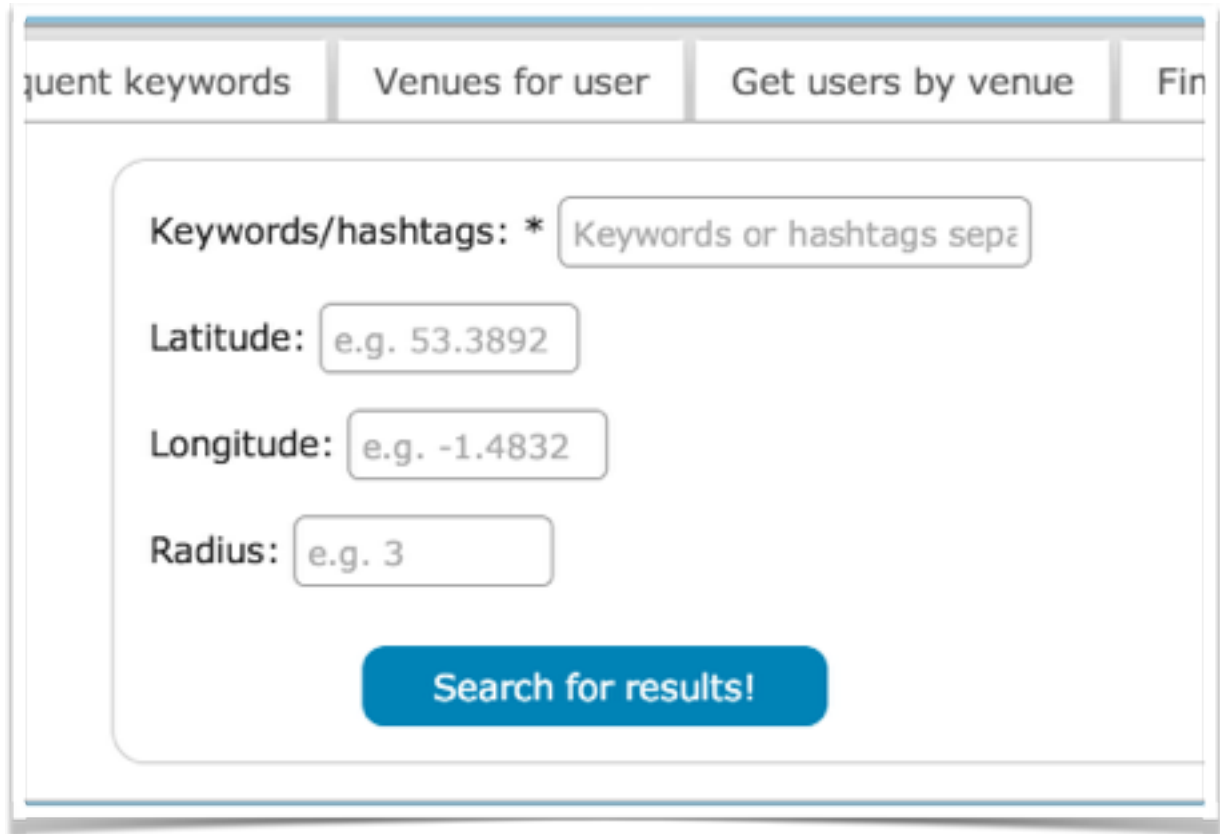


Intelligent Web Report

Accessing the Web and making sense of its content'



The screenshot shows a web application interface with a navigation bar at the top containing four tabs: 'requent keywords', 'Venues for user', 'Get users by venue', and 'Fin'. The 'Venues for user' tab is selected. Below the navigation bar is a search form with the following fields and a button:

- Keywords/hashtags: *** with a text input field containing the placeholder text 'Keywords or hashtags sepe'.
- Latitude:** with a text input field containing the placeholder text 'e.g. 53.3892'.
- Longitude:** with a text input field containing the placeholder text 'e.g. -1.4832'.
- Radius:** with a text input field containing the placeholder text 'e.g. 3'.
- A blue button labeled 'Search for results!'.

26th of May 2014

Team formed of

Florin-Cristian Gavrilă, Claudiu Tarta, Tudor Sirbu

Intelligent Web

Accessing the Web and making sense of its content'

Introduction

The assignment involves using various APIs to capture public data about people's discussions and locations. In order to complete the assignment requirements the Twitter and Foursquare APIs were used.

By using these APIs we managed to return:

- all the tweets containing a keyword and/or a hashtag within a certain radius of a given location;
- a top list of keywords used by a list of given users in the last X days;
- the venues where a certain user has been in the last X days or the venues where the user is currently checking in realtime;
- a list of users who have visited a given venue;

As the assignment requirements mentioned that additional features will bring extra marks, we managed to implement two additional features: one of them uses the Instagram API to retrieve more information about the users that post Instagram pictures on Twitter, and the second one lists the nearby venues within the radius of a given venue. All the data collected from Twitter and Foursquare is stored in a RDF Triplestore and the ontology was created as RDFs.

1.1 Querying the Social Web

Issues

One of the biggest issues faced at this stage was the use of the Twitter Streaming API. Our main challenge was getting the data in realtime from the servlet to the interface. Another issue was the retrieval of relevant information regarding who visited venues in a specific geographic area (part 1.1.2.C) as there is no direct way of querying Twitter and Foursquare for this information. Finally, another issues which came up during the implementation was the extraction of the keywords from a tweet's text. It was crucial to store only relevant words and not very common words like 'yes' , 'I'm', 'and', etc.

Design choices

In designing the solution for part **1.1.1.a** we have decided to query the Twitter API in order to retrieve the tweets that are relevant for the search parameters.[1] If some geographic coordinates are provided as well, the search is refined to match that area only. As part of the

input, the form takes a series of keywords and/or hashtags separated by spaces. The terms are then passed to the Twitter API which internally handles them and returns the relevant tweets.

For part **1.1.2.a** the form takes as input a series of maximum 10 Twitter user ids along with a number of keywords expected to be returned from those users' tweets. The search is even