure), the *johnculviner*'s "jquery.fileDownload" Jquery's download library[15] was used. It allows the developer to use an Ajax-like file-download experience, which is normally not (easily) possible by using the web.
The developer just needs to call the library, give it the URL of the action from which the user can download the file. In GeoTwit, this action is the "fileAction" action of the **SearchController**, which is used either to download or delete the current user's file (you can find more information in the following paragraphs). Here is a sample of code, which is used to download the file:

```
$.fileDownload(jsRoutes.controllers.SearchController.fileAction(/*???*/).url)
    .done(function () {
        // Deletes the file from the server once it has been downloaded.
        deleteFile();
    })
    .fail(function () {
        alert("An error occurred [...]");
    });
```

Figure 3.29: sample of code used to download the generated file

Once the download is finished, the file is immediately removed from the server, thanks to the "deleteFile" function of the *search.js* file. If an error occurs, an alert message containing the URL of the download page is displayed to the user, so he can manually download it. Note that the next time the user will press the "Start Streaming!" button, the file will automatically be removed (in order to avoid storing useless files on the server side).

The "deleteFile()" function only sends an Ajax request to the "fileAction" action of the **SearchController**, in order to delete the file. It does not wait for a response from the server.

Regarding the "fileAction" action of the **SearchController**, this one either downloads or deletes the current user's file, according to its first string parameter (which can either be "download" or "delete"). This action also needs the string(s) that represents the keywords set(s), in order to properly name the file to download. It first gets the file on which the action will be executed, and checks if this file exists. If not, a *BadRequest* response is returned; if yes, the server performs the intended action. If the file needs to be deleted, the server simply executes the "delete" method of the file object; if the file needs to be downloaded, the server has to serve it to the client. In order to accomplish this task, the action must render a file instead of an HTTP result[16]:

```
Ok.sendFile(
  content = file,
  fileName = _ => fileName,
  inline = false
).withHeaders("Set-Cookie" -> "fileDownload=true; path=/")
```

Figure 3.30: rendering of a file at the end of an action

[15] https://github.com/johnculviner/jquery.fileDownload
[16] More information here: https://www.playframework.com/documentation/2.5.x/ScalaStream

The content contains the *java.io.File* object to send; the file name contains the name of the file that the user will see in the download pop-up, and is formatted as the following: *[SUBJECT_1][_[SUBJECT_2]]_STREAMING_[DATETIME].gt*, where *[SUBJECT_1]* and *[SUBJECT_2]* (optional) represent the keywords sets' strings, and where *[DATETIME]* represents the current date and time, following a *YYYY-MM-DD'T'hh_mm_ss* format. The "inline" parameter indicates if the file must be served as an inline file (which means that it is directly displayed in the web browser instead of being downloaded through a pop-up).

The "Set-Cookie" HTTP header added at the end of the *Result* object is used by the "jquery.fileDownload" library in order to receive a confirmation when the file has been successfully downloaded (in order to remove it from the server).

An example of generated file can be found in the */tmp/EXAMPLE.gt* file.

3.3.13 Importing Generated Files

GeoTwit also allows the user to import a previously generated file, in order to review the results (in a static way; the streaming does not resume). Here again, both the client and server sides are explained, in order to avoid confusion.

The user has the possibility to import a generated file by clicking a button located at the beginning of the Dynamic Mode tab's content. In summary, he has to click the button in order send the file to the "uploadAndParseFile" action of the **SearchController**, which will validate and parse the file, then send back the parsed data to the client as a json format. You can find more details about this process in the paragraphs below.

Client Side

The upload process begins as soon as the user chooses a file to upload. Firstly, in order to be able to send the file to the action with an Ajax request (for getting json data back from the server at the end of the process) and to be able to display a progress-bar about the upload, the basic plugin of the *blueimp*'s "jQuery-File-Upload" library[17]-[18] is used.

Figure 3.31: progress-bar managed by the jQuery-File-Upload library during an upload process

[17] https://github.com/blueimp/jQuery-File-Upload
[18] https://github.com/blueimp/jQuery-File-Upload/wiki/Basic-plugin for the basic plugin

In order to initialize the automatic Ajax upload, the ”fileupload” method is called on the ”file” HTML input. The given parameters indicate that the client expects the server to answer with json data and also provide the refresh function of the progress-bar.

```
// The URL of the route to which we send the file is determinated by the input file's data-url attribute.
$('#importedFile').fileupload({
    dataType: 'json',
    done: function (e, data) {
        // ...
    },
    progressall: function (e, data) {
        var progress = parseInt(data.loaded / data.total * 100, 10);
        $("#progressBarUpload").css("width", progress + "%");
        $("#uploadPercents").text(progress);
    }
});
```

Figure 3.32: initialization of the automatic Ajax upload process on the ”file” input

As mentioned in the code's comment, the URL of the route to which the client sends the file is determined by the HTML input file's ”data-url” attribute, which contains the route of the ”uploadAndParseFile” action.

The ”progressall” function episodically calculates the current progression value of the progress-bar by the current loaded data (this number being received from the server), then refreshes the HTML components.

The ”done” function is called when the upload process is complete: it displays an error if the json data received from the server contains one, and in normal cases, calls the ”loadFileResultsComponents(...)” function of the *search.js* file, which loads the components (map, charts, HTML components) as well as the received data (display of the metadata, the tweets and the results).

Server Side

On the server side, the ”uploadAndParseFile” action first validates the file, which must exist and be a text file, and not be empty. If a validation error occurs, the following json object is returned:

- **error**: *true*;
- **reason**: the reason key of the error, which is either ”fileEmpty”, ”wrongFormat” or ”missingFile”.

If the file is valid, the action calls the "validateAndParseFile(...)" method of the same controller, which validates the contents of the file. This method returns a json object containing either an error if there was one or the parsed data of the file if everything went smoothly:

- The file's content must first have the three "METADATA", "TWEETS" and "RESULTS" sections.
- The "**METADATA**" section must have at least the "FIRST_SUBJECT" (*String*), "LANGUAGE" (*String*) and "COORDINATES" (array of arrays of *Double*) values; the "SECOND_SUBJECT" (*String*) value is optional. The order of the values does not matter, as long as all of them (except the optional one) are present in this section.
- The "**TWEETS**" section can be empty (if there was no received tweet during the analysis) and is directly followed by the "RESULTS" section in this case; if it contains tweets, these have to be properly formatted (see the contained data in the "Generation and Export of Files in Streaming Mode" chapter above): the date must be as the *YYYY-MM-DD'T'hh:mm:ss* format, the longitude and latitude must be *Double* types, and the user's name and tweet content must be bounded by quotation marks. There must be strictly one tweet per line.
- The "**RESULTS**" section must contain the "ELAPSED_TIME" (*String*) value, and the data linked to the charts ("GTRT", "GRT", "ATRT", "ART", "AGVW" (arrays of two arrays of *Double*, which can be empty => "[[],[]]"), and GPRT (simple array of *Double*, which can also be empty => "[]")). The order does not matter here, as long as each of these values is contained in this section.
- There should not exist any unnecessary spaces, tabulations or line breaks.

While parsing the file for validating it, the algorithm also simultaneously collects all the data in order to gain time; if a validation error occurs, this data is dropped. The algorithm is divided into three parts:

- Validation and collection of the "METADATA" section, and retrieval of the other section's start lines.
- Validation and collection of the tweets if the metadata were valid and if the file contains the two other sections.
- Validation and collection of the results if the two other parts were valid, the return of the json results.

In order to validate and collect the values, the algorithm uses regular expressions with parentheses (in order to group data and get them as variables) and match cases to validate them. For example, if the algorithm wants to get the language value of the metadata, it will use the following regular expression, which will match a line beginning with "LANGUAGE:" and followed with a value containing at least one character:

```
val languageRE = (LANGUAGE_STRING + "(.+)").r
```

Figure 3.33: Regex used to match the language string in the imported file

The parentheses contained within the Regex (regular expression) is used to group the value in order to get it as a variable. The algorithm then tries to match the parsed file's lines with this regular expression, by collecting the group as a "l" variable.

```
for (line <- lines.tail) {
  line match {
    //...
    case languageRE(l) => language = l
    //...
  }
}
```

Figure 3.34: matching of the regex used for the language

The validation of the other metadata and results applies the same principle. Tweet validation is a little more complicated: the Regex validates and collects 6 different variables separated by ';'.

```
// Regular expression used to validate a Tweet entry and get its data.
val tweetRE = ("((?>first|second)-subject#\\d+);(\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2});" +
  "(-?\\d+\\.?\\d*);(-?\\d+\\.?\\d*);\"(.+)\";\"(.+)\"").r
```

Figure 3.35: Regex used to validate and get data of tweets

These variables must be in the following order and must be formatted as follows:

- The tweet's identifier, which must start either by "first" or "second" (the "?>" characters indicates to Scala that we do not want to group the character contained in bounded parentheses as a variable), followed by "-subject#", then by a number. For example: first-subject#2.
- The date, which must contain a string formatted as: *XXXX-XX-XXTXX:XX:XX*, where 'X' is a digit and where 'T' is the *'T'* character. For example: 2016-07-22T15:30:14.
- The longitude, which must be a double. For example: -87.9997769.
- The latitude, which also must be a double. For example: 47.8106521.
- The user's name, which must contain at least one character and must be bounded by quotation marks (note that only the bounded content will be taken into the variable). Quotes are used to avoid bugs with a string that would contain a colon. For example: "Speedway Jobs".
- The content, which must follow the same rules as the user's name. For example: "Hello everybody!".

Once every value of the file is valid and collected, the method sends back a json object, containing the following attributes:

- **error**: *false* (indicates that there was no error);
- **firstSubject**, **secondSubject** (can be an empty string), **language**, **coordinates**;
- **tweets** => an array containing each tweet's values: **subjectIdentifier** (either "first" or "second"), **dateAndTime**, **longitude**, **latitude**, **user** and **content**;
- **results** => an object containing the results: **elapsedTime**, **gtrt**, **grt**, **gprt**, **atrt**, **art** and **agvw**.

If there was an error, the method returns a json object containing:

- **error**: *true*;
- **reason**: the key reason of the error, which is "fileNotValid".

This object is forwarded to the client through the "uploadAndParseFile" action. If everything was valid, the client displays data on the screen.

3.3.14 Retrieving of Static tweets

Issues due to Limitations

Before explaining the concerned algorithms, more details have to be laid out about the function of Twitter's REST API when a large amount of data is available for the current user's request. Indeed, Twitter's server cannot send the whole amount of data at one time from a certain number of tweets, because of the problems that this action can cause (mainly slowness). In order to avoid these kinds of problems, Twitter sends the results by "pages", each page containing a certain amount (up to 100 - configurable with Twitter4J) of tweets. If there are still too many results, the API also limits the number of pages, which complicates the process: indeed, the developer has to get the lowest ID of the received tweets set, in order to make a new request that collects a new tweet set from this ID. Here is an example with a maximum of three tweets per page, and a maximum of two pages per results' set:

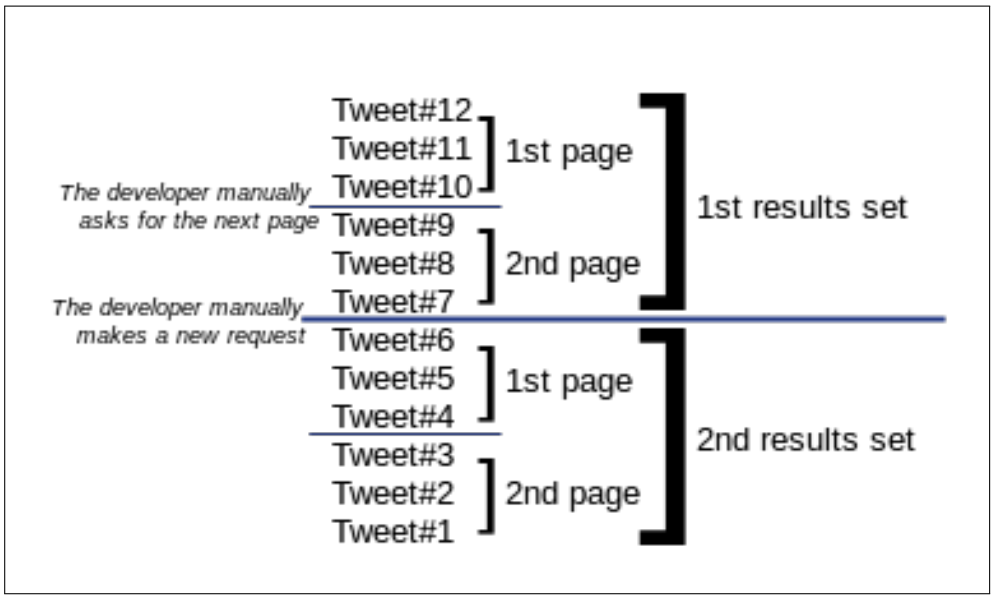


Figure 3.36: example of the limitations of the Twitter's REST API

In this example, the first request collects the tweets from #12 to #7 (from the highest to the lowest IDs, because Twitter logically firstly sends the most recent tweets), by generating two pages. The developer has to manually ask for the next page every time all the tweets of a page are collected, and has to execute a new sub-request with the new page. Once the first request does not have pages anymore, the developer has to get the lowest ID of the retrieved tweets (here: #7) in order to make a new request to collect the tweets from #6, and so on.

This system raises a huge issue: like it was said before in the analysis part of this document, Twitter indeed limits the number of requests that a user can make in a 15 minutes' period. Since every page's sub-request is considered as a request, and since the user may have to do various requests to get all results, the limit can quickly be reached. For this reason, the number of results are limited in GeoTwit.

If this case occurs and the obtained results are limited, the user has to be more specific about his subject(s) and date range. Twitter's REST API unfortunately only supports dates and is incompatible with specific timing, so one cannot fine-tune further than days.

Algorithms

When the user clicks on the search button of the static mode's page, the client makes an Ajax request to the server in order to retrieve the list of static tweets (see the "Retrieval of tweets with the Server" chapter in the client's static part below). The server uses the Twitter's REST API to collect tweets in the "staticResults" action of the **SearchController**; it first gets and validates the sent parameters, then creates the queries.

```
// Creates the query, according to the user's parameters.
val query: Query = new Query(fk)
query.setCount(100)
query.setSince(dateFormat.format(fd))
query.setUntil(dateFormat.format(td))
query.geoCode(new GeoLocation(lat, lng), rad, "km")
if (!lan.isEmpty) {
  query.setLang(lan)
}
```

Figure 3.37: creation of a query for asking the Twitter's REST API with Twitter4J

The "setCount" method is used to indicate the maximum number of tweets per page (which can be up to 100 maximum). "setSince" and "setUntil" methods take a *YYYY-MM-DD* date as a string value and are used to filter the tweets respectively by their start and end dates.

The action then calls the "getStatictweets(...)" method for each subject, which gets the tweets related to the given parameters and builds a json array object with them. Specifically, this method applies the algorithm seen above on pages and results' sets in order to avoid limitations as much as possible. Firstly, the "maximumNumberOfRequests" parameters indicates to the algorithm the maximum number of requests it can make before risking reaching Twitter's upper limit. After a few tests, it turns out that the maximum number of requests is close to 175 per account over a duration of 15 minutes (this value may be greater, but it provides a lower bound). The following example was generated by putting "println" after each request in the code:

```
Processing requets #169
Processing requets #170
Processing requets #171
Processing requets #172
Processing requets #173
Processing requets #174
Processing requets #175
Processing requets #176
Processing requets #177
Processing requets #178
Processing requets #179
```

Figure 3.38: example of the number of requests executed before getting an error in a period of 15 minutes

In order to keep within the margin of error, this number is truncated to **170** in GeoTwit (approximately 17'000 tweets), which means that if the user enters two different subjects, each search must at the most make **85** requests (approximately 8'500 tweets per subject). This calculation is managed by the "staticResults" action: the maximum number is contained in the constant variable "MAXIMUM_NUMBER_OF_REQUESTS"; when the action calls the "getStatictweets(...)" method, it indicates the number of requests to be made by the number of subjects that the user selected and by this constant value.

Due to these limitations and issues, the static mode is thus less developed than the streaming mode in GeoTwit. Also for the same reasons, **the "Accelerated Streaming View" tools that were planned in the specifications were aborted**; as a compromise, the file import functionality was developed in the dynamic mode.

3.4 Implementation Details - Client Side

3.4.1 Both Streaming and Static Modes

Structure of the client's JavaScript file

Since all the JavaScript code is contained in the */public/javascripts/search.js* file, there cannot be a decent UML schema. For this reason, here is the list of functions contained in this file (remember that JavaScript is a non-typed language), ordered by their order of appearance:

- *secondsToHhMmSs(seconds)*
- *cloneObject(obj)*[1]
- *erasePolygons(map)*
- *drawPolygons(map, coordinates, readFileMode = false)*
- *invertCoordinates(coordinates)*
- *isPointInSelectedArea(latitude, longitude, polygonsCoordinates)*[2]
- *loadCountriesList()*
- *loadLanguagesList()*
- *loadSearchMap(mapId, hasRectangle, hasCircle)*
- *loadStaticMap()*
- *loadResultsMap(mode)*
- *loadStreamingResultsCharts(hasSecondStream, refresh = true)*
- *calculateAndDisplaySpeedValues()*
- *loadStreamingResultsComponents()*
- *loadFileResultsComponents(data)*
- *loadStaticResultsComponents()*
- *selectCountryOnMap(countryName, map)*

- *initWebSocket()*
- *stopStreaming(sendSocket, displayExportPopup = true)*
- *deleteFile()*
- *addtweetOnPage(mode, subjectNumber, latitude, longitude, date, user, content)*
- *getStatictweets()*
- *validateDynamicFields()*
- *validateStaticFields()*
- *getAndFormatKeywords(keywordsSetNumber, mode)*
- *$(document).ready(function()*: occurs when the page successfully loaded all DOM elements.

Most of these functions have been explained in detail in the following chapters.

Used JavaScript Libraries

Several JavaScript external libraries have been used in GeoTwit, so as to avoid reinventing the proverbial wheel. About half of them are NPM[19] packages located in the */node_modules* directory, while the other half are external JavaScript libraries found on the web and located in the */public/javascripts* directory.

1. **NPM packages**:
    - **Chart.js**[20]: allows the developer to draw simple and responsive HTML5 charts.
    - **Leaflet**[21]: allows the developer to build interactive maps coupled with OpenStreetMap.
    - **Leaflet.markercluster**[22]: this plug-in for Leaflet is used to automatically group markers by clusters on the maps.
    - **Leaflet.draw**[23]: a vector drawing and editing plug-in for Leaflet, used to draw rectangles and circles on the maps (coupled with the next library).
    - **Leaflet.draw.drag**[24]: this library is used coupled with the *Leaflet.draw* one, in order to allow the user to draw and drag polygons (rectangles and circles) on the maps with a special toolbar.
    - **Moment.js**[25]: allows the developer to parse, validate, manipulate, and display dates in JavaScript.

[19] https://www.npmjs.com/
[20] http://www.chartjs.org/
[21] http://leafletjs.com/
[22] https://github.com/Leaflet/Leaflet.markercluster
[23] https://github.com/Leaflet/Leaflet.draw
[24] https://github.com/w8r/Leaflet.draw.drag
[25] http://momentjs.com/

Since NPM packages must normally be used with a Node.js server, the NPM package "browserify"[26] is used. As a reminder, this package allows the developer to use NPM's packages within a simple JavaScript's client application. As soon as the */public/javascripts/search.js* file is updated, the developer has to type the following command into a console to generate the */public/javascripts/search-bundle.js* file.

```
browserify search.js -o search-bundle.js
```

This *search-bundle.js* file contains a working version of the *search.js* file, with all the code of the NPM's packages included in it. This file is thus the one that is read by the web browser

2. **External JavaScript libraries**:
   - **jQuery**[27]: provides an easier way to manipulate DOM elements than simple JavaScript code.
   - **Bootstrap**[28]: provides HTML components, CSS classes and JavaScript code to easily develop responsive interfaces.
   - **shapefile-js**[29]: the *calvinmetcalf*'s library provides a tool to read Shapefile-formatted[30] files and parse them in json.
   - **pickadate.js**[31]: provides JavaScript date and time pickers.
   - **jquery.fileDownload**[32]: the *johnculviner*'s library provides an easy way to download file with Ajax requests.
   - **jQuery-File-Upload**[33]: the *blueimp*'s library is used to easily upload files with Ajax and to show a progress-bar.

Several of these libraries were already encountered previously in this document; the rest of them will be explained in the following chapters.

Dates Format

Since a lot of dates are used in GeoTwit (in the static mode's search page, and in the results pages of both modes) and since they have to be regularly formatted, validated and converted to string objects, the "Moment.js" library is used to simplify this process. This well-known library allows the developer to easily work with dates of both *Date* and *String* types.

[26] http://browserify.org/
[27] https://jquery.com/
[28] http://getbootstrap.com/
[29] https://github.com/calvinmetcalf/shapefile-js
[30] https://en.wikipedia.org/wiki/Shapefile
[31] http://amsul.ca/pickadate.js/
[32] https://github.com/johnculviner/jquery.fileDownload
[33] https://github.com/blueimp/jQuery-File-Upload

Building of the Twitter's Query String

In order to filter tweets with the Twitter API, the developer has to give a query string. It is possible to add AND and OR filter's operator by respectively separating the words with spaces (" ") and commas (",") for the streaming mode or with the " OR " and " AND " strings for the static mode, since parentheses are not supported by the API. Note that the AND keyword takes priority over the OR one, since comma-separated terms are considered as phrases by the API; here are some examples with in streaming mode (the method is the same for static mode):

- $job \rightarrow$ interpreted as "job"
- $jobengineer \rightarrow$ job AND engineer
- $job, engineer \rightarrow$ job OR engineer
- $jobengineer, nursing \rightarrow$ (job AND engineer) OR nursing
- $job, engineernursing \rightarrow$ job OR (engineer AND nursing)
- $job, engineernursing, potato \rightarrow$ job OR (engineer AND nursing) OR potato

Once the user filled the keywords sets (with spaces between words) and pressed the search button, the "getAndFormatKeywords(...)" function is called. Three cases are possible from there:

1. The user only filled the AND part ("All these keywords...") → there is nothing to do here since the words are already separated by spaces.
2. The user only filled the OR part ("One of these keywords...") → the algorithm replaces the spaces with commas.
3. The user filled the AND and the OR parts → since the AND takes priority over the OR and since we want to search tweets that contain all the AND words and one of more of the OR words (for example: "(dog AND cat) AND (eat OR drink)", the algorithm has to manually build the string, by taking each of the OR words and by adding them one by one at the end of the AND words, separating all the possibilities with commas. For example, if the user entered "**dog cat**" in the AND field and "**eat drink**" in the OR field, the resulting query will be "**dog cat eat,dog cat drink**", which is interpreted as "(dog AND cat AND eat) OR (dog AND cat AND drink)".

According to the documentation, the text of the tweet and some entity fields are considered for matches. Specifically, the text attribute of the tweet, the "expanded_url" and "display_url" values for links and media, the text for hashtags, and the "screen_name" value for user mentions are checked for matches. Each phrase of the query (terms between commas, like "**dog cat eat**" and "**dog cat drink**") must contain up to 60 characters maximum, otherwise the API will generate an error. In all cases, the function also locally saves a human-comprehensive query (with "AND" and "OR" keywords, and parentheses), in order to display it on the results page. These queries are generated by parsing the query sent to the server.

Selection of the Language of the tweets

In GeoTwit, the user can choose a language with a drop-down menu, in order to filter the received tweets by their language.

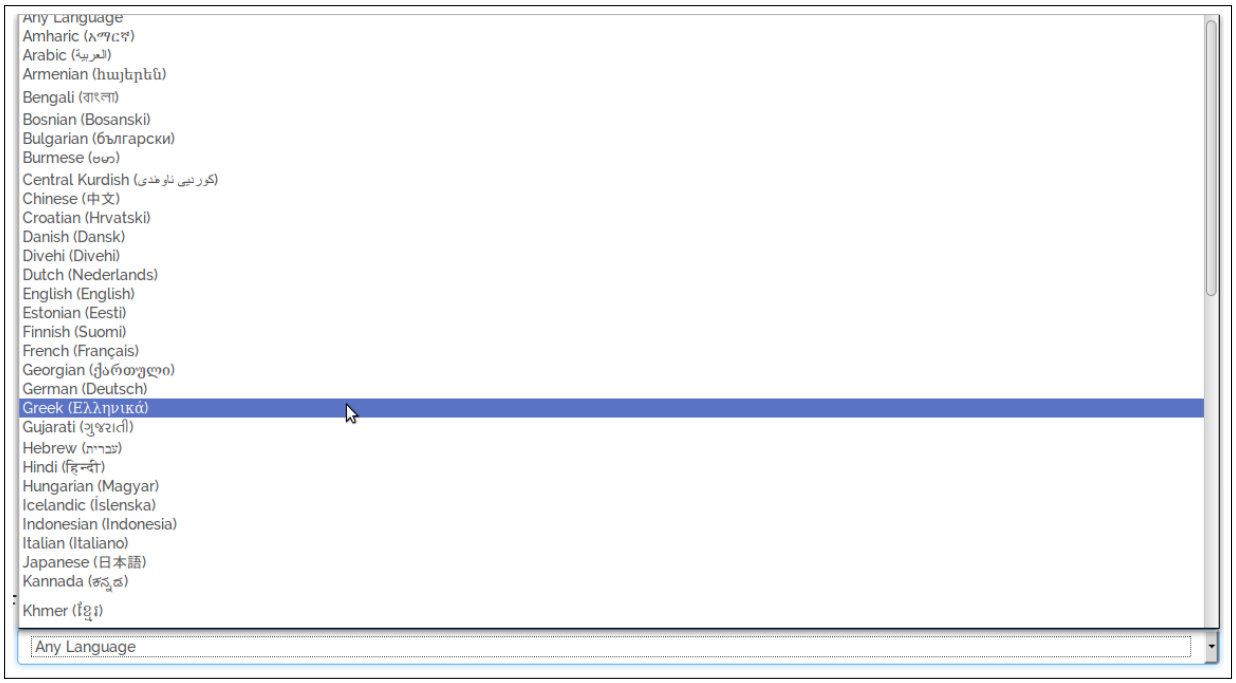


Figure 3.39: selection of a tweets' language in GeoTwit

Since this languages' drop-down menu is located both in the dynamic and static modes, and for scalability issues, the language list is stored in json format, located in the */public/data/languages.json* file. It is then loaded in the "loadLanguagesList()" function of the JavaScript file. The json file is composed of an array of languages, in which each element has "value" (the BCP 47 language identifier ("en", "fr", etc.) sent to the Twitter's API) and "text" (the displayed text) attributes.

The identification of a tweet's language[3] seems to be performed by keywords analysis. However, since tweets' content can be very short, it may be difficult to properly determine the true language, so errors can be possible:

Text	Language	Explanation
Justin Bieber <3	und (Undefined)	NOT English; contains only a name.
Schalke XI v Chelsea: Fahrmann, Neustadter, Santana, Howedes, Uchida, Fuchs, Kirchhoff, Boateng, Hoger, Choupo-Moting, Huntelaar.	und (Undefined)	Contains only place/team/player names.
Ate spaghetti at La tratoria napolitana	en (English)	The name of the restaurant is in Italian, but the "main" language is English. An English-only speaker would understand this Tweet.
#NowListening Universo - Lodovica Comello @XYZ @XYZ	und (Undefined)	Italian song title and artist are just names. #NowListening is English but could be used by non-English speaker too.
#My #hot #naughty #neighbour #in #dallas: http://t.co/0dLJ 北京	en (English)	There is a Chinese word at the end, but the strongly prevailing language is English
Hahaha (•_•) (•_•)>⌐■-■ (⌐■_■) YEAHHH!	und (Undefined)	Emoticons and interjections only.
Que bonito!	und (Undefined)	Could be both Spanish and Portuguese
Pozor pozor	und (Undefined)	Could be Czech, Serbian, Croatian, Slovenian, ...
So warm in Berlin!	und (Undefined)	A valid sentence in both German and English
"Last Christmas" - Der Jose Carreras unter den Weihnachtsliedern.	de (German)	Contains an English song title and Spanish name, but is understandable to a German-only speaker.
Bécs <3	hu (Hungarian)	This is the Hungarian name for "Vienna", which is a proper name, but exists only in Hungarian
Estoy muy cansado voy a acostarme .... sooo tired goin to bedd	und (Undefined)	Strong mixture of Spanish and English, no clear "main" language

Figure 3.40: example of difficulties that the language-identification algorithm can encounter[3]

The algorithm also seems to learn from human-identified languages with sets of tweets.

Selection of the Area(s) in the Search Page

In both the streaming and the static modes of the application, the user has to choose a location and the area with-in which the tweets will be searched. In the streaming mode, he has the choice to either select a default area in a given list of countries or draw a rectangle area by dragging-and-dropping the cursor on the map. In the static mode, the user can only select a circle area by drag-and-dropping the cursor, since the Twitter's Rest API asks for a circle (it would be tricky and computationally expensive to calculate a bounding circle of the complex polygon representing a country).

Selection of a default area in the list of countries
As a reminder (see the prototype application's "Leaflet - Countries' Borders" chapter above), the country the user selects in the drop-down menu contains a value corresponding to the English name of the country. As soon as a country is selected, the JavaScript code of the "selectCountryOnMap(...)" function reads and converts to Json the "TM_WORLD_BORDERS-0.3" Shapefile[34], which contains all the borders of almost each country of the worlds (note that this file is also read to add the country list in the drop-down menu at the loading of the page, in the "loadCountriesList()" function). Then, the algorithm seeks for the selected country in the Json-converted data, and uses the related coordinates to draw the polygon and the map. The client also sends the server the coordinates of a rectangle bounding the selected country's territories (since a country can have many, like French territories of France and Corsica) as said before (see the "readyToStream" web socket's type). The coordinates of this bounding rectangle are calculated in the "selectCountryOnMap(...)" function: if the country only has one territory (like Switzerland), the algorithm just selects the maximum and minimum latitude and longitude points of the polygon representing the country; if it has many territories, the algorithm selects the maximum and minimum coordinates of all different territories. The server will then search for tweets located inside this rectangle, and send them to the client.

Manual selection of a rectangle/circle
The code of this solution is contained in the */public/javascripts/search.js* file, specifically in the "loadSearchMap()" function. In order to be able to draw a rectangle/circle on the map with drag-and-drop, the *w8r*'s "Leaflet.draw.drag" library[35] is used. This library further uses the Leaflet's "Leaflet.draw" library[36], which adds support for drawing and editing polygons on Leaflet maps. Note that these libraries are NPM packages, and are thus used with the "browserify" package.

In the code, right after the map's initialization, three objects are set: the first one will contain the drawn items, and the two last ones will contain draw control's objects:

1. *drawnItems*: this object is added to the map as a feature group and will contain the drawn rectangle/circle.

2. *drawnControlFull[mapId]*, where "mapId" is the ID (either "streaming" or "static") of the current results' map: this object represents a toolbar that allows the user to draw a rectangle/circle on the map and which is located at the top-left corner of the map element. It is the default drawing that appears when there is no polygon drawn, and disappears as soon as the user draws one.

Figure 3.41: drawing of a rectangle on the dynamic map

[34]https://en.wikipedia.org/wiki/Shapefile
[35]https://github.com/w8r/Leaflet.draw.drag
[36]https://github.com/Leaflet/Leaflet.draw

3. *drawnControlEditOnly[mapId]*: this object represents another toolbar that allows the user to update the drawn rectangle/circle. It only appears when the polygon has been drawn.

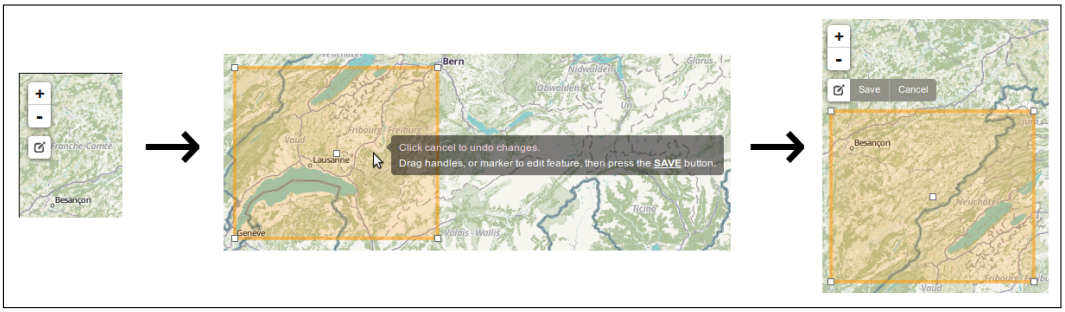


Figure 3.42: addition of the rectangle on the dynamic map

As soon as the user draws or edits a rectangle or a circle, the new coordinates are saved in the "boundingRectangleLatLngs" object (the latitude and longitude coordinates of each corner) or in the "circleLatLngRad" (the latitude and longitude of the circle's center' point and the radius in kilometers) object, which will later be sent to the server.

Grouping of the tweets on the Map

Since there can be a lot of displayed tweets on the map, they have to be grouped by clusters. Fortunately, the Leaflet library offers a plug-in for its library, called "Leaflet.markercluster"[37], that allows the developer to easily group markers on a map by creating cluster groups.

In GeoTwit, only one cluster group is created (at the initialization of the results maps, in the "loadResultsMap" function):

```
// Adds the cluster goup object as a layer to the map of the streaming's results,
// in order to automatically group markers with the Leaflet.markercluster library.
// The cluster group is disabled from a zoom level of 6.
markers = L.markerClusterGroup({
    disableClusteringAtZoom: 6,
    maxClusterRadius: 50
});
resultMaps[mode].addLayer(markers);
```

Figure 3.43: creation of the cluster group in GeoTwit

The developer then just has to add a marker in this cluster group in order to have the library automatically manage the grouping.

Once a new tweet is received, the client adds it to the cluster group if it belongs to the area; if it is not in the area, the client just ignores it.

```
// Adds the new Tweet on the map.
// The Leaflet.markercluster library will automatically group Tweets on the map.
markers.addLayer(L.marker([data.latitude, data.longitude], {icon: markersIcons[data.keywordsSet]}));
```

Figure 3.44: adding of a tweet in the cluster group, in GeoTwit

[37] https://github.com/Leaflet/Leaflet.markercluster

3.4.2 Streaming Mode

Interface of the Streaming Process' Results

During the streaming process, the user can access plenty of information, including:

- The elapsed time since the beginning of the streaming process, updated in the "speedInterval" timer object, which is itself initialized in the "loadStreamingResultsComponents()" function called when the client received the "successfulInit" web socket type.
- The human-comprehensive queries (for example "dog AND (eat OR drink)"), generated in the "getAndFormatKeywords(...)" function (see the "Building of the Twitter's Query String" chapter) when the client sends the user parameters to the server.
- The map displaying all the received tweets located in the selected area (see the "Reception of tweets from the Server" chapter below).
- Various charts allowing the user to better analyze the current streaming process and the data related to it (see the "Charts" chapter below).
- The content of the last 100 received tweets; this number can be changed with the "MAX_DISPLAYED_T constant variable.
- The total number of received tweets with geolocation tags, which correspond to the "nbReceivedtweets" variable.
- The average speed of received tweets per minute, which is calculated each second by the "speedInterval" timer object (just like the elapsed time information). The calculation is very simple and just divides the number of received tweets by the elapsed time, then multiplies it by 60 (in order to obtain minutes) and rounds it to two decimals. The tricky part of the algorithm is that it also calculates the color value of the speed text (which is green when the current speed is good, and red when it is not - the quality of the speed being determined by the "GOOD_SPEED" constant value). In fact, the algorithm calculates the hue (between "0 → red" and "120 → green") and the lightness values (between 50% for the bad speed and 25% for the good speed) of an HSL color[38], by the calculated speed per minute.

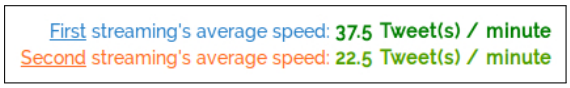


Figure 3.45: first example of speed's color

Figure 3.46: second example of color

The "speedInterval" timer is stopped when the user clicks on the "Stop Streaming" button.

[38] http://www.w3schools.com/colors/colors_hsl.asp

If the user filled in two keywords sets, the data related to the first one (human-comprehensive query, markers on the map, curves of the charts, number of received tweets, average speed's label, content of the tweets, etc.) will appear in blue, while the ones related to the second one will appear in orange. Be careful not to confuse the subject's colors with the cluster groups' colors, which appear in green when there are not a lot of tweets in the group, and in red when there are a lot of tweets.

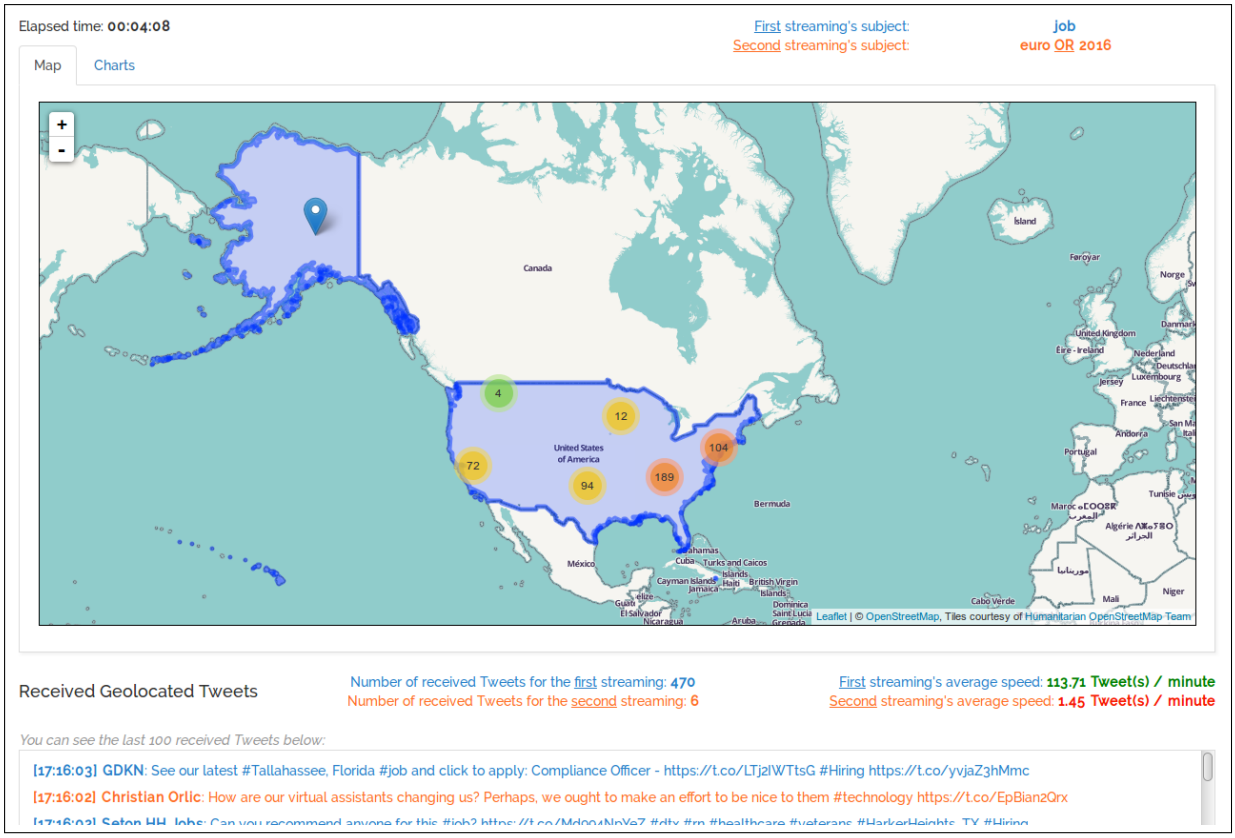


Figure 3.47: example of an interface of the streaming process' results

Charts are properly explained in the "Charts" chapter below.

Reception of tweets from the Server

Once the streaming process begins, the client first draws the saved selected area (either the rectangle or the complex polygon bounding the area or country). When the server receives a tweet matching the filter, the language and the bounding rectangle, it is sent to the client. Once the client receives the "newtweet" web socket message, it first saves the total number of received tweets (both with or without geolocation) contained in the web socket's content, then acts according to the selected location's type (determined by a Boolean variable set when the user selected the area):

- If the user manually selects a rectangular area on the map, it means that the received tweet is certainly in the selected area, so the client just displays it on the map and in the received tweets panel (with the "addtweetOnPage(...)" function). It also increments the local number of received tweets, in order to properly calculate the average reception speed.
- It is more complicated if the user selects a default area/country from the drop-down menu: since the server searches for tweets within the bounding rectangle, some of the received tweets may not be in the selected area.

Figure 3.48: example of a tweet located in the bounding box of the selected country's territories, but not in the territories themselves

In this case, the client must check that the received tweet belongs to the complex polygon. In order to properly accomplish this task, the Point Inclusion in Polygon algorithm (*PNPOLY*)[2] was used with an updated version of the *substack*'s "point-in-polygon" library found on GitHub[39], which allows the developer to check if a given point is inside a given polygon. Since the *PNPOLY* algorithm is well explained on the given web site, it will be briefly laid out here. The important points here are: according to the documentation, the algorithm runs a semi-infinite ray horizontally (increasing x, fixed y) out from the test point, and counts how many edges it crosses. At each crossing, the ray switches between inside and outside. This is called the Jordan curve theorem. The case of the ray going through a vertex is handled correctly via a careful selection of inequalities. The ray is tested against each edge to determine if the point is in the half-plane to the left of the extended edge and if the point's Y coordinate is within the edge's Y-range.

[39]https://github.com/substack/point-in-polygon/blob/master/index.js

Charts

In order to draw charts in GeoTwit, the Chart.js library is used[40]. There are several other well-known libraries (like D3.js[41]), but they are either too complicated for the simple use-case of this application or not as user-friendly and well documented as Chart.js. In GeoTwit, there are six possible charts, ordered in two categories: tweets having geolocation tags, and all tweets (both with or without geolocation tags). The following screenshots were taken during an example test made with the "job" (blue) and "euro OR 2016" (orange) keywords sets and located in the U.S.A.

1. Charts related to **tweets having geolocation tags**:

   (a) **Total of received tweets, by time**: this "line" chart displays the total of received tweets that have a geolocation tag, first by seconds, and then by minutes once 60 seconds have elapsed.

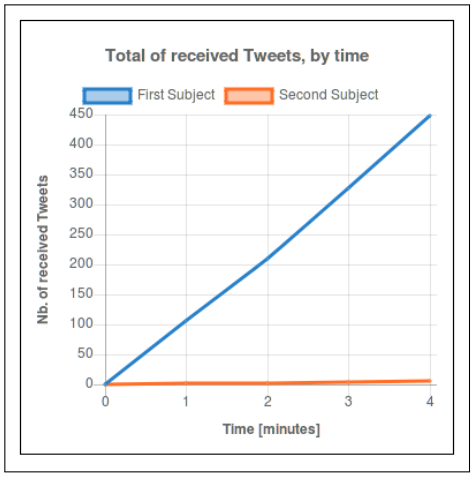


Figure 3.49: example of the "GTRT" chart

   (b) **Reception of tweets, by time**: this "line" chart displays the number of received tweets during time periods (first by seconds, and then by minutes once 60 seconds have elapsed).

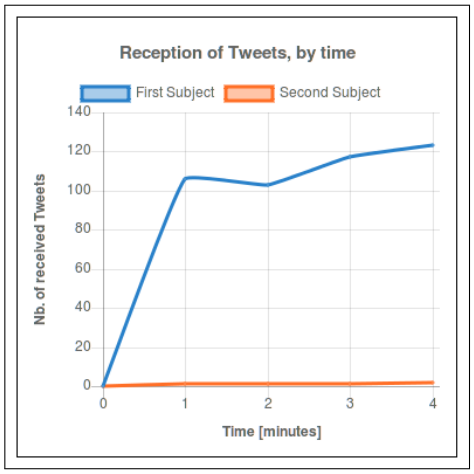


Figure 3.50: example of the "GRT" chart

[40] http://www.chartjs.org/
[41] https://d3js.org/

(c) **Parts of the received tweets by subject**: this "doughnut" chart compares the part of received tweets for each subject, and is only visible when the user has filled both the keywords sets.

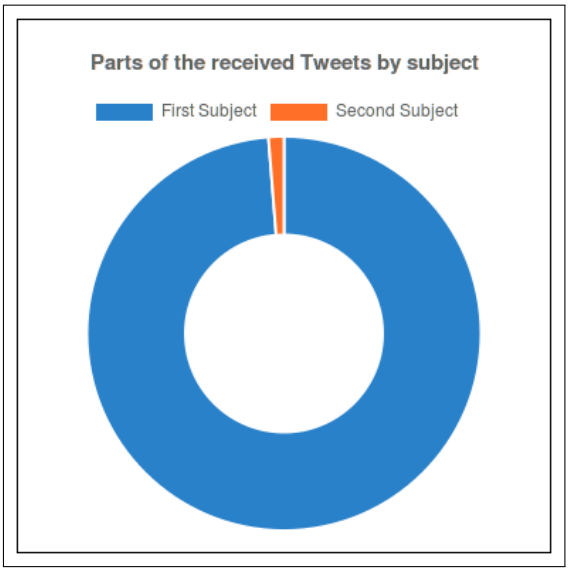


Figure 3.51: example of the "GPRT" chart

2. Charts related to all kinds of tweets (**with and <u>without</u> geolocation tags**):

(a) **Total of received tweets, by time**: this "line" chart displays the total of all received tweets (that do or do not have a geolocation tag), first by seconds, and then by minutes once 60 seconds have elapsed.

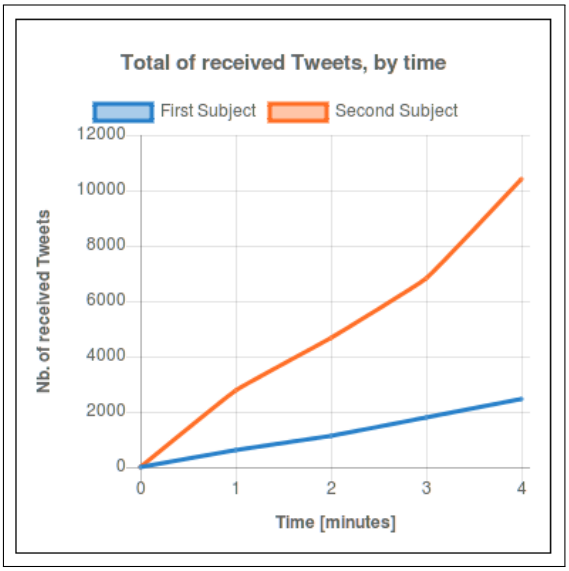


Figure 3.52: example of the "ATRT" chart

(b) **Reception of tweets, by time**: this "line" chart displays the number of all received tweets (that do or do not have a geolocation tag) during time periods (first by seconds, and then by minutes once 60 seconds have elapsed).

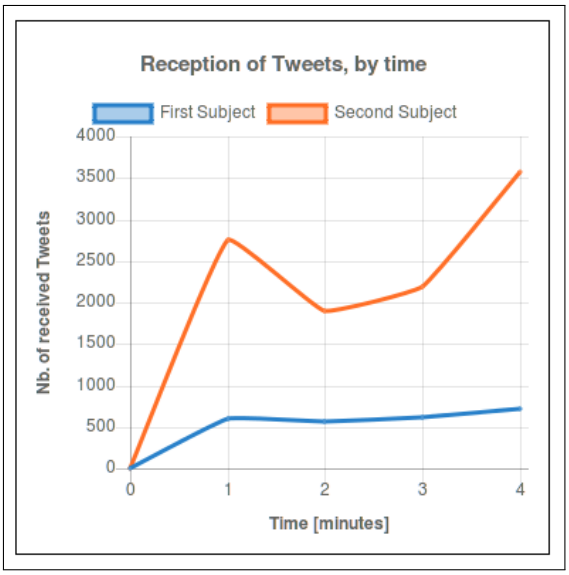


Figure 3.53: example of the "ART" chart

(c) **tweets with geolocation vs. tweets without**: this "bar" chart displays the percentage of tweets that have geolocation tags versus the percentage of those which do not have one.

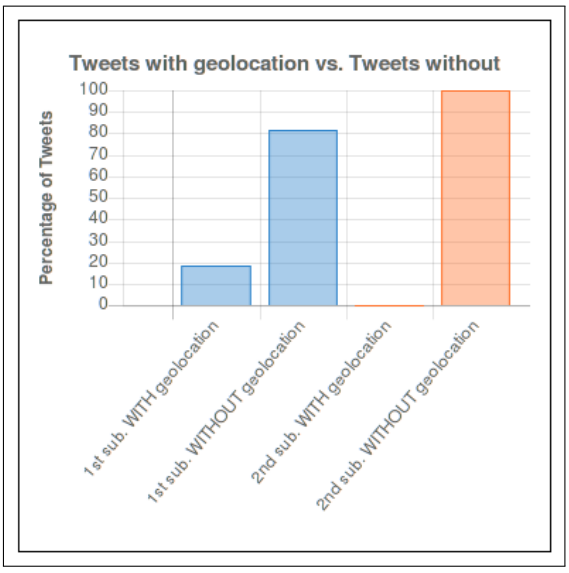


Figure 3.54: example of the "AGVW" chart

All these charts are responsive and react when the user moves the mouse over their elements.

Since the library is well documented, everything will not be explained here; the initialization of the charts and the timings of refreshment are located in the "loadStreamingResultsCharts(...)" function. Firstly, the contexts of the charts (the HTML elements) are initialized, then the empty datasets of the charts and the charts themselves. The interesting parts of the algorithm concern the refreshment of the charts; there are two timers refreshing the charts each second (and then each minute for the first one):

1. **lineChartsUpdateInterval**: this timer refreshes all the "line" charts (so 4 on the 6 charts) each second, and then each minute when 60 seconds have elapsed. All the data is evaluated with the "nbReceivedtweets" and "lastNbReceivedtweets" objects, which respectively contain the total number of received tweets (for the tweets with geolocation and for all the tweets) and the last count of received tweets that were calculated during the last tick of the timer.

2. **doughnutBarChartsUpdateInterval**: this timer refreshes the "doughnut" and "bar" charts each second. All the data is evaluated using the "nbReceivedtweets" object. The "bar" chart contains two datasets in case of multiple keywords sets (one for each keywords set), in order to display the two subjects' colors. Each data set contains 4 values (2 for the current subject and two null values for the other subject), in order to only display 4 bars (and not 8: *2 subjects x 4 bars*).

Once the charts are initialized, the timers are started. They are stopped when the user clicks on the "Stop Streaming" button.

3.4.3 Static Mode

Selection of the Dates

In the static mode, the user has to select the search's start and end dates in the *YYYY-MM-DD* format. Since the Twitter's REST API only allows the developers to get tweets from a maximum of 9 days past, the user cannot select dates incompatible with this condition.

In order to facilitate the selection of dates, the *amsul*'s date picker "pickadate.js" library[42] is used, with the "classic" theme. This library is located in the *public/javascripts* folder in the "picker.min.js", "picker.date.min.js" and the "legacy.js" (this last one is used for older versions of Internet Explorer) files.

The date picker is loaded on dates field as soon as the Search page loads successfully.

```
// Loads the date picker on the date fields.
$("#staticFromDate, #staticToDate").pickadate({
    "format": "yyyy-mm-dd",
    min: moment().subtract(MAX_DAYS_STATIC_SEARCH, 'd').toDate(),
    max: new Date()
});
```

Figure 3.55: loading of the date picker on the dates fields

[42] https://github.com/amsul/pickadate.js

Retrieval of tweets with the Server

Once the user has filled all the fields, the client sends an Ajax request to the server, in order to retrieve the results. This Ajax request is located in the "getStatictweets()"method of the *search.js* file. The following parameters are sent in a json format:

- **firstKeywords**: the first keyword set filled by the user, formatted with the "getAndFormatKeywords(...)" function (see the "Building of the Twitter's Query String" chapter above).
- **secondKeywords**: the second formatted keyword set filled by the user (it can be empty).
- **fromDate**, **toDate**: the *YYYY-MM-DD* formatted dates from and to where the tweets will be searched.
- **locationLat**, **locationLon**, **locationRad**: the latitude and longitude coordinates of the circle's middle point, and the radius of the circle in kilometers.
- **language**: the BCP 47 language identifier of the search's language.

Once the server responds to the Ajax request with json, the response message can contain either an error (in this case, the client just displays an error message) or the list of tweets. Since there can be a lot of tweets, the data transfer can take a while, so the search button changes its display:

Figure 3.56: display of the search button after the user has clicked on it in order to search for tweets

Once the tweets' lists have been received, the client first calculates the numbers of tweets, then concatenates the lists (since there is one list per keywords set), sorts the new-created list by the dates and times of the tweets, then adds each tweet one by one on the results map (only for the tweets having a geolocation tag) with the "addtweetOnPage(...)" function, and adds each tweet in the tweets panel.

The tweets are also grouped with cluster groups, like in the streaming mode's results (see the "Reception of tweets from the Server" chapter below).

Interface of the Static Mode's Results

Once the static search is done, the user can access plenty of information, including:

- The human-comprehensive queries (for example "dog AND (eat OR drink)"), generated in the "getAndFormatKeywords(...)" function (see the "Building of the Twitter's Query String" chapter) when the client sends the user parameters to the server.
- The map that displays all the tweets located in the selected area. Note that some tweets can be displayed outside the circle, due to the search algorithm of the Twitter's REST API, which is a little tricky.
- The content of all the tweets.
- The total number of received tweets, both with and without geolocation tags.

If the user filled the two keywords sets, the data related to the first one will appear in blue, while the ones related to the second one will appear in orange.

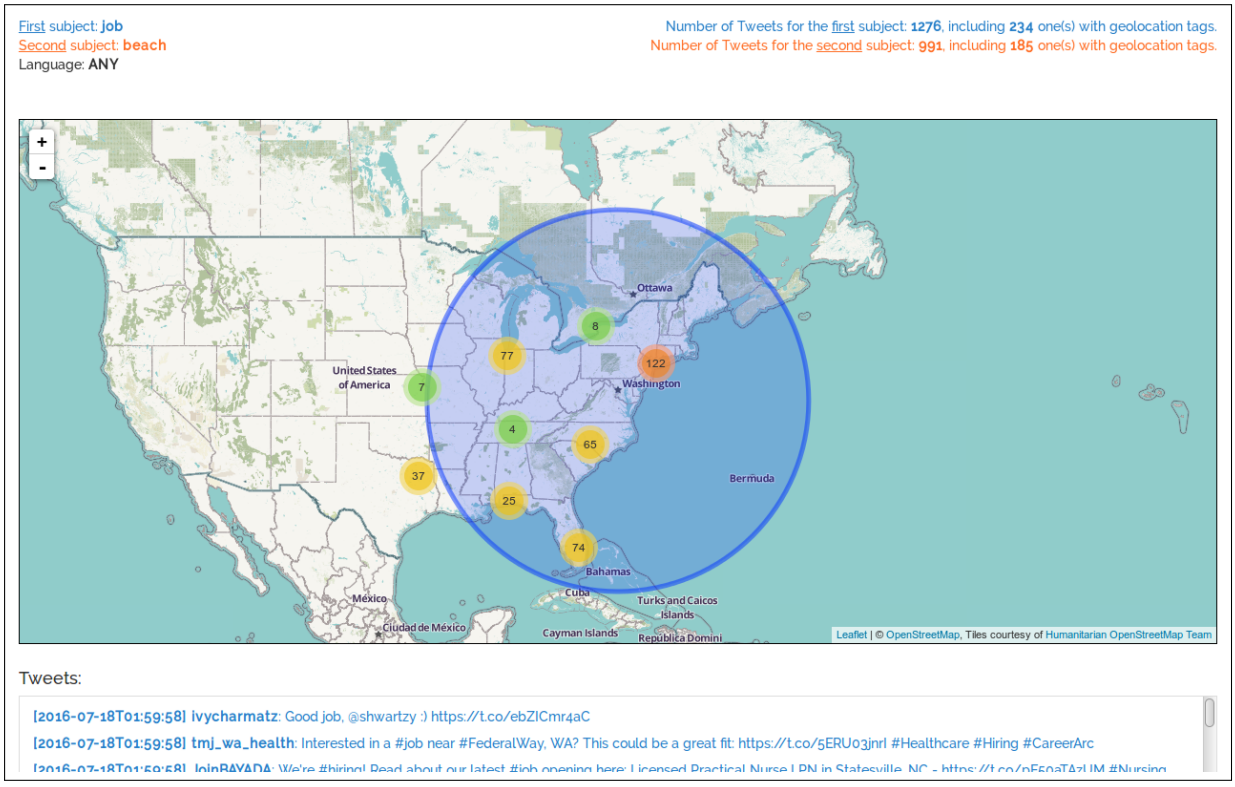


Figure 3.57: example of the interface of the static mode's results

Chapter 4

Tests

This chapter contains all the documentation related to the application's tests.
A Twitter account was specially created for this project and has been used for most of these tests:

- **Username**: GeoTwitHeig
- **Password**: @heigVdRocks42

Feel free to use these credentials for your own tests.

4.1 Functional Tests

All the development process and thus the functional tests were firstly processed with the version 47.0 of Firefox on Linux Mint. Though, other web browsers were tested in the "Web Browsers - Compatibilities" chapter above.

#	**Description**	**Results**
	Miscellaneous	
1	If the user tries to access a non-existent page, a "Not Found" page is displayed.	OK
2	The user can access the about and help pages, regardless of whether he is connected or not.	OK
3	The web site's GUI is stable.	OK
4	The web site's GUI is responsive.	Only for desktop applications
5	Maps are responsive and allow the user to zoom in and out and to move around the world by drag-and-dropping the content.	OK
	Home page and connection process	
6	The user can properly access the application's home page if he is disconnected.	OK

#	Description	Results
7	If the user is not connected and try to access pages different than the home page, he is redirected on the home page with an error message.	OK
8	The user can access the Twitter's authentication page by clicking either on the "Get Started" or on the "Connect" button.	OK
9	If the user denied the authentication process or if an error occurred, he is properly redirected on the home page with an explicit error message.	OK
10	Once the user successfully connected, he is able to navigate through the web site and to access the tools. He is however redirected to the search page if he tries to access the home page.	OK
11	If the connected user clicks on the "Disconnect" button, he is properly disconnected, the session is entirely destroyed and he cannot access search pages anymore.	OK
12	The session lasts 7 days and the user is not disconnected before this time limit, except if a modification was done in the server's code.	OK
Main Search Page		
13	The user is redirected on the main search page if he clicks on the GeoTwit's logo located on the header bar	OK
14	The user can navigate through both the dynamic and static modes with the tab system.	OK
Dynamic Mode - Files Importation		
15	By clicking on the importation button, a pop-up window that allows the user to select a file appears.	OK
16	If the user selects a wrong type of file (i.e. an image), an error is displayed.	OK
17	If the selected file is an empty text file, an error is displayed.	OK
18	If the selected file's content is not a valid GeoTwit content, an error is displayed.	OK
19	During the importation of the file, a progress-bar is displayed and is automatically refreshed.	OK
20	Once the importation was successfully done, the user is redirected on the result page and can access the information.	OK
21	The information displayed are the same as the ones contained in the file and thus the imported ones.	OK

#	Description	Results
	Dynamic Mode - Fields Validation	
22	The user cannot write more than 60 characters in the keywords sets' fields.	OK
22	If the user tries to bypass the characters' limitation by editing the text field's properties, the server still displays an error message.	OK
23	If the user did not set at least one keywords set when clicking on the "Start Streaming" button, an error message appears.	OK
24	The countries included in the default area list are listed in alphabetical order.	OK
25	If the user did not either choose a default area in the countries' list or manually draw a rectangle on the map, an error message is displayed.	OK
26	The user is able to draw only one rectangle on the map and is also able to update it.	OK
27	If a rectangle is drawn or it the user selected a country, all existing polygons are removed from the map.	OK
28	The languages list contains all the possible languages in alphabetic order.	OK
29	Once all fields are valid, the user is properly redirected to the results page.	OK
30	When the user clicks on the button and everything is valid, the server properly removes the current user's existing file-to-export in its side, if there was one.	OK
	Dynamic Mode - Results Page	
31	The results page's components are properly loaded once the user accessed it.	OK
32	If the user only filled one subject, only the blue components are displayed and enabled.	OK
33	If the user filled two subjects, the orange components also appears and are enabled.	OK
34	The displayed information correspond to the ones the user filled.	OK
35	The elapsed time's timer is enabled and working.	OK
36	The user can switch between the map and the charts by using the tab system of the page.	OK
37	Tweets are received in the selected location only.	OK

#	Description	Results
38	The average speed is correctly refreshed and changes its color according to the speed (green if good, red if bad).	OK
39	Once a tweet is received, it is correctly displayed on the map and in the tweets panel below with the right color, and the number of received tweet is updated.	OK
40	Only the last 100 received tweets are displayed in the tweets panel below	OK
41	Group clusters are properly working and disappear once the user zoomed at a zoom level of 8 or more.	OK
42	There are 5 available charts if the user entered only one subject, and 6 otherwise.	OK
43	The charts are correctly refreshed each second, then each minute once 60 seconds passed. In addition, they correctly differentiate the subjects if the user filled two subjects.	OK
44	The charts are decent, correct, and responsive.	OK
45	The streaming process is stopped once the user clicked on the "Stop Streaming" button, a new "Make a new search" button appears, and an exportation pop-up is instantaneously displayed.	OK
46	If the user either clicked the "Cancel" button of the pop-up window or closed it, the file is removed from the server side.	OK
47	If the user clicked the "Ok" button, he is able to save the file-to-export wherever he wants on his computer, then the file is removed from the server side. He is then able to export this file from the main search page.	OK
48	Once the user clicked on the "Make a new search" button, he is redirected on the main search page.	OK
Static Mode - Fields Validation		
49	The keywords sets fields act like the dynamic mode's ones.	OK
50	The user can only select dates between the current date and the date it was 9 days ago.	OK
51	If the dates are either not set or not valid, an error message is displayed.	OK
52	An error message is displayed if the user did not select a circle area on the map.	OK
53	The user is able to draw only one circle on the map and is able to update it.	OK
54	The languages list contains all the possible languages in alphabetic order.	OK

#	Description	Results
55	Once all the fields are valid, the user can search for results by clicking on the "View Results!" button.	OK
56	The button changes of behavior during the search process.	OK
57	If there was no result, a pop-up message is displayed.	OK
58	If the user exceeded the limit of requests made, a pop-up message is displayed.	OK
59	If there was no error, the user can access the results page.	OK
Static Mode - Results Page		
60	The results page act like the dynamic mode's result page, except that the user cannot access charts and everything is fixed.	OK
61	The tweet panel has no number limit and displays the tweets both with and without geolocation tags.	OK
62	Tweets are received in the selected location only.	Certain tweets are displayed outside the circle, but this issue is apparently due to the Twitter's algorithm, which is tricky.
Security		
63	If the user tries to bypass the fields' limitations (like the number of characters) by editing the text field's properties in order to bypass the client's error message, the server still displays an error message.	OK
64	All Ajax functions are properly protected on the server side (the user must be connected, the parameters are validated, etc.).	OK
65	The web socket server checks that the user is connected before starting the streaming process.	OK
66	The web socket server checks the identity of the client before accepting the discussion.	No

All tests thus successfully passed, except the #4 one (which was thus not specified in the specifications) and the #62 one (due to the Twitter's algorithms), which partially passed. Since the security of the application was not a main point, basic security issues are covered, but more complex ones like the #66 test was not developed.

4.1.1 Web Browsers - Compatibilities

As mentioned above, all the development process and thus the functional tests were processed with the version 47.0 of Firefox on Linux Mint.

Since the end users can have different operating systems and/or web browsers, the functional tests' list was also tested with a variety of combinations. Here are the obtained results:

#	**Operating System**	**Web Browser**	**Results**
1	Linux Mint	Mozilla Firefox 47	OK
2	Linux Mint	Chromium 51	Some components' borders are missing.
3	Windows7	Mozilla Firefox 47	OK
4	Windows7	Google Chrome 51	Some components' borders are missing.
5	Windows7	Internet Explorer 11	Certain components (like charts) are either too big or too small; some little bugs of interface.

In summary, all other recent web browsers seem to be able to run the application (sometimes with small bugs of interface, but nothing to worry about).

In order to make Internet Explorer work with the application, the default values of the parameters of the JavaScript functions had to be removed, since this web browser does not properly follow the JavaScript's ES6/ES2015 norms. Also, in order to make the file importation work with this browser, the "uploadAndParse" action of the **SearchController** now also validates the "text/plain" file's content type (used by IE) in addition to the basic "application/octet-stream" used by all other browsers.

4.2 Load Tests

Two load tests' types were analyzed:

1. **Tests about the number of received tweets**: these tests are used to analyze the application's reaction when there is a huge amount of data to process, display and/or import.

   The static mode is easy to test here: indeed, since the number of requests is limited to 170 (about 17'000 tweets) by the application's server, we just have to find a subject that contains a number of tweets big enough to reach this limit. The "job" keyword in the U.S.A. always gave good results over the other tests, and was thus used here. Firstly, here are the obtained results:

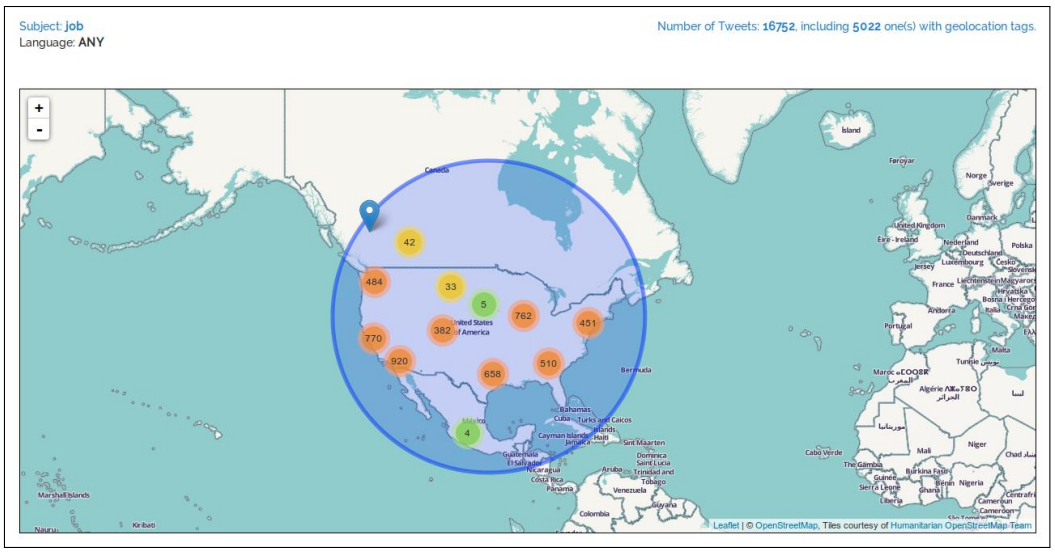


Figure 4.1: results of the load tests for the static mode with the "job" keyword

As expected, about 17'000 tweets were received, including about 5'000 ones with geolocation tags. The search process was timed and took 1 minute and 54 seconds. The map was very fluid since the tweets were grouped by cluster groups; however, when the map was zoomed enough to disable the cluster groups (zoom level of 8 or more) in areas with high tweets densities (like the West Coast), the loading took about 10 seconds, but the map was still fluid after the process. When the groups were enabled again by zooming out, the loading took about 2 seconds. This load test was then a success.

Concerning the dynamic mode, the streaming process had to run for hours in order to properly accumulate a charge wide enough. In order to get a huge amount of tweets, GeoTwit listened for the "job" and "beach" keywords sets in the U.S.A. (since there is a huge amount of tweets posted with geolocation tags for these subjects in general) for 13 hours and 30 minutes between 21:00 on 26.07.2016 and 10:30 on 27.07.2016. Firstly, here are the obtained results:

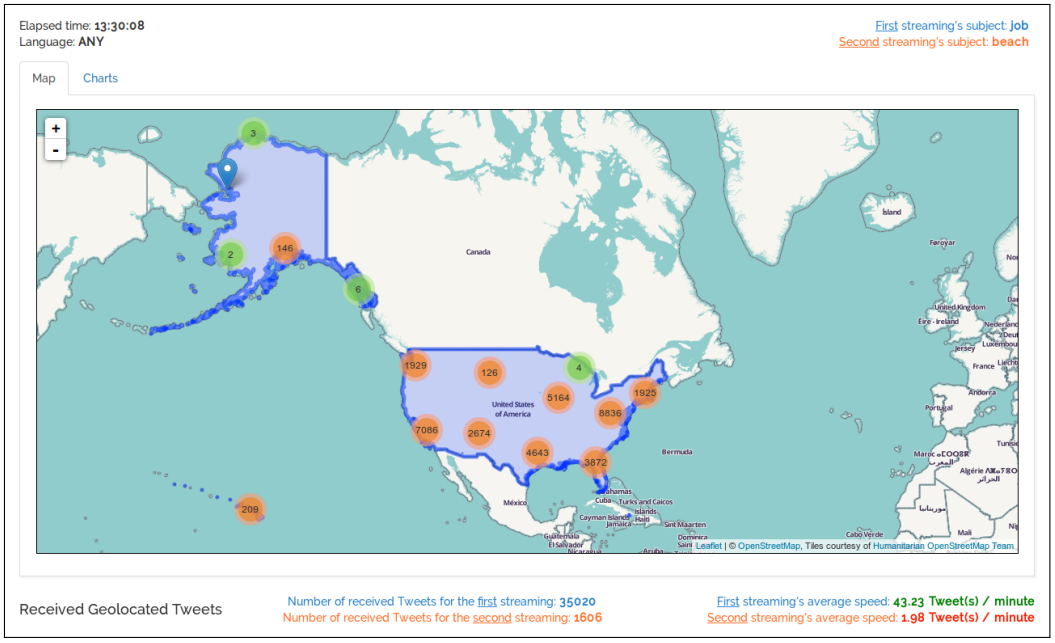


Figure 4.2: results' map of the streaming mode's load tests with the "job" and "beach" keywords

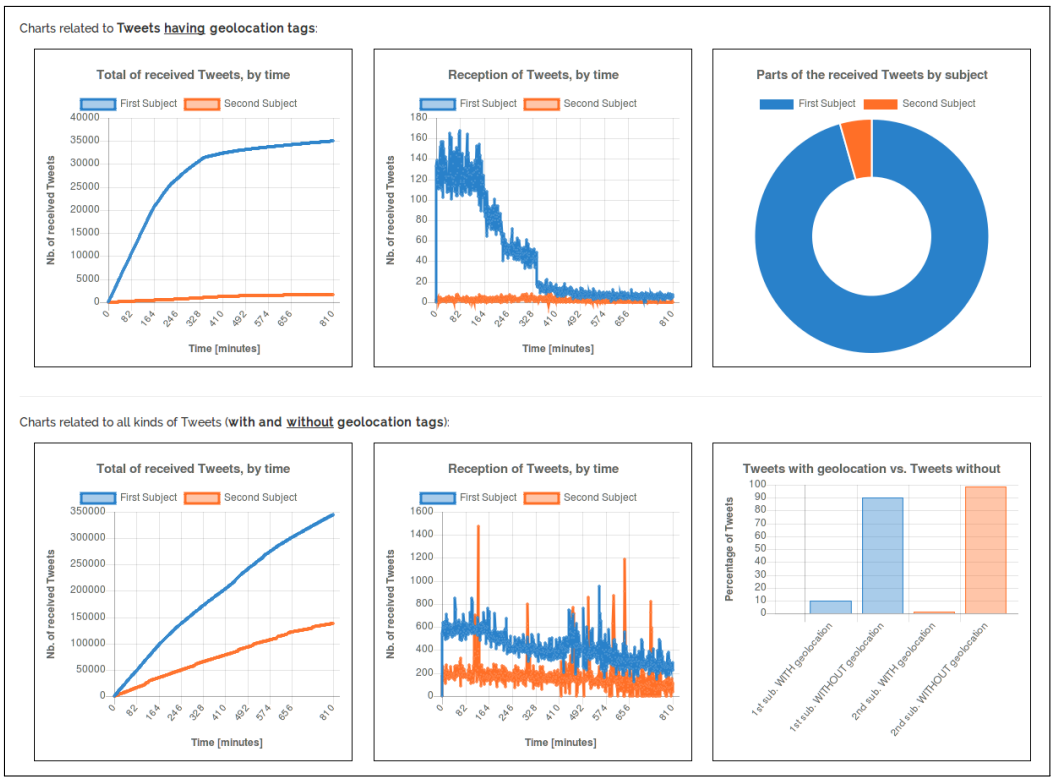


Figure 4.3: charts of the streaming mode's load tests with the "job" and "beach" keywords

The first comment we can do is about the geolocated reception, whose form is pretty logarithmic. This can be explained by the fact that it was the evening (18:00) in the U.S.A. approximately since the 180th minute, so there were less people/bots posting for jobs.

Concerning the load tests' observations, the page was still very fluid: it was possible to navigate the map and to interact with the charts without having any slowness, thanks to the cluster groups. However, zooming the map at a zoom level of 8 or more (which thus removes the cluster groups) on the area with the highest density took some time (about 40 seconds) to load the markers; once they were properly loaded, some little slowness began to appear, but the map was still well usable, though.

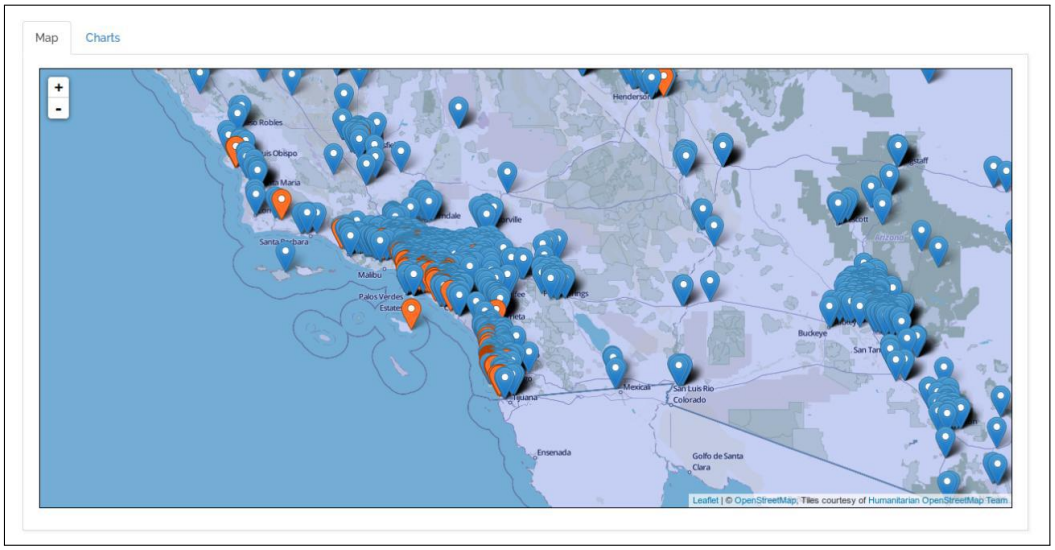


Figure 4.4: map of the load tests' results zoomed at a zoom level of 8

At the end of the process, the backup file was easily exported. The importation process took about 1 minute and 42 seconds (during which the overloaded web browser displayed some "script not answering" pop-up messages), but was successful in the end. Once loaded, the reaction of the file results' map was exactly the same as the previous map. This load test was also successful, then.

2. **Tests about the number of applications that simultaneously search for tweets**: since several Twitter accounts are required for properly performing these tests and since it would be annoying to create these account and connect with them, only two accounts were used, but with 4 different browsers in two different machines. The processes perfectly worked and there was not even a single lag.

   Of course, the number of applications was not wide enough to hope to get interesting reactions, but it though confirms that the server can easily manage several clients at the same time.

4.3 Subjects Analyzed with the Streaming Mode

4.3.1 The ”job” keyword

When listening to the ”job” keyword in the territories of the U.S.A., there was always a great tweets reception (about 130 geolocated tweets per minute) at any time of the day (in the U.S.A., so during the European afternoon and evening). Since the majority of these tweets always contained the same texts (”Want to work at...?”, ”We’re hiring!”, ”Can you recommend anyone for this job?”, ”This job might be a great fit for you”, ”If you’re looking for a job”, etc.) regardless to their locations (which were almost everywhere in the territories), it was concluded that they were posted by bots per time interval. A way to guarantee the veracity of this assumption is to look at the charts generated for this keyword, which are linear in time during the day:

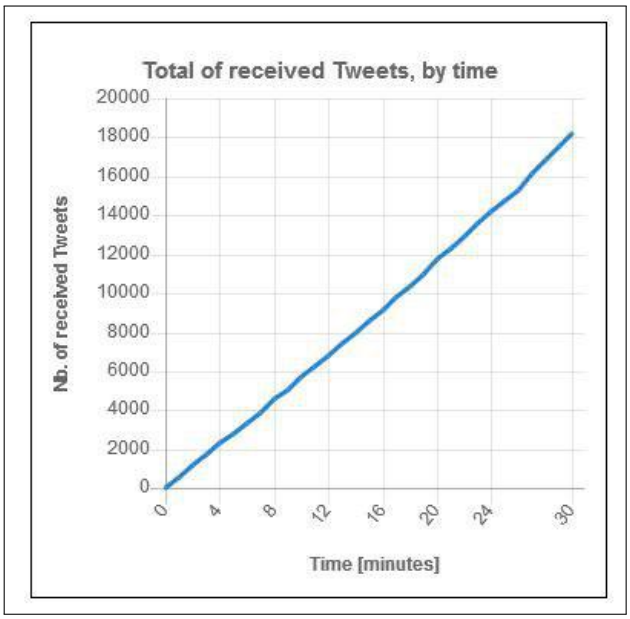


Figure 4.5: total number of received tweets by time, for the ”job” subject in the U.S.A.

This situation was very interesting throughout the application’s implementation process, because it assured a way to always have data to test.

4.3.2 Subjects by Countries

This section contains analyses separated by countries. The majority of them is related to the subjects already analyzed by the prototype application in the table of the "Interesting Twitter's Subjects" chapter, but during hours instead of minutes.

United Kingdom

Both the "bbc" and the "london" subjects were first analyzed here at the same time, on 18.08.2016 between 15:00 and 17:00.

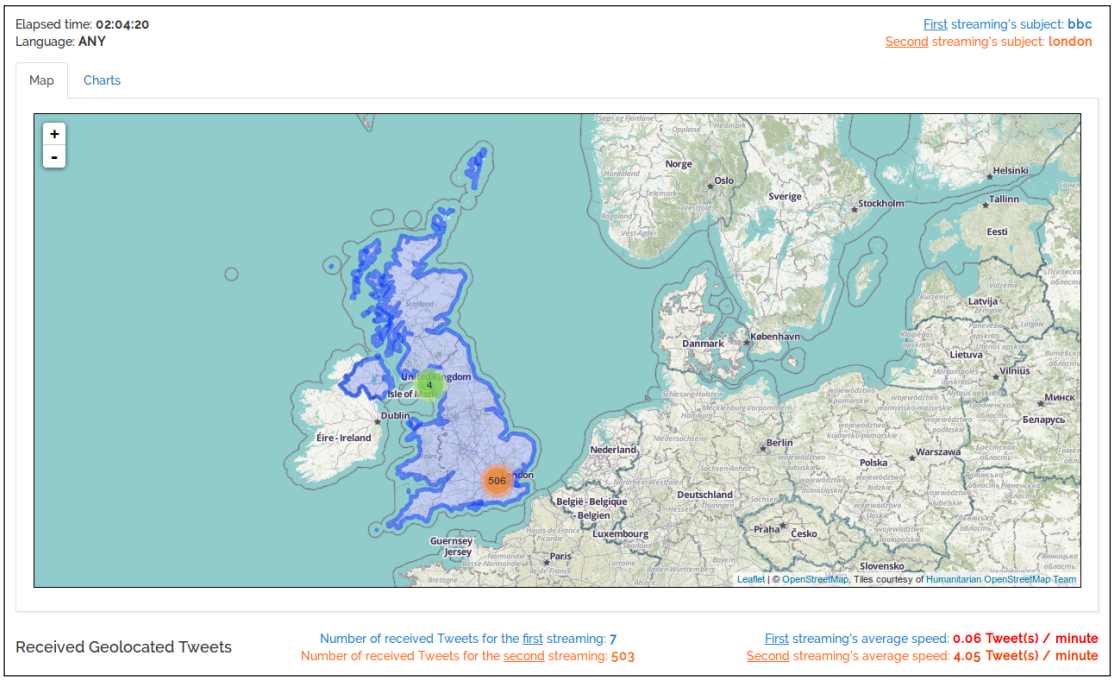


Figure 4.6: analysis' map of the "bbc" and "london" keywords in UK

Figure 4.7: analysis' charts of the "bbc" and "london" keywords in UK

The "bbc" subject gave bad results for the geolocated tweets, with an average speed reception of only 0.06 tweet per minute (99.97% of the tweets did not have geolocation data). The "london" subject was better, although not amazing either (here, 97.84% did not have geolocation tag). It is interesting to note than the reception of all kinds of tweets of the first subject was in a "stair" shape, while the second subject's one was pretty linear, but in the end their values were very close. Thus, British people seem to be more apt to post tweets related to London than tweets related to the BBC.

July 19 was the hottest day of the Summer in UK. The idea of analyzing this subject came from the fact that people tend to like complaining about the weather on the social networks, in general. Indeed, by analyzing the "hot" or "sun" keywords, about 370 geolocated tweets and about 160'000 tweets of all kinds were posted between 13:00 and 16:00.

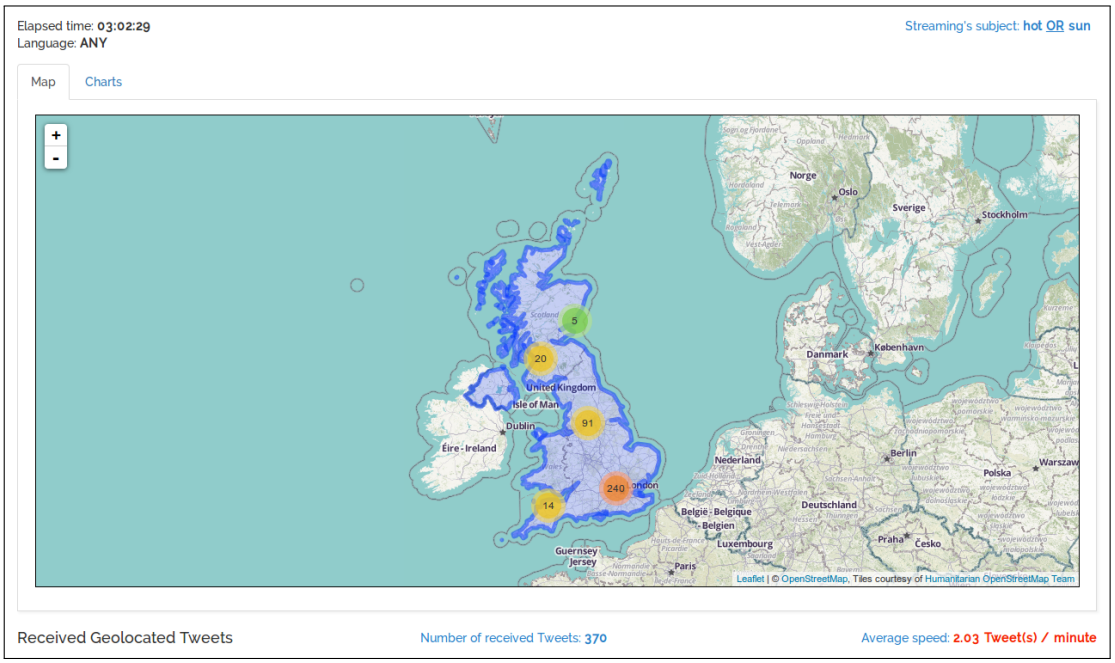


Figure 4.8: analysis' map of the "hot" or "sun" keywords in UK during the hottest day

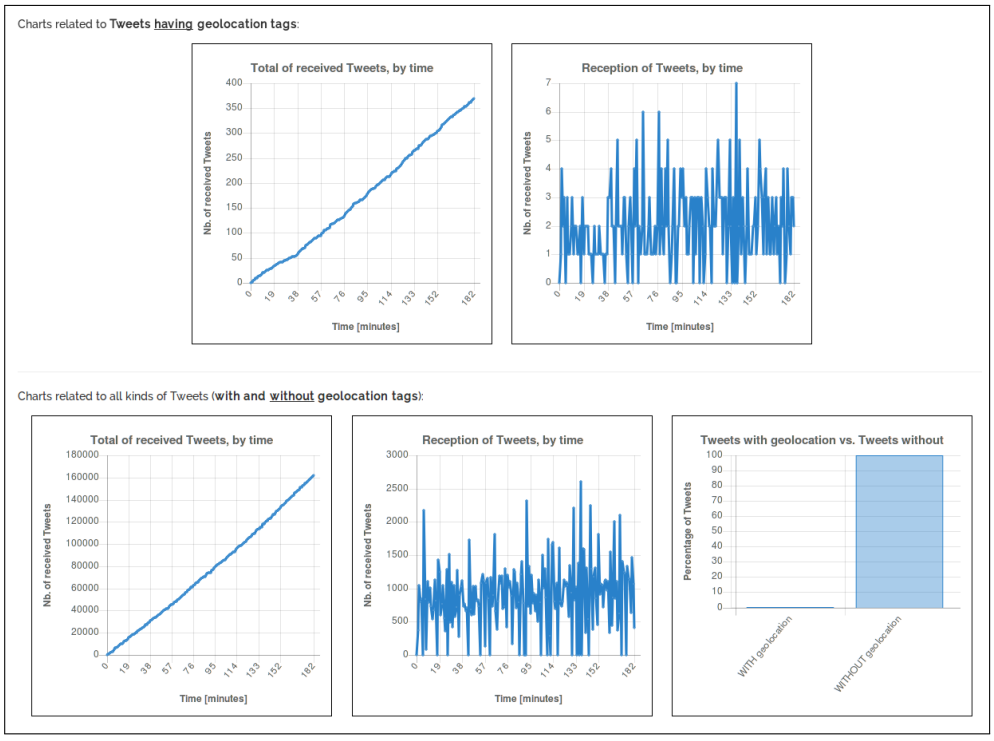


Figure 4.9: analysis' charts of the "hot" or "sun" keywords in UK during the hottest day

About 99.77% of the posted tweets did unfortunately not have geolocation data.

Italy

Both the ”pizza” and the ”roma” subjects were analyzed at the same time on 18.08.2016 between 17:00 and 19:00. Like in the prototype application's analysis, not a single tweet was received during the streaming process, so it was decided that no other test will be performed for this country.

France

The word ”paris” was tested on 19.07.2016 between 17:00 and 19:00.

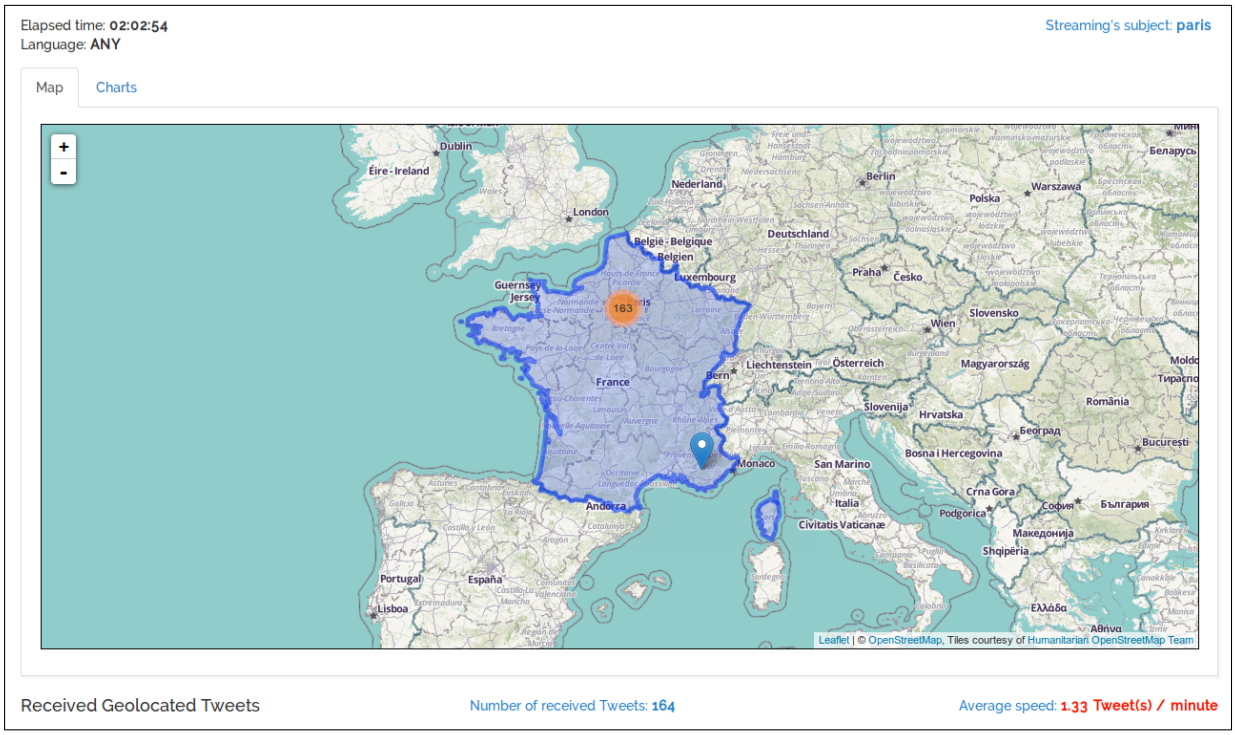


Figure 4.10: analysis' map of the ”paris” keyword in France

Here are the charts' results:

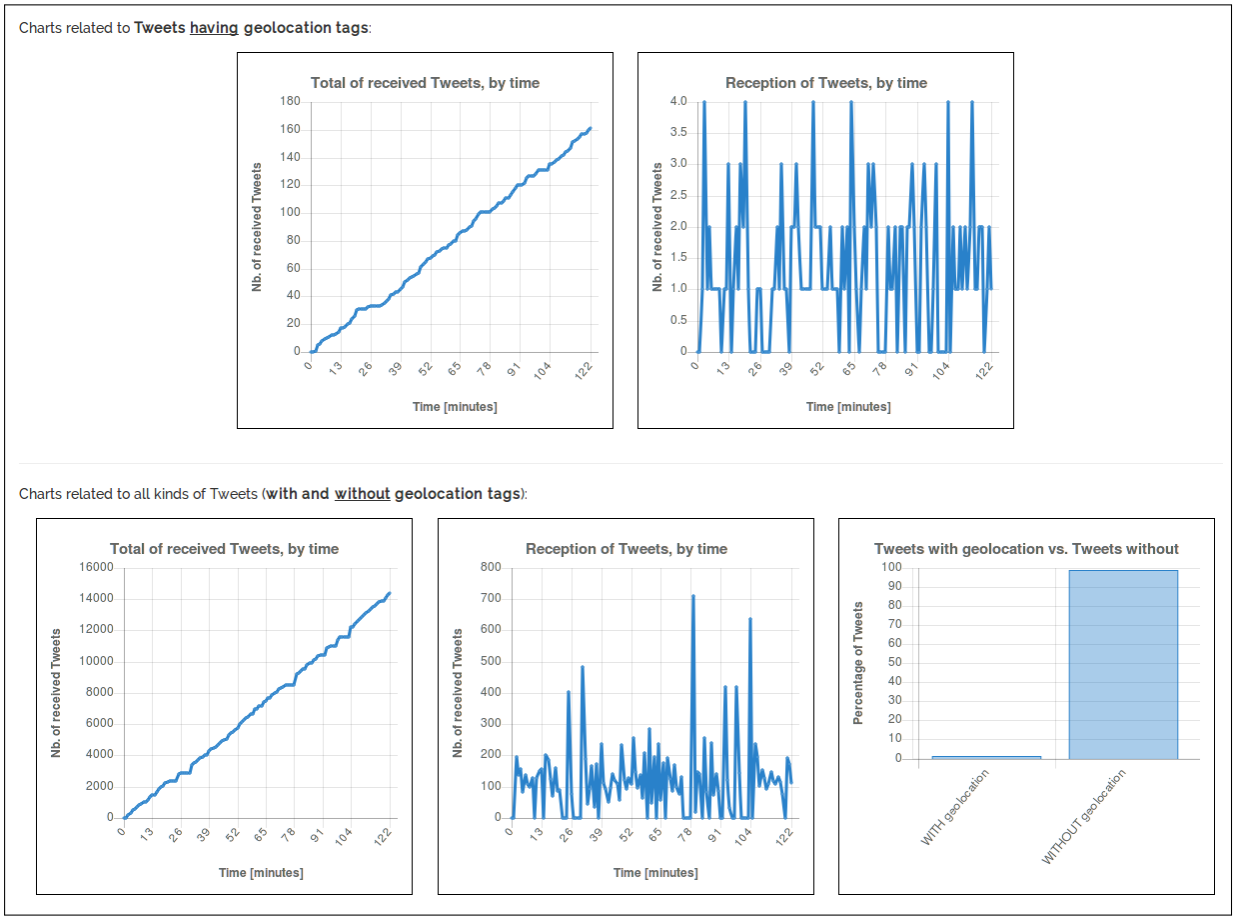


Figure 4.11: analysis' charts of the "paris" keyword in France

As expected, a lot of tweets were received from Paris, since it was the main concerned subject.

France was also analyzed among other countries during the final match of the Euro2016 event (see the "Final Match of the Euro2016 Event" chapter below).

Switzerland

Switzerland is a small country, but does that mean that people do not post from there on Twitter? A first analysis was made on July 20 and 21 from 21:00 to 04:00, notably during the live show of Iron Maiden at the Paleo Festival of Nyon (with the "paleo" subject), and gave just one result at 22:23:

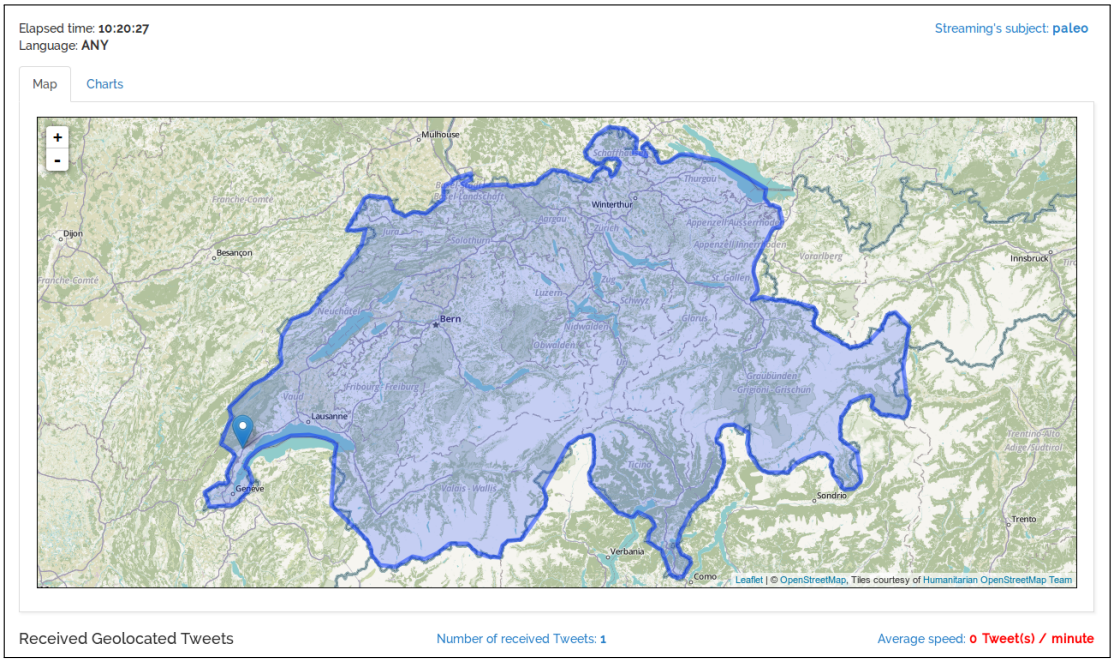


Figure 4.12: analysis' map of the "paleo" keyword in Switzerland

The second Swiss analysis was made on July 21 between 14:00 and 16:00, and was about the main Swiss industries/companies ("coop OR glencore OR logitech OR migros OR nestle OR novartis OR swatch"); it unfortunately gave no result. Out of curiosity, these keywords were tested the next-day for about two hours in the afternoon for the rest of the world, in order to see the impact of these companies:

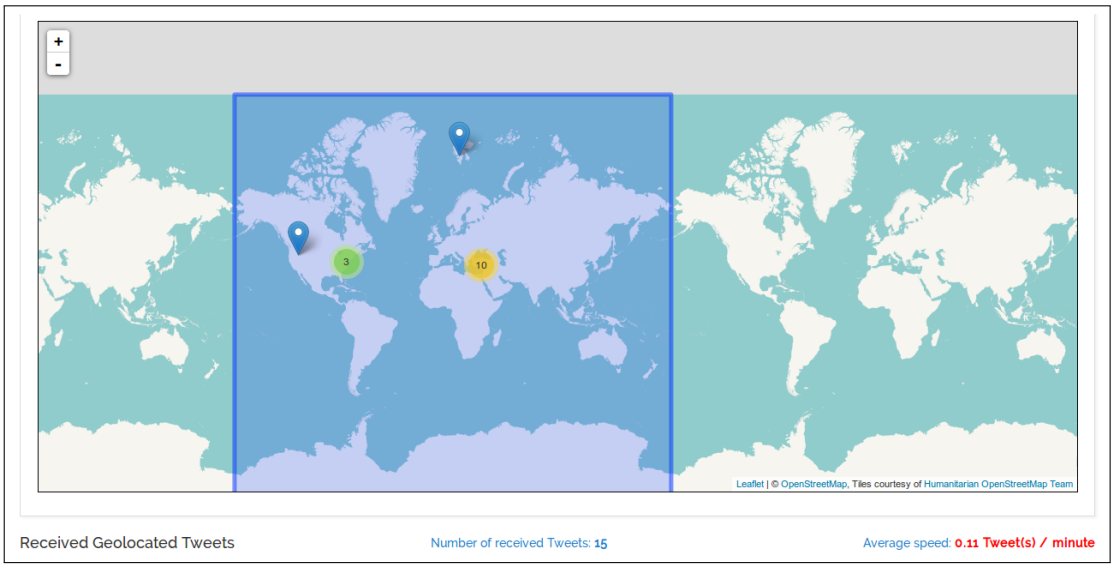


Figure 4.13: analysis' map of the keywords related to Swiss companies in the world

Within these 15 received tweets, eight of them were related to ”migros” (apparently, a famous Turkish mall has this name and might be related to the Swiss company), five of them were false positives with the ”coop” keyword, one was related to ”nestle”, and the last one was related to ”novartis”. The charts were simply linear.

In conclusion, like we also saw with the prototype application, Switzerland is not a really good country to analyze.

U.S.A.

U.S.A. is the country whose people post the most tweets. In addition of the ”job” keyword, other subjects like ”beach” or the name of the election candidates gives good results. The following test was made on July 27 from 16:30 to 19:00 and compares the ”clinton” and ”trump” subjects, which are the names of the two current election’s favorites:

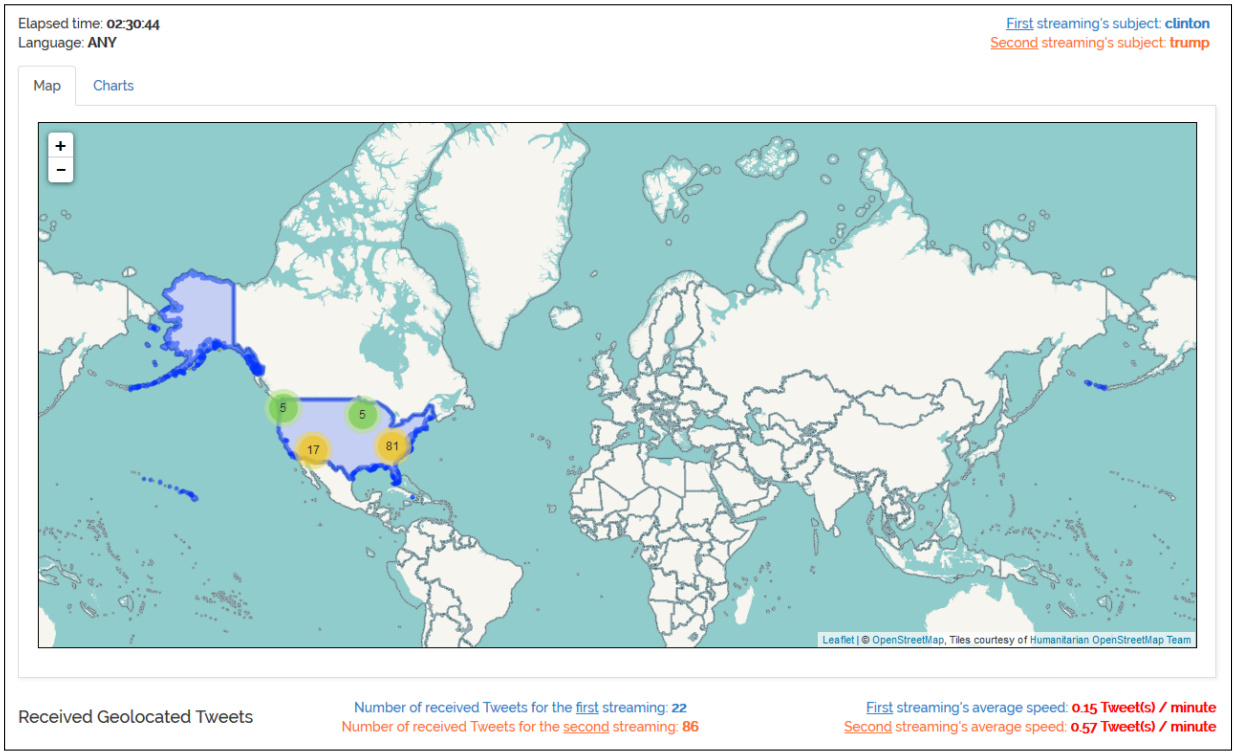


Figure 4.14: analysis’ map of the ”clinton” and ”trump” keywords in the U.S.A.

A lot of tweet were posted from the East Coast, maybe because the time right there was more favorable during the analysis.

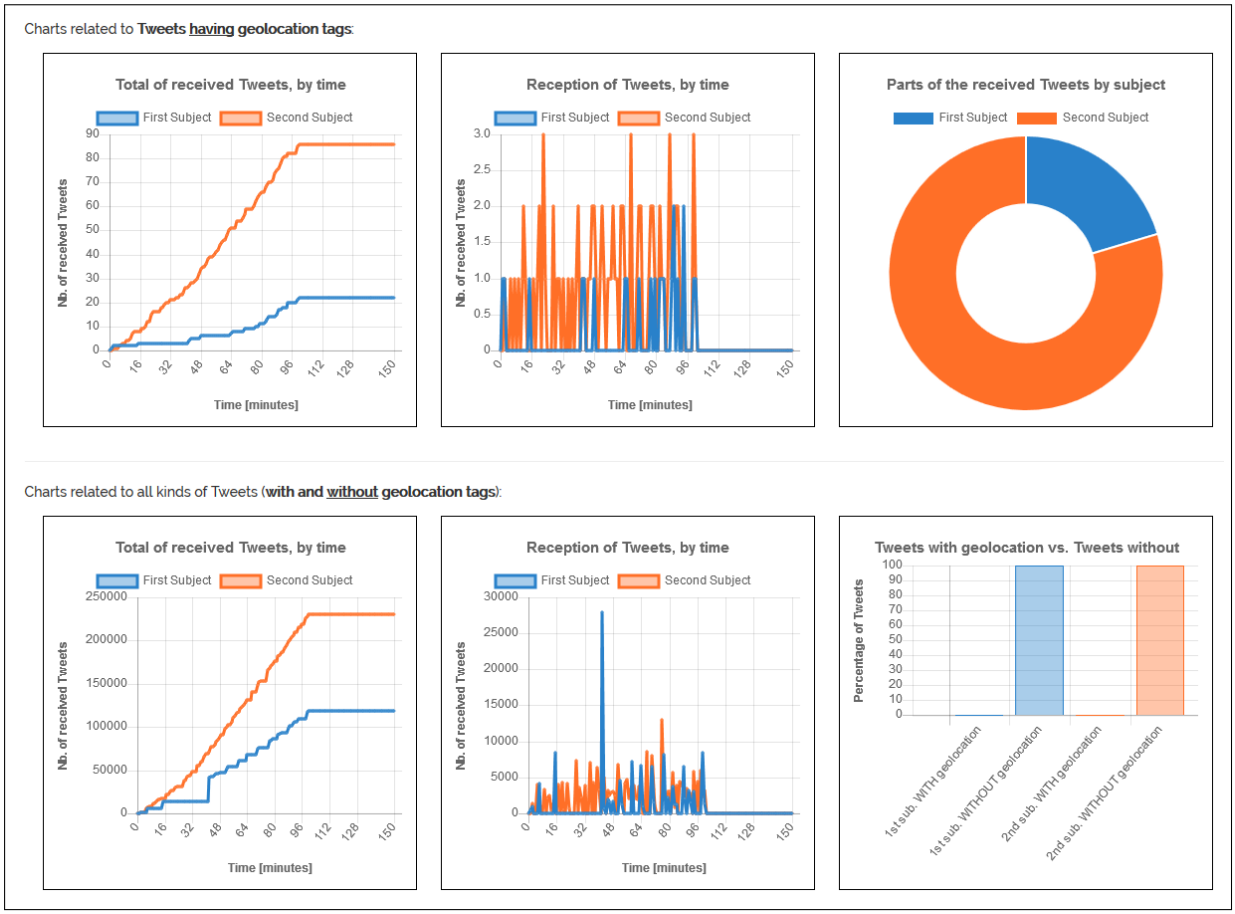


Figure 4.15: analysis' charts of the "clinton" and "trump" keywords in the U.S.A.

For reasons, there was no post during the last analyzed hour. People speak more about the "trump" subject, but a lot of them complain because they do not want him as a president, so this information does not mean that Donald Trump will necessarily be elected. Be aware about the data interpretations!

Argentina

In order to analyze a country that belongs to an area different than Europe and North America, the Argentina was chosen. The ”argentina” keyword was analyzed between 13:30 and 14:45 (between 08:30 and 10:30 in the targeted country) on July 27.

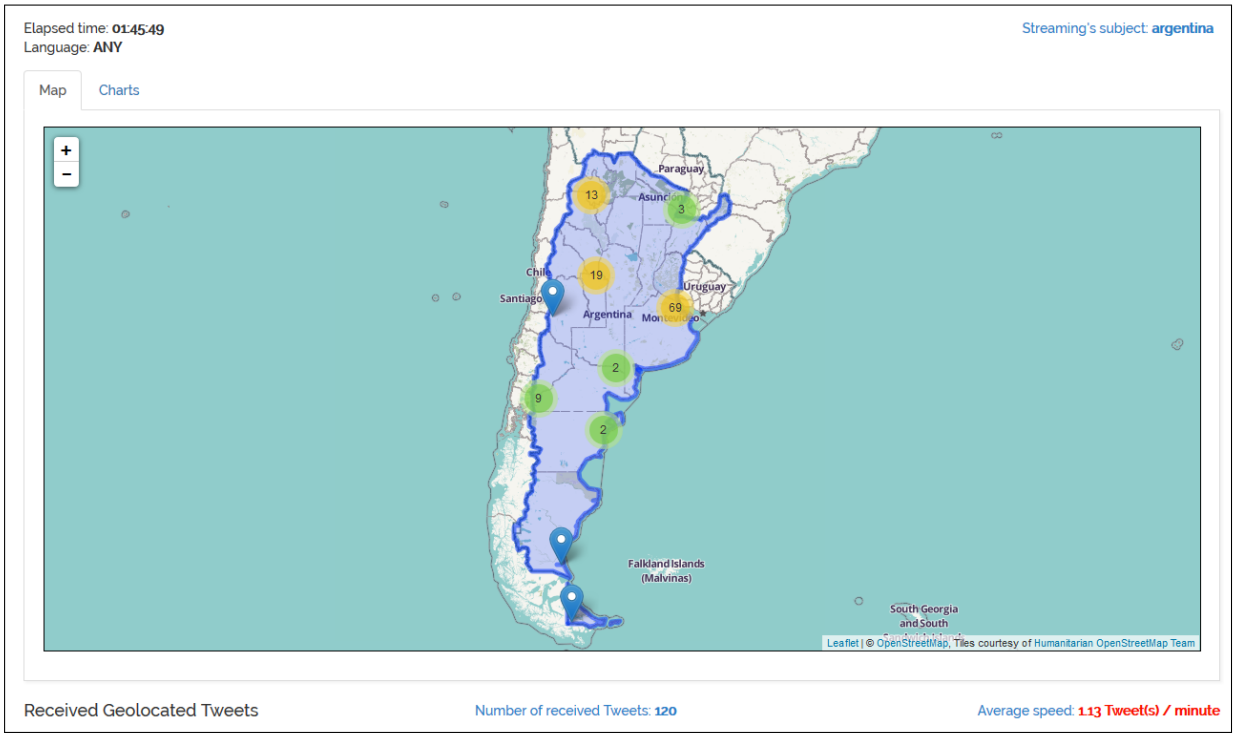


Figure 4.16: analysis' map of the ”argentina” keywords in Argentina

Surprisingly, more tweets than expected were collected; the charts were simply linear.

4.3.3 Final Match of the Euro2016 Event

In addition of the analysis of the Euro2016's final match with the prototype application, the GeoTwit application was developed enough to realize the first and complete analysis of an event with it. Two instances of the application were launched with two different Twitter's accounts (since there cannot be more than two simultaneous streaming processes by account), in order to make two different analyses:

1. The first analysis concerned the keyword ”euro2016” within all Europe.
2. The second analysis was composed of two keywords sets (and thus streaming processes), which analyzed the tweets related to the ”euro2016” keyword and either to France or Portugal within all Europe; these keywords sets were respectively ”euro2016 AND (france OR fra OR francais OR french OR bleus)” and ”euro2016 AND (portugal OR por OR portuguese)”.

Unfortunately, since it was the first large-scale test ever, the first analyzed crashed 30 minutes after the beginning and was unable to properly restart. Fortunately, the prototype application also analyzed the ”euro2016” keyword, but only in France though. Because of this issue, only the second analysis' results are presented below.

The duration of the analysis was **02:33:46**, and the application received **392 tweets** during it, including **228** (about **1.48 tweets / minute**) related to France and **164** related to Portugal (about **1.07 tweets / minute**). As expected, there were more tweets related to France, since this country hosted the event. Here are the maps results (note that since the application was still in development, the interface of the following screenshots may be different as the final interface of the application):

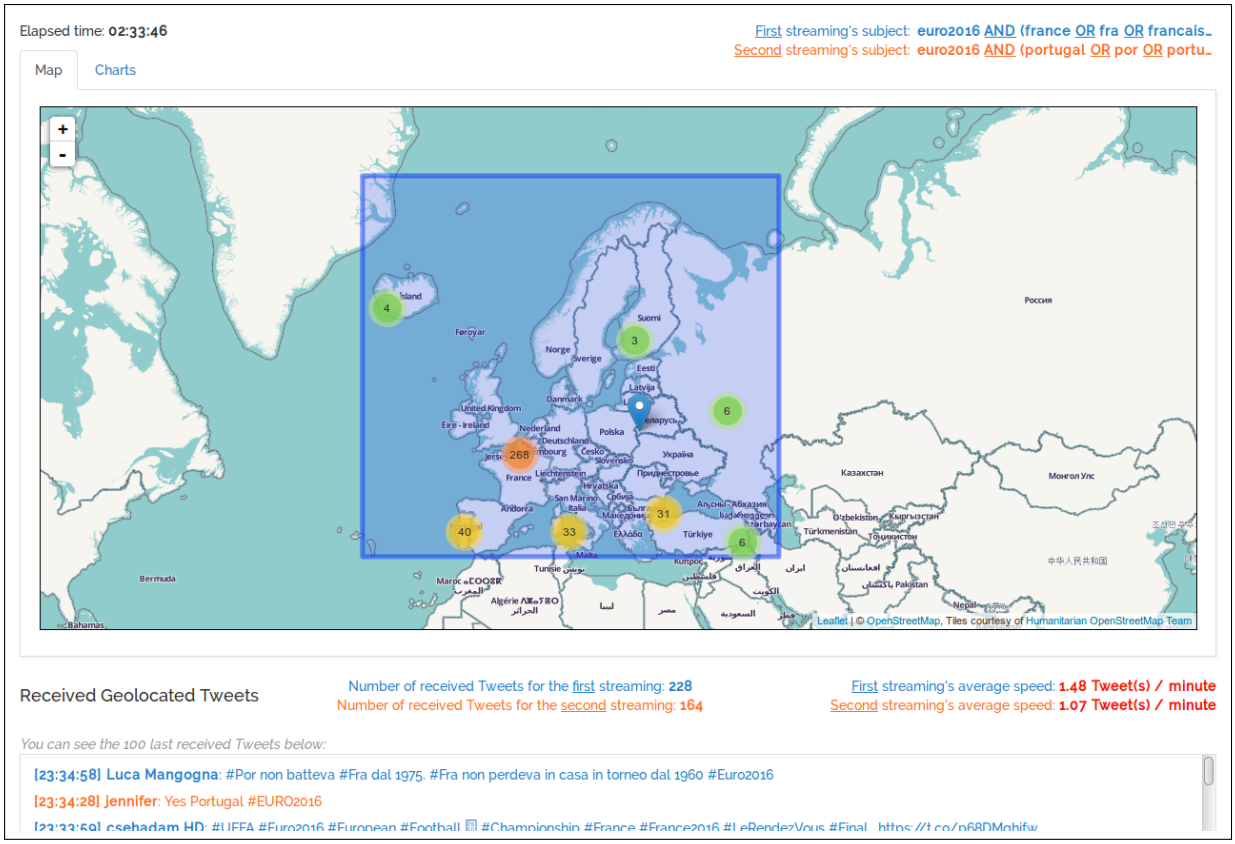


Figure 4.17: tweets map of the FR-PT final match

The whole Europe posted tweets, and a lot of them were posted from France and Portugal, as expected. By zooming on the map, more details appear:

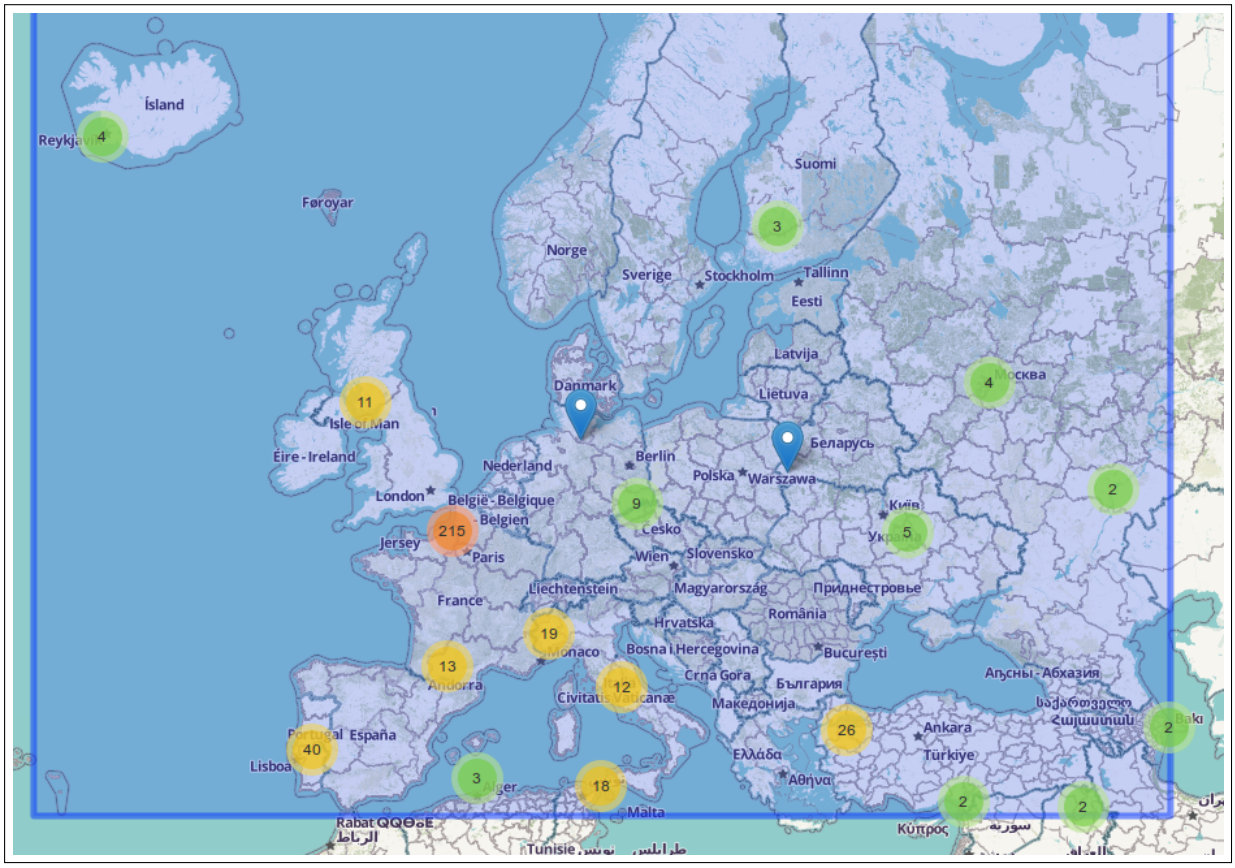


Figure 4.18: first zoom on the tweets map of the FR-PT final match

About 215 on 392 received tweets were posted in the Paris area, where the match occurred.

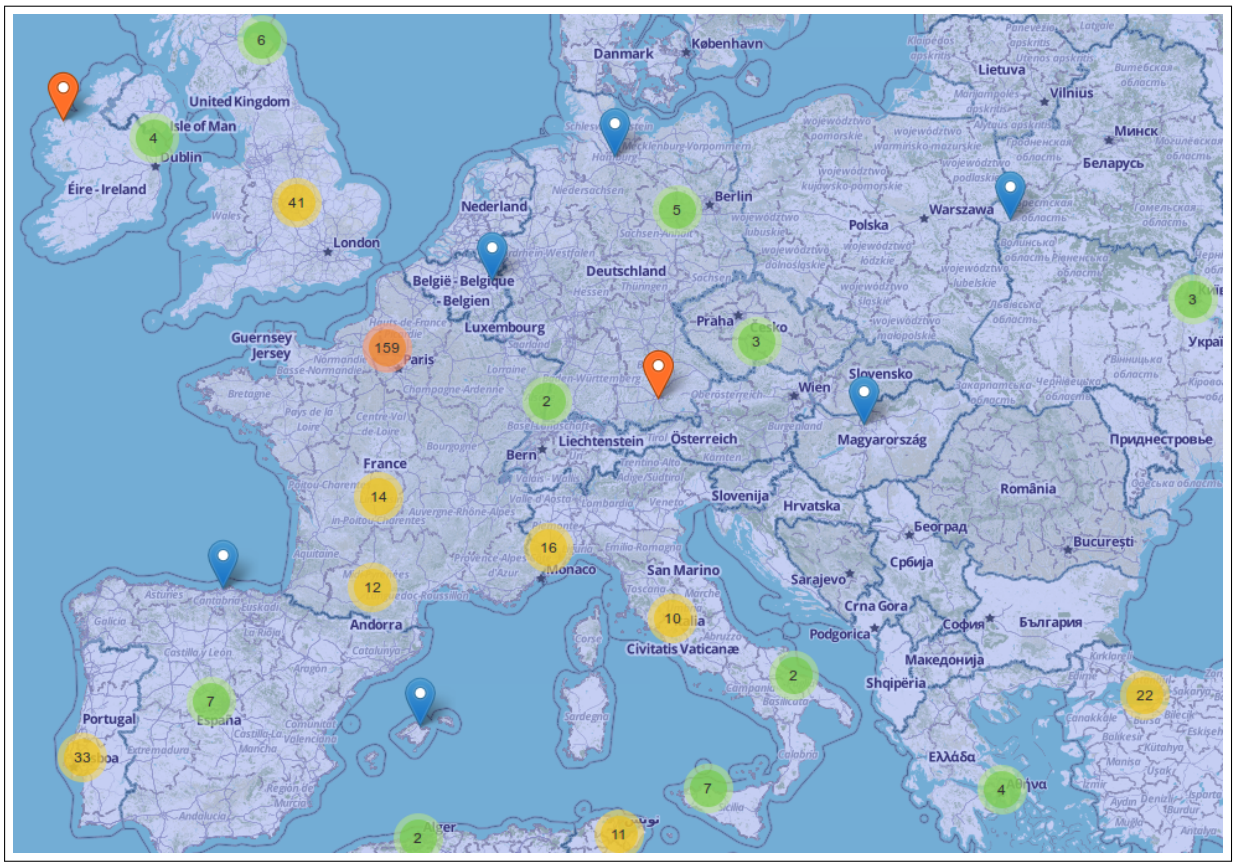


Figure 4.19: second zoom on the tweets map of the FR-PT final match

Without France, the distribution of tweets through Europe was pretty uniform. There still were less posted tweets from the East part of Europe.

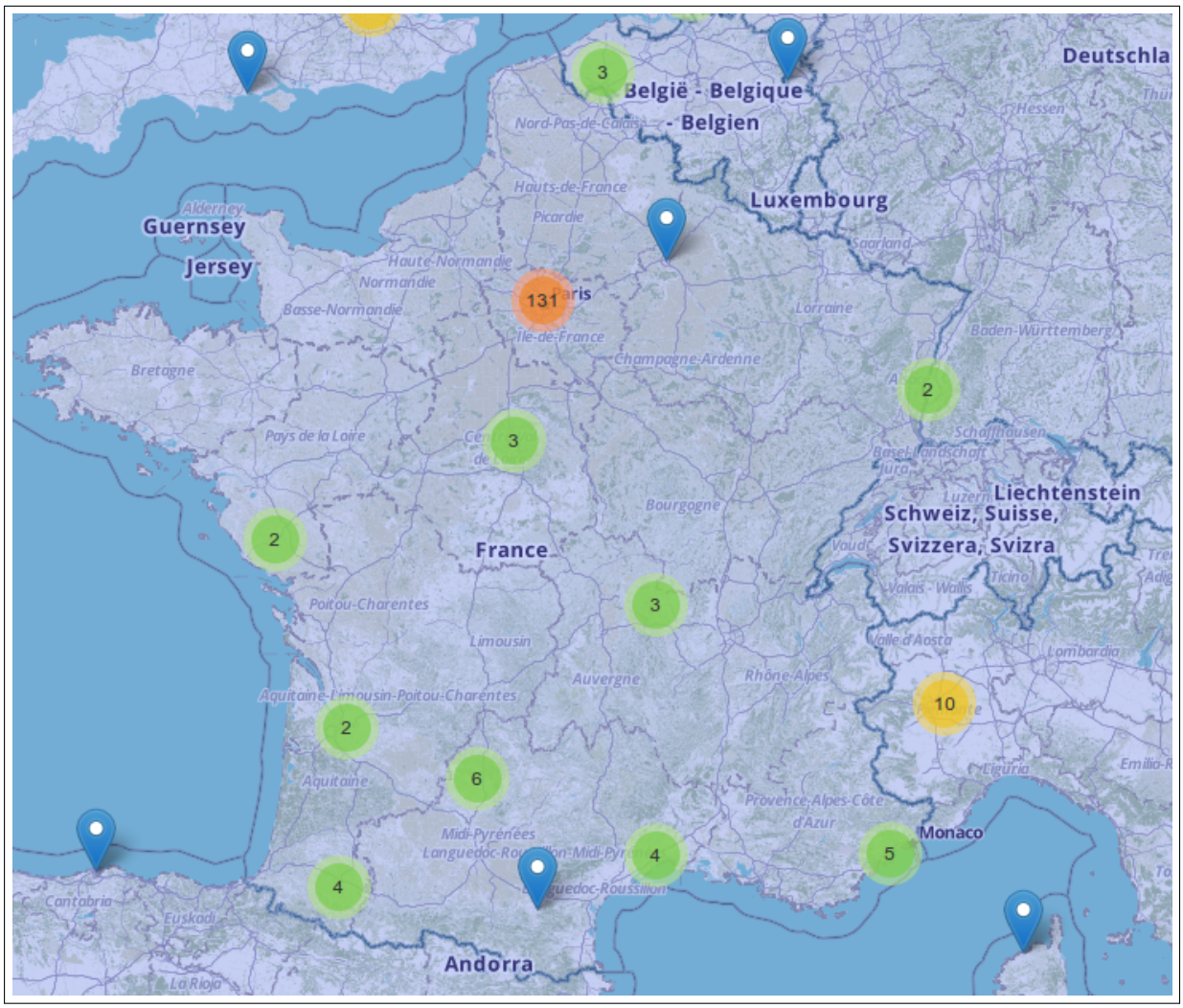


Figure 4.20: France map of the FR-PT final match's results

There was no received tweet from Switzerland. In the opposite, people sent a lot of tweets from Paris. In addition of the maps results, the application also gave useful graphs about the reception rate of tweets. The red annotations were manually added with Gimp:

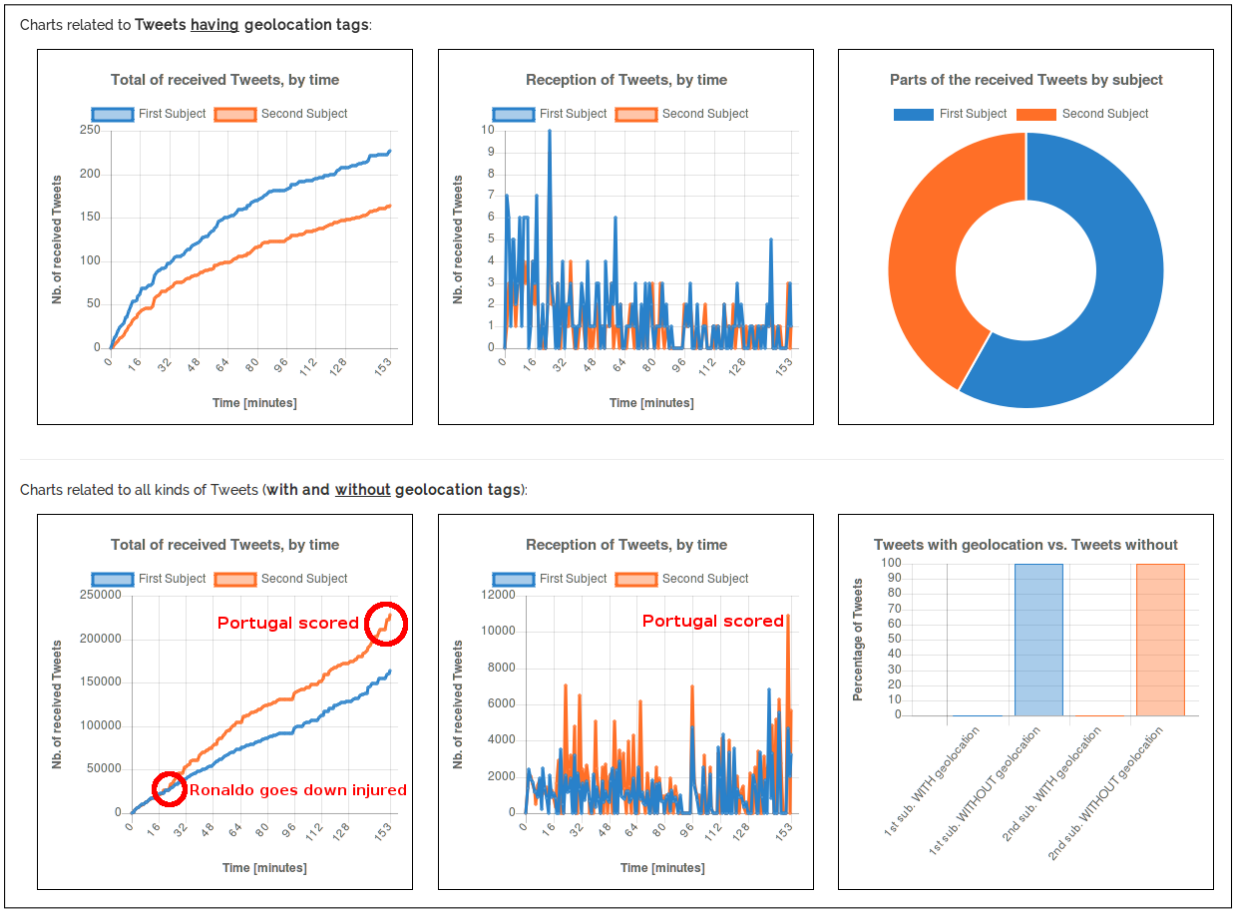


Figure 4.21: France map of the FR-PT final match's results

In the first part of graphs (related to tweets that have geolocation tags), both France-related and Portugal-related tweets' receptions took a logarithmic form. At the beginning of the match, the application received a lot of France-related tweets, and then the receptions smoothed. Concerning the graphs related to all kinds of tweets (with and without geolocation tags), the curves were more linear than before. The reception of the Portugal's tweets had two peaks of data, both when the favorite player of Portugal (Cristiano Ronaldo) goes down injured and when the country scored at the end of the match, giving a final reception of 228'165 tweets related to Portugal, against 164'265 ones related to France. Finally, it should be noted than only a tiny part of posted tweets had geolocation tags (0.14% for France-related tweets and 0.07% for Portugal-related tweets), which seems to follow the average percentages received in the prototype application.

4.3.4 Pokémon GO

One of the events not to be missed of the semester was without a doubt the Pokémon GO game's release. It indeed created a massive flow of data through the social medias. As a reminder and according to Wikipedia, this application is a free-to-play location-based augmented reality mobile game, which allows players to capture, battle and train virtual Pokémons who appear on device screens as though in the real world. It makes use of GPS and the camera of compatible devices.

This test was a worldwide one and was done between 13:00 and 16:00 (Swiss hours) on the 19.07.2016.

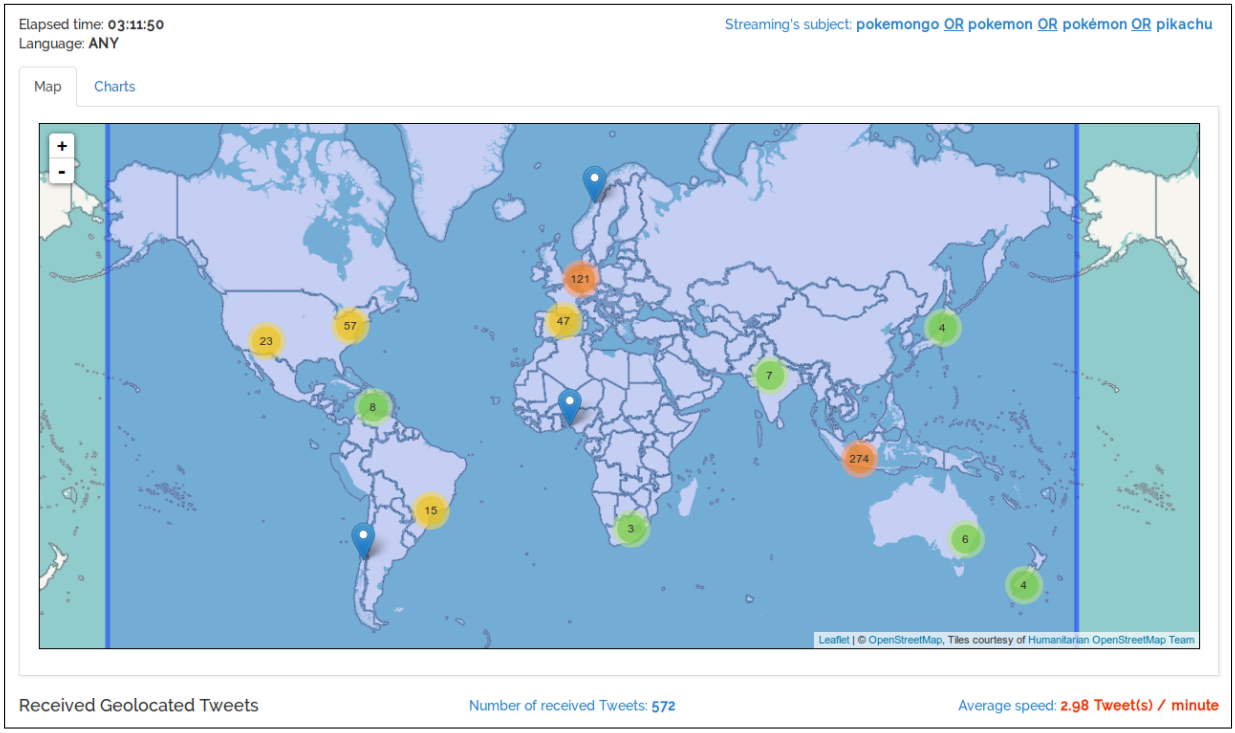


Figure 4.22: world map of the Pokémon GO analysis

A large amount of tweets was posted from the South of Asia and in Europe. It is interesting to note that almost all parts of the world posted about this topic.

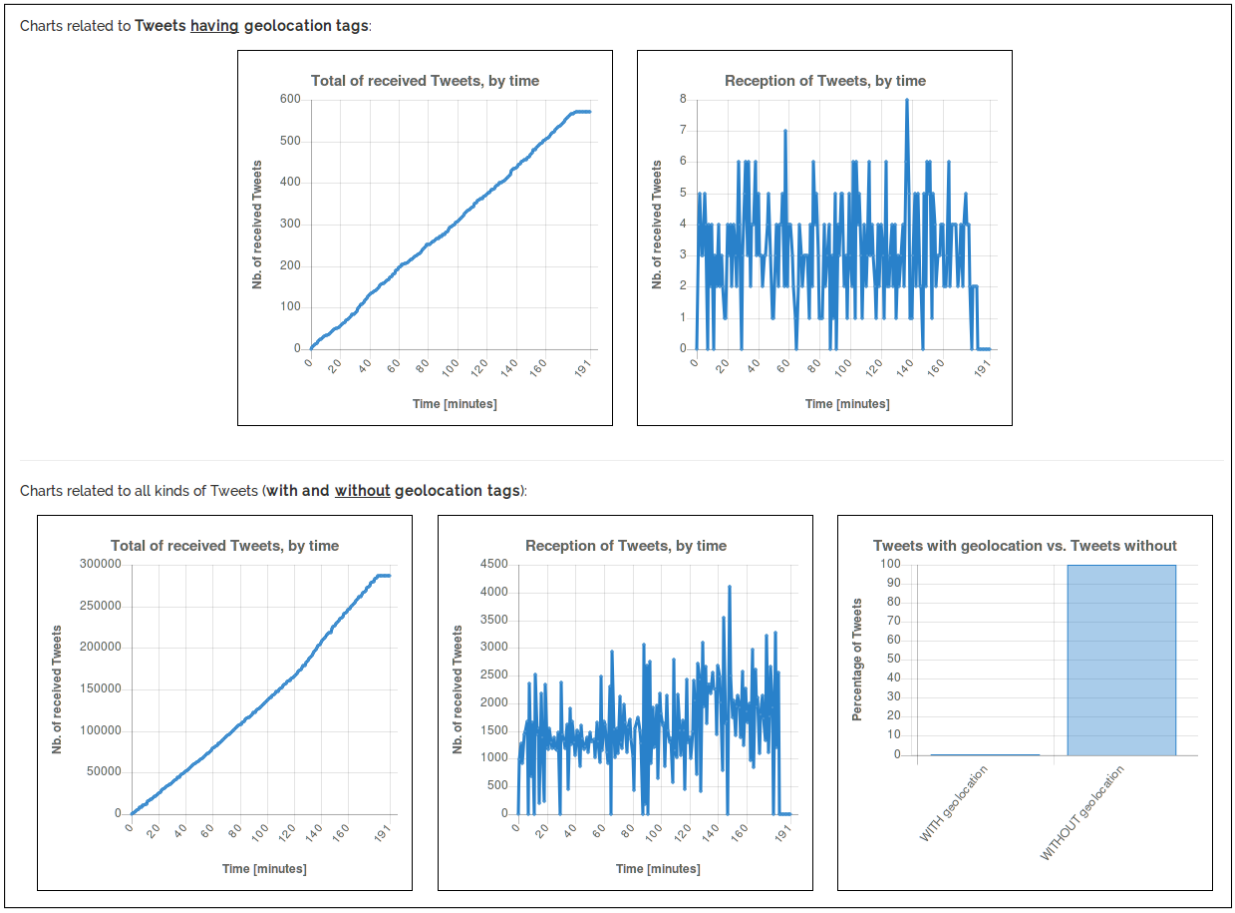


Figure 4.23: charts of the Pokémon GO analysis

The charts are pretty linear in time.

4.3.5 Where are people going on vacation?

If you ever wondered where are people going on vacation during Summer, these results can go some way toward answering your question:

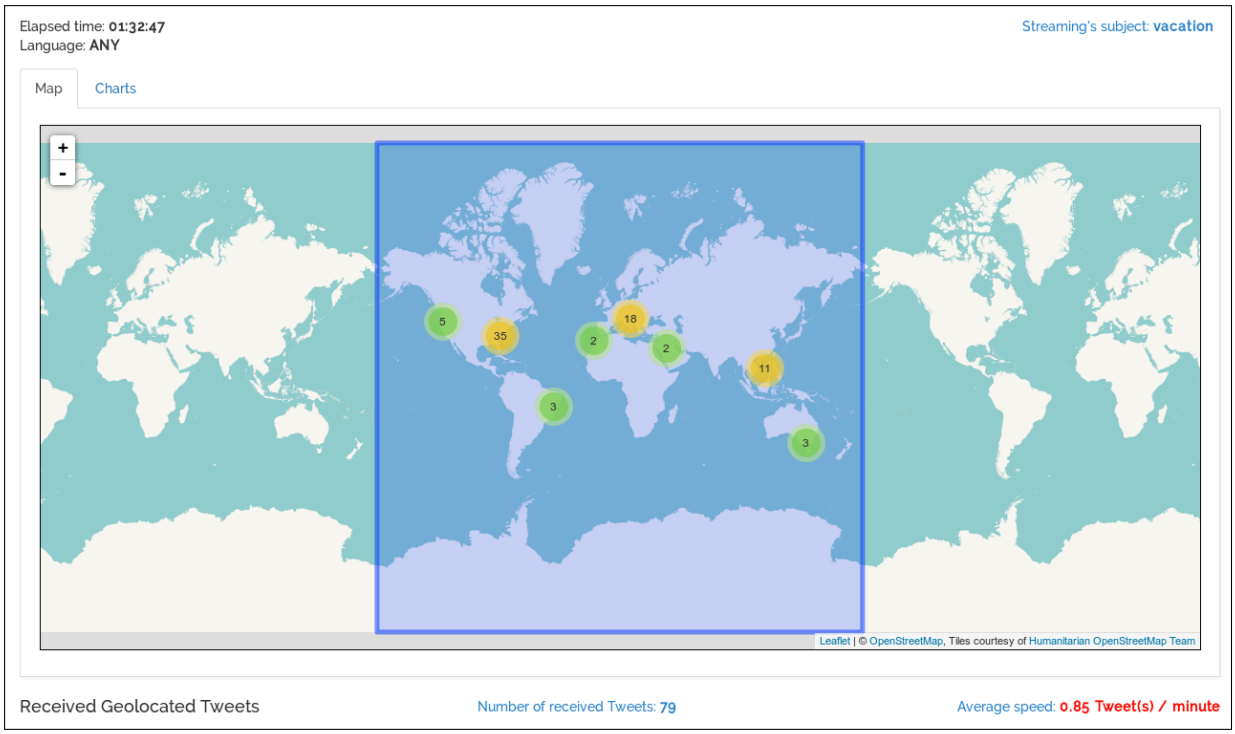


Figure 4.24: analysis' map of the "vacation" keyword in the whole world

This analysis was performed on July 17 from 16:00 to 17:30. People tend to spend their vacation mostly in the U.S.A., in central Europe and in the South of Asia.

4.3.6 Ranking of the Countries by their Post Rates

Finally, here is a little ranking made with the tested countries about their post rates of geolocated tweets:

1. U.S.A
2. U.K.
3. France
4. Argentina
5. Switzerland
6. Italy

Chapter 5

Conclusions

5.1 Achievements

5.1.1 Implementations

All the initial specifications were properly achieved, and non-planned features have been developed as well. Here is a chronological list of the implementation's steps and achievements that went into the completion of this project:

1. Use of the Twitter's APIs, including:
   - Retrieval and filtering of tweets by one or more keyword(s).
   - Retrieval of geographic information attached to a tweet.
   - Analysis of the percentage of tweets having geographic information (necessary for further processing) and the search of keywords having relevant results, in order to determine the best keywords for use in order to test the different project features.
2. Use of a cartographic library in order to use geographic maps.
3. Development of the tweet's acquisition and analysis parts for the dynamic mode first, *then for the static one.*
4. Development of the web interface of the application, allowing the user to enter subjects to compare, to choose the geographic areas on the map, to visualize results and to interact with the map.
5. Development of a grouping algorithm that groups the tweets according to the map's zoom level.
6. *Development of charts that list relevant interesting information about the results.*
7. *Built-in possibility to export the queried data as a text file at the end of a streaming process and to import this file thereafter.*
8. Testing and validation.

Italic parts are the developed extra features that were not planned at the beginning of the semester, since some points were still fuzzy at that time.

The work was regular and homogeneous over the whole semester, as you can see in the following GitHub's chart:

Figure 5.1: graph of the GitHub commits that were done during the semester

There are more commits from June, because the work process jumped from half-time to full-time at this moment.

5.1.2 Known Issues

Even if a particular attention was paid during the development of this project, some issues can still persist. Here are the current known ones:

- If a user simultaneously runs two streaming processes on the same computer, but in two different tabs or browsers, both of these processes will write the same file-to-export at the same time, which could give strange results.
- Minor bugs of display are present in the browsers different than Firefox.

5.2 Future Extensions

Here is a non-exhaustive list of various extensions and improvements, which could be developed in a potential future developmental update:

- Improvement of the GUI of the web application.
- Improvement of the code's structure (creation of multiple files for the JavaScript code, etc.).
- Implementation of a user management platform (Sign-up, Log-in) coupled with a database, in order to back up the different obtained results.
- Storage and display of the most successful keywords as well as the areas that have the best geographic results within the application.
- Retrieval and display of the trending topics on Twitter.
- The possibility to separate the results by their phrases (the words contained between each "OR"), in order to have improved comparison tools.
- The possibility to hide/show the results of a subject, when there is several of them.
- The possibility to manually configure and enable/disable the cluster groups on the map.
- Addition of a progress-bar that indicates the current progression of the static search's process (since it could take times).
- Addition of filters for the search tools, like the possibility to filter tweets by their status (recent or popular) in the static tool.
- Find a way to be less limited by Twitter in the static search.
- Addition of social widgets (sharing of aggregated results, notes, etc.).
- Addition of a gamification system (allowing users to accumulate points and trophies by the results rate of their search, implementation of user ranking, etc.).
- Improvement of the security of the application (especially for the web sockets' management).

5.3 Encountered Problems

There are two types of encountered problems: the technical and the organizational ones.

5.3.1 Technical Problems

Here is the list of the main technical encountered problems:

- Since the Play Framework is pretty new compared to other older technologies, the documentation of this framework is logically poorer. Due to this little problem, I spent many time on significant things in order to properly use them, for example with all the session management and the protection of the application, trying to prevent a non-connected user to access the web site's content. In order to solve this problem, I had to search the web a lot and use my knowledge in other programming languages.
- I spent many hours trying to integrate the Twitter4J library in the Play project. It was indeed different when I tried in the pure Java prototype application, where everything was simple. I searched for a while to find the right path to include in the project's build file, and when I finally succeeded, I spent many other hours trying to complete the Twitter's connection process, since it was not properly documented in the library and some things changed with Scala.
- The learning of the different libraries and technologies took some time, but nothing really significant.
- A lot of tiny (but time-consuming) things has to be implemented in order to make the application work, like the algorithm that checks if a point is inside a polygon, the one that selects the borders of a country's territories, the web sockets server, etc.

5.3.2 Organizational Problems

And here is the list of the organizational problems:

- The main organizational problems were the time management and the personal organization: since there was way more work than I thought in the other courses of the semester, I never was able to work during the 16 planned hours per week. However, I made myself sure to assiduously work during the free hours. The time management was also a little bit chaotic during the two last weeks of the project.
- I spend time translating the first documentation's version from French to English (since it was not planned in the beginning), but it was not a huge problem in the end.

5.4 Final Discussions

The overall organization and execution of the project has been very smooth. Indeed, every planned task was successfully completed, in addition to a few new ones, which is a personal success. It has been a truly rewarding experience to work my way through this autonomous project over the duration of the semester, owing partly to the steep learning curve that it has provided (like a new language and data mining techniques). Despite not being perfect, I certainly feel satisfied at the end of this enriching process and proud of the end-results.

Regarding the future of the application, GeoTwit has several potentially interesting use-cases; for example, a real-time comparison of a population's voting behavior across different parts of a country (e.g. German, French and Italian parts of Switzerland) during national elections akin to exit polls, or an analysis of people's reactions to the passing or proposal of a new law, an analysis of the Twitter's data during a major event/festival, a comparison of certain related subjects (e.g. comparing geographical variation of perception of Java and Scala languages), or lighter hearted topics, for instance, geographic variations in use of the keyword "vacation".

A final word on the metaphysical relevance of this project: GeoTwit can philosophically be regarded as a data-mining tool that intersects human behavior and technology via the platform of social networks, allowing an end user to analyze and compare ethnological subjects in conjunction and with the aid of technological advancements.

List of Figures

Bibliography

[1] elliotbonneville, *4 creative ways to clone objects in JavaScript*, November 2011: http://heyjavascript.com/4-creative-ways-to-clone-objects/

[2] W. Randolph Franklin, *PNPOLY - Point Inclusion in Polygon Test*, February 2016: https://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html

[3] Mitja Trampus, *Twitter - Evaluating language identification performance*, November 2015: https://blog.twitter.com/2015/evaluating-language-identification-performance

[4] Play Framework Team, *Play 2.5.x documentation*, https://www.playframework.com/documentation/2.5.x/Home

[5] Twitter Team, *Documentation of Twitter API*, https://dev.twitter.com/overview/documentation

[6] Twitter4J Team, *Twitter4J-4.0.4 JavaDoc*, http://twitter4j.org/javadoc/index.html

[7] Leaflet Team, *Leaflet 0.7 API Reference*, http://leafletjs.com/reference.html

Appendix A

Content of the Project's Package

Here is the content of the project's files:

- ***app/GeoTwit*** → contains the GeoTwit application's code.
- ***doc*** → contains the project's thesis.
- ***prototypes*** → contains the four developed prototype applications:
  - *leaflet*
  - *leafletAndTwitter4J*
  - *leaflet_countries_borders*
  - *twitter4jDesktop*
- ***README.md*** → contains information about the content of the package and the instructions of installation.

Appendix B

Instruction Manual

B.1 Installation

In order to use the application, please first install Play Framework with activator on your computer[1]. Once done, go in the */app/GeoTwit* folder, type "**activator run**" in a console and open the *http://localhost:9000* URL. The process will take a while the first time you access this page, since it has to download the libraries and compile the application.

B.2 Software Usage

Firstly, you can access the "About" and "Help" pages anywhere in the application with or without being connected by clicking on the related links.

B.2.1 Home Page

1. Once you successfully reached the web site, you access the GeoTwit's home page. In this page, you can visualize the different features offer by the application, then log in with your Twitter account in order to be able to use them. Click on the *Get Started* button located at the bottom of the page to connect.

Figure B.1: home page

[1] https://www.playframework.com/documentation/2.5.x/Installing

2. By clicking on the connection button, you will have to authorize GeoTwit to use your Twitter account, in order to allow it to read public tweets with your account (none of your account's data are saved or used but the user's name, which is displayed later). Enter you credentials and then click on the "Sign In" button.

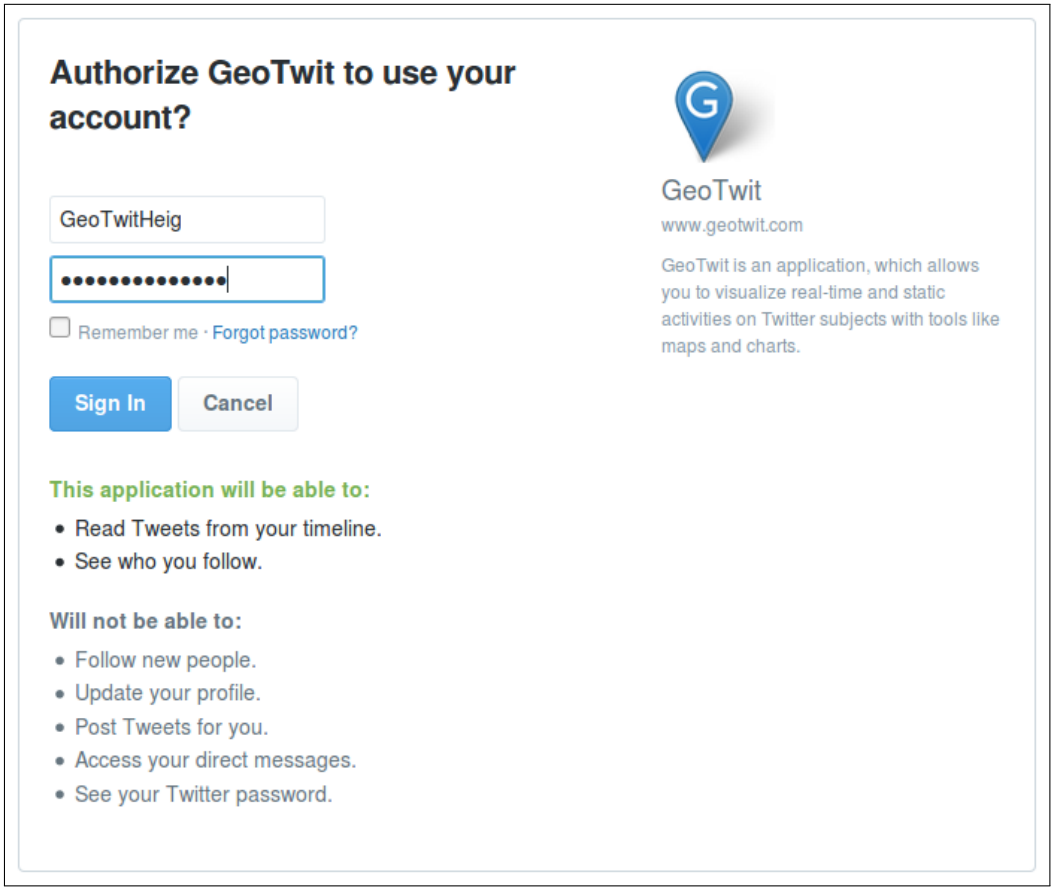


Figure B.2: Authorization page of GeoTwit

You can use the special GeoTwit's Twitter account (*GeoTwitHeig - @heigVdRocks42*) for your tests, if you wish.

If you click the "Cancel" button, or if something went wrong, you will be redirected on the home page with an explicit error message.

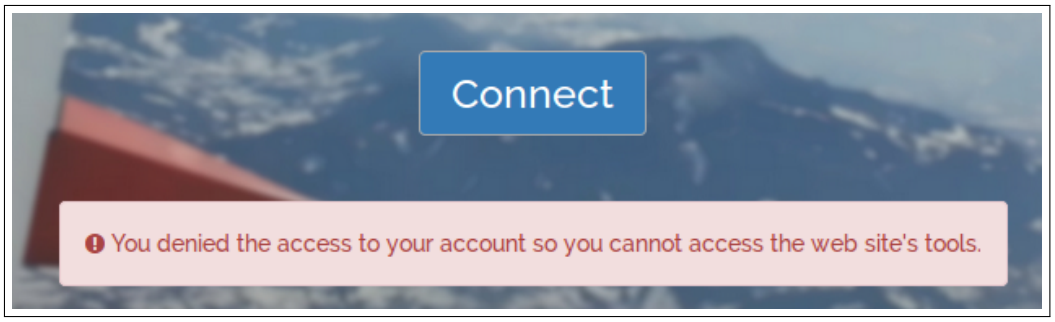


Figure B.3: explicit error message that appears when something went wrong during the connection process

B.2.2 Search Pages

1. Once connected, you can access the application's tools through the search pages. These pages contain a header element, a content element and a footer element:

   - **Header element**: this element is located on the top-most part of the interface, and contains the GeoTwit logo as well as a message that indicates with which account you are still connected, and finally a disconnect button that disconnects you and redirects you on the home page once clicked.

Figure B.4: header element of the search pages

   If you click on the logo, you will be redirected on the main search page.

   - **Content element**: this element is located between the header and the footer, and contains the content of the page you currently are in. You can find more information in the following paragraphs.
   - **Footer element**: this element is located on the bottom-most part of the interface, and contains links to respectively access the *Help* and *About* pages.

Figure B.5: footer element of the search pages

2. The main search page is the first page you reach once you successfully connected. It contains a tabs system that allows you to navigate through the dynamic and static modes.

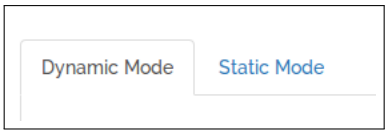


Figure B.6: tabs system used to navigate through the modes of the application

B.2.3 Dynamic Mode

The dynamic (also called "streaming") mode is the default and most finalized mode of the application. It allows you to listen to a Twitter's stream of data in order to receive and analyze tweets that match the entered filters in real-time.

1. In the interface of this mode, you can firstly import a previously exported GeoTwit (".gt") text file by clicking on the importation's button.

Figure B.7: button used to import a previously exported GeoTwit file

If the file is either not in the right format (i.e. an image), is empty or is not valid, an error will be displayed.

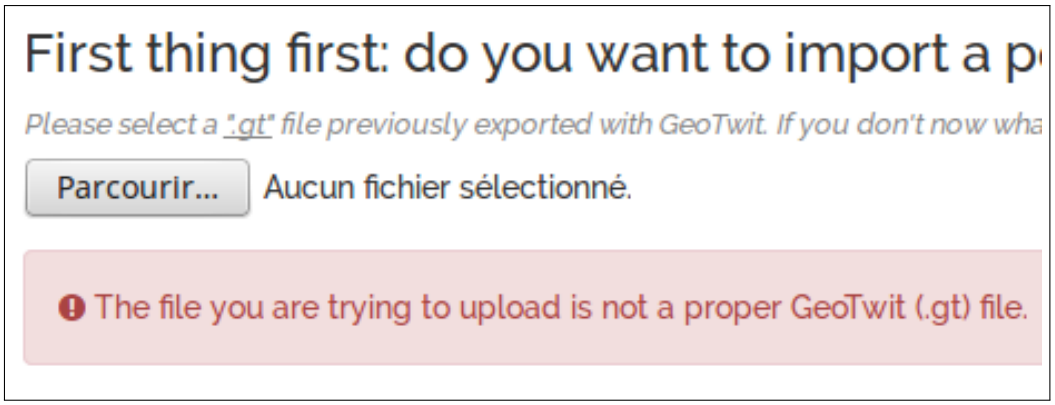


Figure B.8: error message displayed when an importation error occurs

If everything went well, you will be redirected on a frozen version of the dynamic mode's results page that displays the results contained in the file (see the points below).

2. If you want to start a new streaming process, you have to properly fill the form:

   - **Keywords Sets**: you have to fill the first keywords set at least, in order to provide a query to filter tweets. For each of these keywords sets, you can fill both the "One of these keywords..." and/or the "All these keywords..." fields separately, which are respectively used to create OR and AND filters.

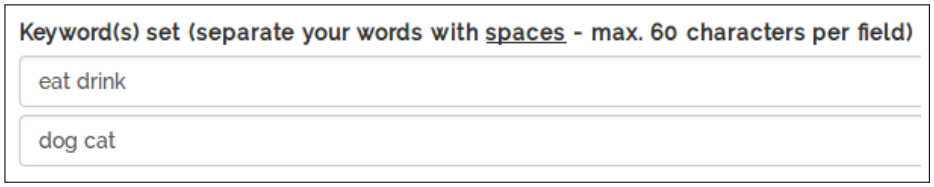


Figure B.9: example of value that can be contained in the first keywords set field

AND filters have priority over the OR ones. In the figure above, the generated query filter can be interpreted as "(dog AND cat) AND (eat OR drink)".

If you fill the second optional keywords set, a second streaming process will be started simultaneously in order to compare the subjects of each subject between them. As mentioned, each field of the keywords sets must at most be 60 characters long.

- **Location**: when selecting a location, you have to choice to either choose a default area among a list of purposed countries or manually draw a rectangle on the map (but never both options at the same time).

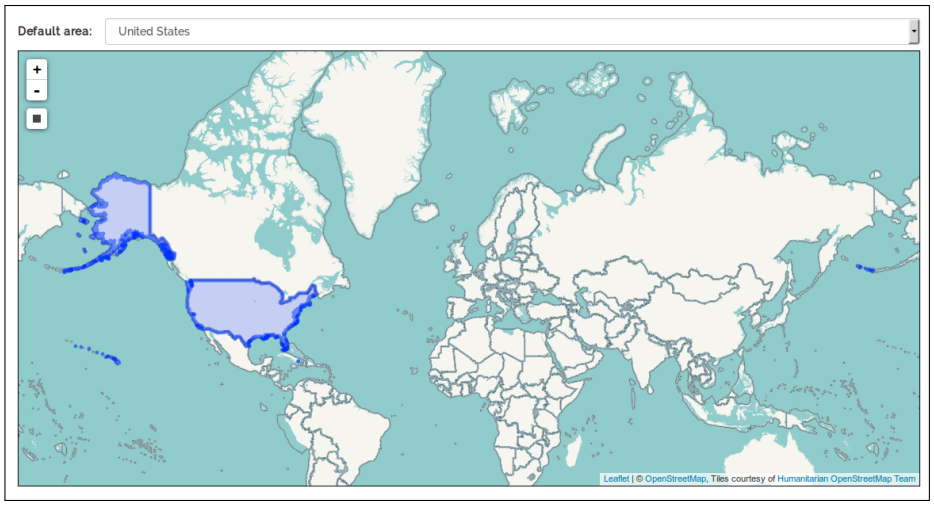


Figure B.10: selection of the location with the default area option

If you want to draw a rectangle, click on the button located on the top-left part of the map.

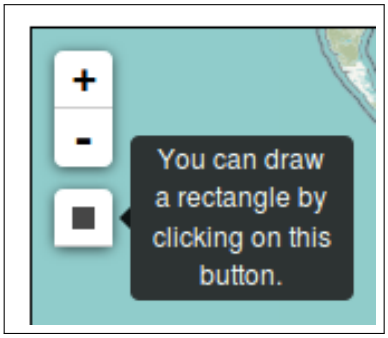


Figure B.11: button that allows the user to draw a rectangle on the map

Then drag-and-drop an area on the map to draw the rectangle.

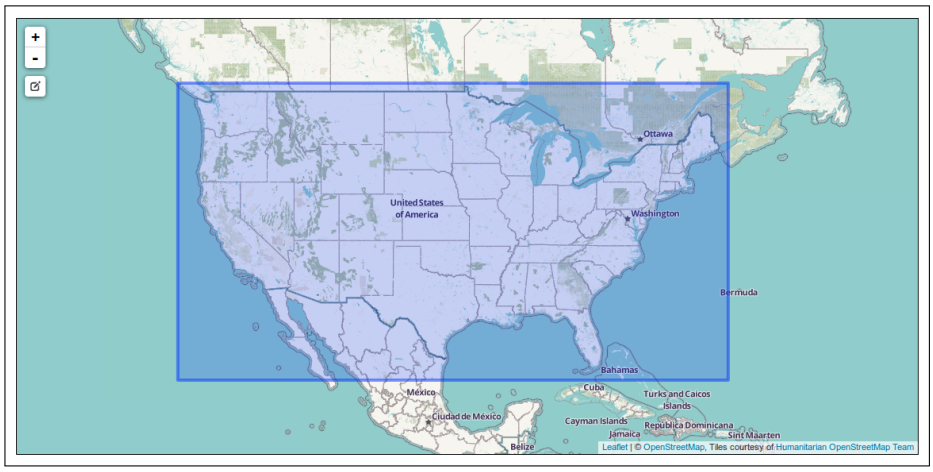


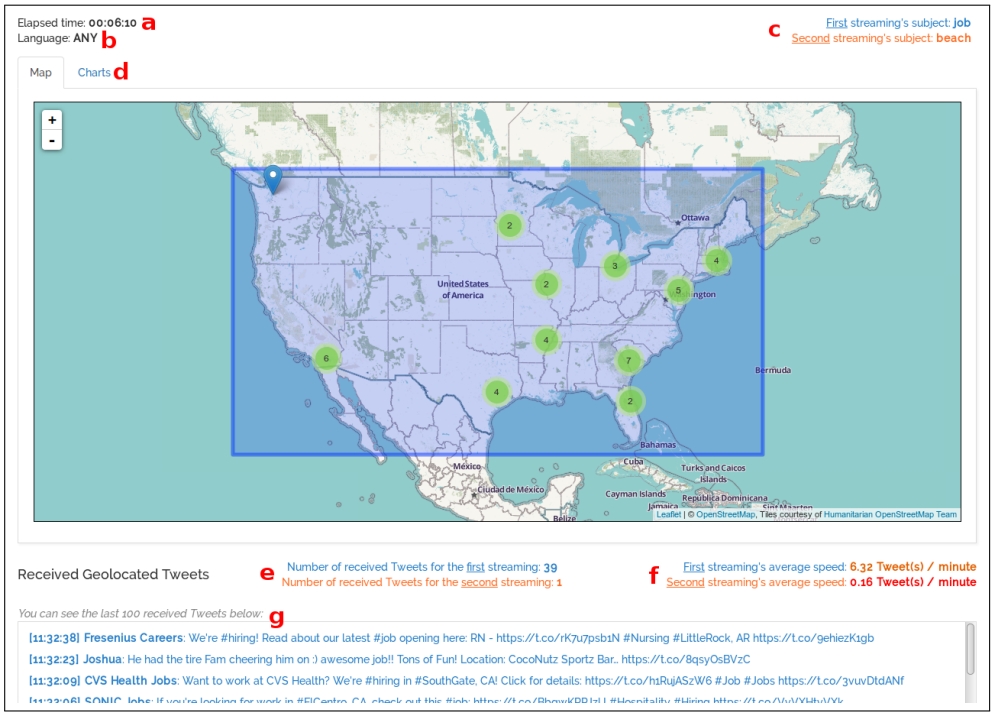
Figure B.12: selection of the location by drawing a rectangle

You can observe than the "Draw" button no longer exists once you drew the polygon, and was replaced by a "Edit" button, which allows you to edit the drawn rectangle.

- **Language**: you can either keep the default "Any Language" selected value or select a language among the list of purposed languages, in order to filter tweets by their writing's language.

Figure B.13: selection of the language used to filter received tweets

3. When you click on the "Start Streaming" button, an error message will appear if one of the fields is not valid. In the opposite, if everything is valid, you will be redirected on the results page, which displays the real-time results.



Several information are available on this page, among which:

(a) **the *hh:mm:ss* time elapsed** since the beginning of the process;

(b) **the language** used to filter the incoming tweets;

(c) **the human-comprehensive values of the keywords set / subjects**, identified by colors: blue for the first subject (always displayed), and orange for the second one (only displayed if you set the second keywords set);

(d) **a tabs system** that allows you to navigate through the "map" and "charts" views:

- in the **map** view, you can see the received tweets grouped by clusters; once you zoom the map enough (a zoom level of 8 or more), these clusters disappear and you can access the tweets' markers one by one. By moving the cursor over a marker, the related tweet's content is displayed:

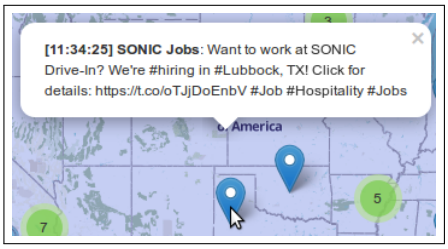


Figure B.15: display of the tweet content by moving the mouse over the related marker

- in the **charts** view, you can access up to 6 different charts about the reception rate of tweets for each subject. If you filled only one keywords set, the third "parts" chart will not exist.

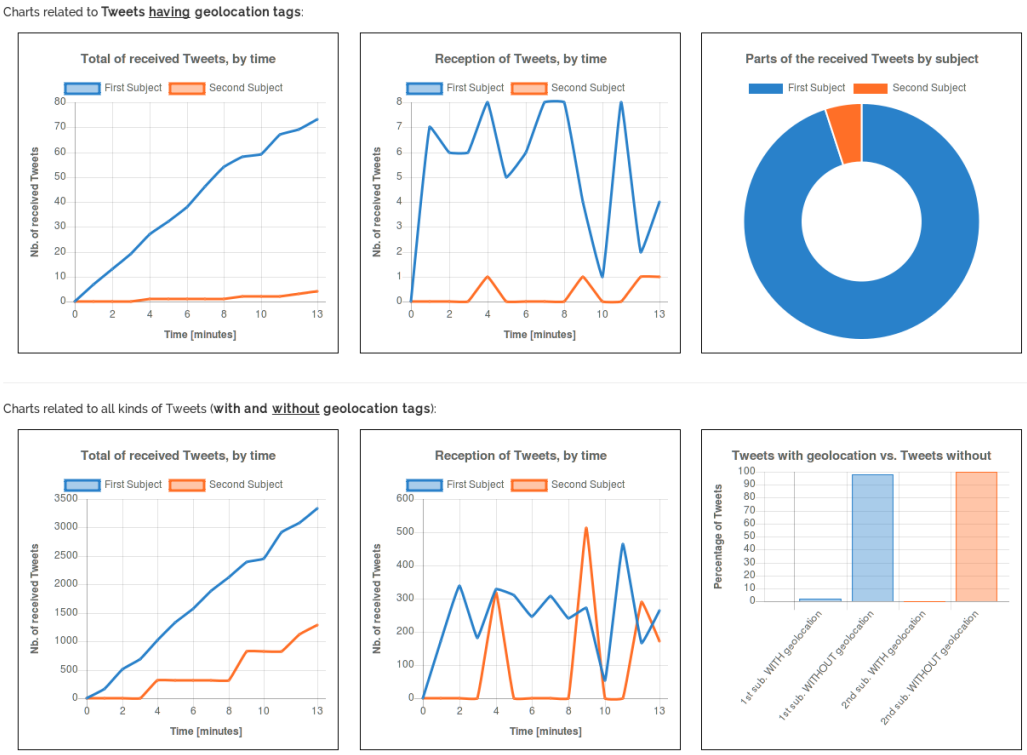


Figure B.16: chart view

The colors of the elements are the same as their related subjects. These charts are firstly refreshed each second until 60 seconds, then each minute. When you move the cursor over a graph's element, you will get more information.

(e) **the number of received tweets** for each subject;

(f) **the speed value of the reception** for each subject;

(g) **a panel that contains the last 100 received tweet's content**.

In addition, a "Stop Streaming" button is located on the top-right part of the page.

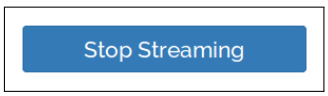


Figure B.17: the button that allows the user to stop the current streaming process

Once you clicked on this button, the current streaming process will stop, a new "Make a new search" button will appear instead of the old "Stop Streaming" one, and a pop-up windows asking you if you want to export the analyzed data as an external file will instantly appear.

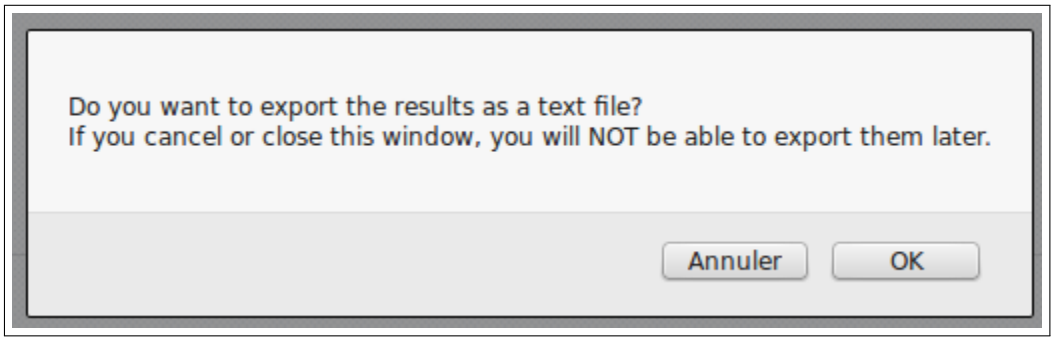


Figure B.18: pop-up that asks the user if he want to export the analyzed data

If you press the "Ok" button, you will be able to save the file wherever you want on your computer in order to import it in the application later. If you press the "Cancel" button, the window will close and you will not be able to export and thus import the file later.

Once the streaming process stopped, you can still consult the results, but everything is frozen. By pressing the "Make a new search" button, you will be redirected on the main search page.

B.2.4 Static Mode

The static mode of the application allows you to retrieve and analyze tweets already posted in the last days in a static and frozen way. Be aware of the fact that Twitter restricts the maximum possible number of retrieved Tweets, so you may not get every possible results. Indeed, you can retrieve a maximum of "only" 17'000 Tweets if you filled only one subject, or "only" 8'500 per subject if you filled the two keywords sets. If the results seem restricted, reduce the dates range in order to get more specific results and try to be more specific with your search criteria.

1. If you want to statically search for tweets, you have to properly fill the form's fields:

   - **Keywords Sets**: they act the same way as the keywords sets from the dynamic mode above, so please refer to them.
   - **Range of Dates**: these values will be used to get the tweets that were posted during this time; they have to be well-formatted *YYYY-MM-DD* dates. Since Twitter allows us to get tweets from at most 9 days ago only, you won't have any result if you search for Tweets before this day. Once you clicked on a field, a date picker element appears:

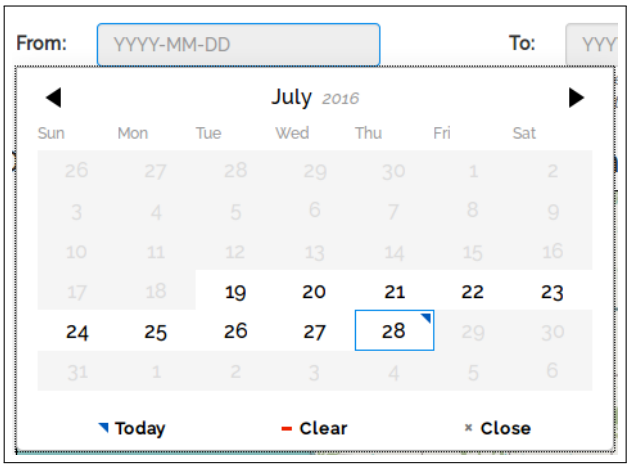


Figure B.19: date-picker element that appears when the user clicked on a date field

   - **Drawing of a Circle**: in order to select an area, you can only draw a circle in the static mode, because of the Twitter's requirements. The drawing process works the same way as the rectangle's drawing of the dynamic mode.
   - **Language**: you can either keep the default "Any Language" selected value or select a language among the list of purposed languages, in order to filter tweets by their writing's language.

2. When you click on the "View Results!" button, an error message will appear if one of the fields is not valid. In the opposite, if everything is valid, the button will change its state and the search process will begin.

Figure B.20: state of the "View Results!" button when the user clicked on it

Be aware of the fact that the number of requests is strongly limited by Twitter. Thus, you may not be able to make more than a few search per periods of 15 minutes, depending on the topics you searched. If the limit is exceeded, an error message will appear and you will not be able to perform search for the 15 next minutes.

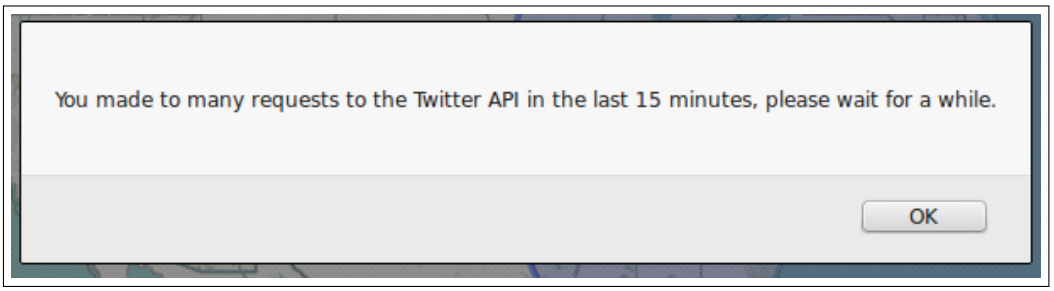


Figure B.21: the pop-up window that appears if the user exceeded the maximum limit rate imposed by Twitter

3. On the search successfully completed, you will be redirected on the results page of the static mode; everything is frozen and you can only access the data in a static way. There is currently no possibility to export the retrieved data from the static mode.

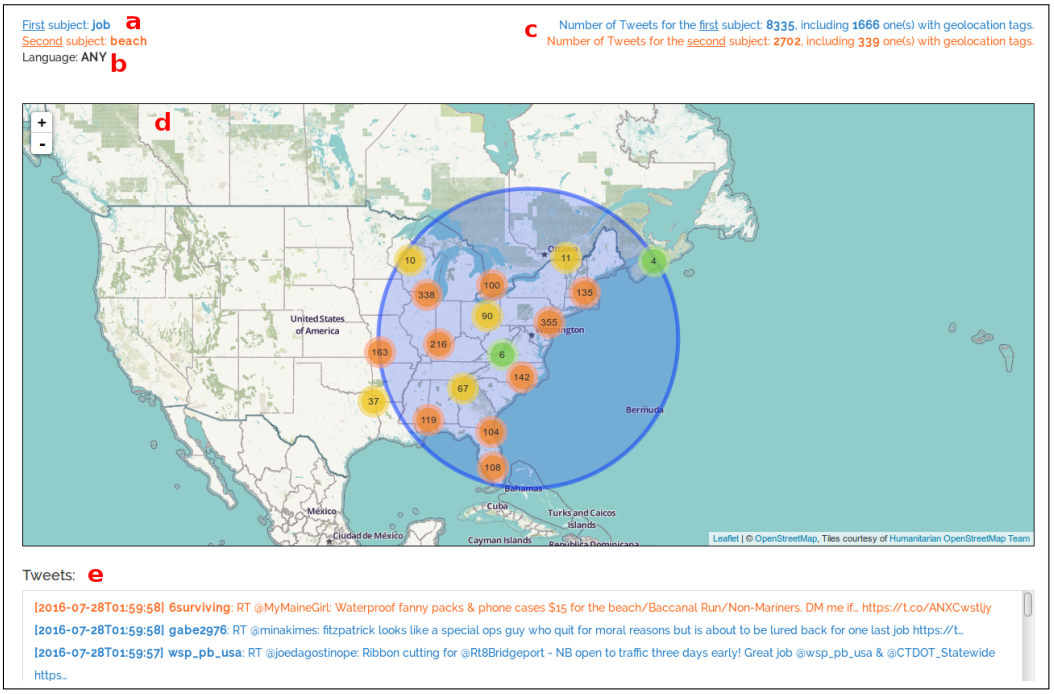


Figure B.22: annotated example of a static results' page with the "job" and "beach" keywords

Due to the complexity of the Twitter's algorithms, some retrieved tweets may be out of the drawn circle. Several information are available on this page, among which:

(a) **the human-comprehensive values of the keywords set / subjects**, identified by colors: blue for the first subject (always displayed), and orange for the second one (only displayed if you set the second keywords set);

(b) **the language** used to filter the incoming tweets;

(c) **the number of received geolocated and non-geolocated tweets** for each subject;

(d) **the map**, in which you can see the retrieved tweets grouped by clusters; once you zoom the map enough (a zoom level of 8 or more), these clusters disappear and you can access the tweets one by one. By moving the cursor over a marker, the related tweet's content is displayed, like in the dynamic mode.

(e) **a panel that contains all (both with and without geolocation tags) the retrieved tweet's content**.

4. By clicking on the "Make a new search" button, you will be redirected on the main search page.

Appendix C

Initial Specifications

C.1 Functionalities

The application will provide the following features:

- The reading of keywords and the selection of geographic areas on the map by the user.
- The retrieval of tweets, using the Twitter's APIs. Note that only a certain percentage of these tweets include geographic information, necessary for the future operations; a first filtering will thus be operated here.
- The analysis and the filtering of tweets via calculation of the number of tweets by areas and by subjects.
- The visualization of the results on maps.
- The interaction (zoom-in, zoom-out, etc.) with the maps. The development of this feature will involve the use of appropriate algorithms (like tweet grouping) and libraries.

C.2 Tasks

Here is a chronological list of the implementation's steps and achievements that will be achieved in this project:

1. Use of the Twitter's APIs, including:
    - Retrieval and filtering of tweets by one or more keyword(s).
    - Retrieval of geographic information attached to a tweet.
    - Analysis of the percentage of tweets having geographic information (necessary for further processing) and the search of keywords having relevant results, in order to determine the best keywords for use in order to test the different project features.
2. Use of the OpenStreetMap API as well as the "Leaflet" library in order to use geographic maps.
3. Development of the tweet's acquisition and analysis parts.
4. Development of the web interface of the application, allowing the user to enter subjects to compare, to choose the geographic areas on the map, to visualize results and to interact with the map.
5. Development of a grouping algorithm that groups the tweets according to the map's zoom level.
6. Testing and validation.

C.3 Used Technologies

The suggested tools for this project were the *Scala* language coupled with *Play Framework*, while noting other technological choices were also possible.

Due to both the personal interests of the student and a certain curiosity for unknown technologies in general, it was decided that these choices will be retained.

Lacking proficiency in either of these technologies, the student will initially be required to learn the Scala language and the use of Play Framework before starting the development on the application.

Note that different web languages (like *JavaScript* in particular) could be used to complement the client side of the application.

C.4 Future Extensions

Here is a non-exhaustive list of various extensions and improvements, which could be developed if time allows or in a potential future developmental update:

- Implementation of a user management platform (Sign-up, Log-in), in order to back up the different obtained results.
- Storage of the most successful keywords as well as the areas having the best geographic results within the application.
- Implementation of two modes of result display: a real-time updated mode, allowing the application to automatically receive and process new tweets in accordance with the filters (this feature will in any case be developed), and a static way, which could only display the results once without any further updating, and which could even simulate a streaming by analyzing the tweet's publication dates.
- Addition of social widgets (sharing of aggregated results, notes, etc.).
- Addition of a point based system (allowing users to accumulate points and trophies by the results rate of their search, implementation of user ranking, etc.).

Appendix D

Planning

D.1 Initial Planning

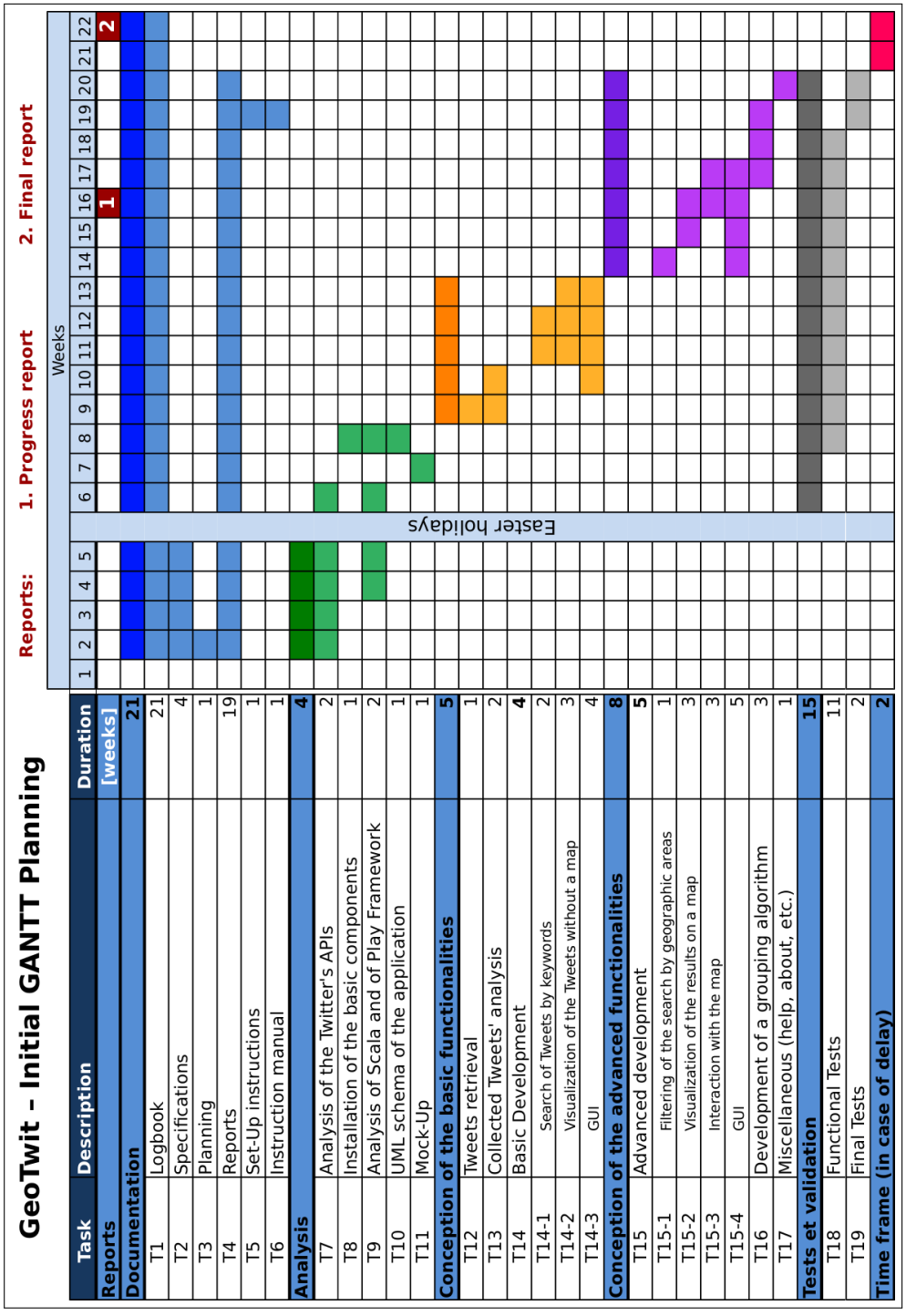


Figure D.1: initial planning of the project

This planning was largely followed during the first half of the semester (until the 17th week). From there, a rescheduling of the 5 full-time weeks was produced.

D.2 Second Half Planning - Rescheduling

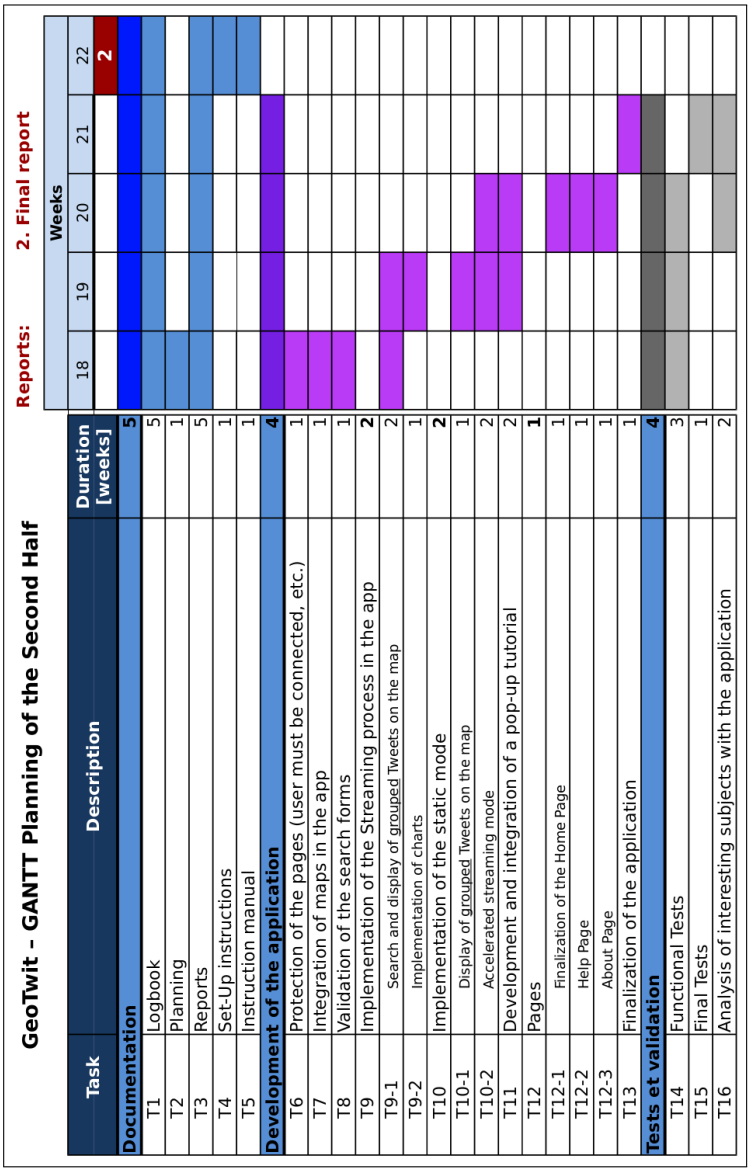


Figure D.2: planning of the second half of the project

This planning was also largely followed, except that the work was better distributed between the first three weeks and the fourth one, which was planned as a time frame (in case of delay).

Appendix E

Logbook

Week	Date	Hours	Task
1	22.02.2016	4	Conduction of various research on Scala and on the Play framework
2	01.03.16	0.5	Meeting with Mrs. Fatemi
	02.03.16	2	Writing of the specifications and the logbook
	04.03.16	2	Writing of the planning
	06.03.16	2	Writing of the planning and creation of the GitHub deposit
3	07.03.16	6	Analysis of the Twitter's API and of Twitter4J, and consequently writing of the analysis documentation
	08.03.16	0.5	Meeting with Mrs. Fatemi
	08.03.16	1	Analysis of the Twitter's API and of Twitter4J, and consequently writing of the analysis documentation
	13.03.16	1	Analysis of the Twitter's API and of Twitter4J, and consequently writing of the analysis documentation
4	15.03.16	0.5	Meeting with Mrs. Fatemi
	15.03.16	0.5	Analysis of the libraries usable for the geolocation as well as the interesting keywords for Switzerland, and production of a planning of news
	16.03.16	3	Analysis of the libraries usable for the geolocation as well as the interesting keywords for Switzerland, and production of a planning of news
5	21.03.16	4.5	Installation and production of tests on Twitter4J
	22.03.16	0.5	Meeting with Mrs. Fatemi
	27.03.16	1.5	Production of stats in the prototype application
	28.03.16	3.5	Analysis of various Twitter's research
6	04.04.16	0.5	Finalization of the specifications
	04.04.16	3.5	Analysis of libraries concerning the maps
	05.04.16	0.5	Meeting with Mrs. Fatemi
	06.04.16	1	Adaptation of the map vizualisation app for the Tweets' visualization

Week	Date	Hours	Task
	07.04.16	4	Implementation of a web-sockets system between the Java prototype application and the JavaScript one
	08.04.16	2	Implementation of a web-sockets system between the Java prototype application and the JavaScript one
7	11.04.16	3	Display of Tweets in the JavaScript application
	12.04.16	0.5	Meeting with Mrs. Fatemi
	17.04.16	1.5	Production of the application's mock-up
8	18.04.16	3.5	Production of the application's mock-up and analysis of the already existing applications on the Internet
	24.04.16	3.5	Analysis of the percentage of Tweets which have geolocation tags with various parameters
9	25.04.16	4	Analysis and research of interesting subjects and minor modifications on prototype applications
	26.04.16	0.5	Meeting with Mrs. Fatemi
10	02.05.16	3	Modification of the mock-up, according to the 26.04 meeting
	02.05.16	5	Development of a new prototype application, allowing a user to draw a polygon all around a country by double-clicking on it
	02.05.16	1	Translation of documents
	03.05.16	0.5	Meeting with Mrs. Fatemi
11	09.05.16	4	Reading of the Play framework's documentation
12	16.05.16	3	Reading of the Play framework's documentation and production of test-applications
	16.05.16	5	Creation of the application's basics and modification of the documentation
	17.05.16	0.5	Meeting with Mrs. Fatemi
13	23.05.16	2	Importation of the project in the IntelliJ IDE
	23.05.16	6.5	Integration of the Twitter4J library in the project, development of the connection process and modification of the GUI
	23.05.16	1	Modification of the documentation
	24.05.16	0.5	Meeting with Mrs. Fatemi
14	30.05.16	8	Development of the search page's GUI
	31.05.16	1	Development of the search page's GUI
	31.05.16	1	Translation of the analysis part of the documentation
	01.06.16	2	Translation of the analysis part of the documentation

Week	Date	Hours	Task
	02.06.16	1	Translation of the analysis part of the documentation
	03.06.16	1.5	Translation of the analysis part of the documentation
15	06.06.16	2	Translation of the analysis part
	06.06.16	0.5	Writing of the progress report's structure
	06.06.16	0.5	Translation of the specifications
	07.06.16	2	Creation and writing of the progress report
	11.06.16	1.5	Writing of the progress report
	12.06.16	3.5	Writing of the progress report
16	13.06.16	4	Writing of the progress report
	14.06.16	1.5	Writing of the progress report
	15.06.16	5	Writing of the progress report
	16.06.16	5	Writing of the progress report
	17.06.16	1	Writing of the progress report
18	27.06.16	1	Production of the GANTT planning for the second half
	27.06.16	6	Implementation of sessions in GeoTwit and protection of pages (the user must be authenticated, etc.)
	28.06.16	7	Implementation of sessions in GeoTwit and protection of pages (the user must be authenticated, etc.)
	29.06.16	3	Writing of the documentation related to configurations, routes, sessions, flash scopes, cache and actions compositions
	29.06.16	5	Integration of maps in the application
	29.06.16	1	Writing of the documentation related to the drawing of rectangle by drag-and-dropping the cursor on the map
	30.06.16	4.5	Implementation of a web-sockets system between the Play's server and the JavaScript client
	30.06.16	0.5	Meeting with Mrs. Fatemi
	01.07.16	8	Implementation of the streaming process in GeoTwit with web sockets
19	04.07.16	4	Writing of the documentation related to JavaScript routing and web sockets
	05.07.16	2	Writing of the documentation related to JavaScript routing and web sockets
	05.07.16	2	Reading and validation of user's inputs for the streaming process

Week	Date	Hours	Task
	05.07.16	1	Grouping of the Tweets on the map
	05.07.16	3.5	Selection of countries on the map with the drop-down menu
	06.07.16	0.5	Meeting with Mrs. Fatemi
	06.07.16	5	Selection of countries on the map with the drop-down menu
	06.07.16	1.5	Implementation of two streaming processes at the same time, when the user filled the two keywords sets
	07.07.16	0.5	Filtering of Tweets by language
	07.07.16	7	Display of streaming information (number of received Tweets, speed, etc.)
	08.07.16	0.5	Analysis of the FR-DE match of the Euro2016 event
	08.07.16	8.5	Implementation of the streamings' charts
	09.07.16	1	Implementation of the streamings' charts
	09.07.16	1	Handling of exceptions
	10.07.16	1	Writing of the documentation related to the analysis of the Euro 2016's final
20	11.07.16	7	Writing of the documentation related to the last week's implementation
	12.07.16	5	Reading and validation of user's inputs for the static mode
	13.07.16	0.5	Writing of the documentation related to the langages' selection
	13.07.16	4.5	Reading and validation of user's inputs for the static mode and retrieval of Tweets from the server
	13.07.16	0.5	Meeting with Mrs. Fatemi
	14.07.16	6.5	Implementation of the static mode
	15.07.16	4	Implementation of the static mode
	15.07.16	3	Generation of the backup files of the streaming's results
21	18.07.16	4	Generation of the backup files of the streaming's results
	18.07.16	4	Writing of the documentation related to the static mode and the generation of backup files
	18.07.16	1	Review of the documentation
	19.07.16	6	Review of the documentation
	19.07.16	2	Writing of the documentation related to the generation of the backup files
	20.07.16	0.5	Meeting with Mrs. Fatemi

Week	Date	Hours	Task
	20.07.16	3.5	Implementation of the about and help pages and modification of the GUI
	21.07.16	8	Implementation of the files' importation
	22.07.16	5	Implementation of the files' importation
	22.07.16	3.5	Writing of the documentation related to the files' importation
	23.07.16	2	Writing of the documentation related to the files' importation
	23.07.16	4	Implementation of the static mode
	23.07.16	2.5	Writing of the documentation related to the static mode
	24.07.16	3	Writing of the documentation related to the static mode
	24.07.16	3.5	Writing of the LaTeX document
22	25.07.16	14.5	Writing and review of the LaTeX document and writing of the tests
	26.07.16	10	Writing and review of the LaTeX document and writing of the tests
	27.07.16	15.5	Writing and review of the LaTeX document and writing of the tests
	28.07.16	10	Writing and review of the LaTeX document and writing of the tests
	29.07.16	5	Finalisation and report printout
TOTAL		**347.5**	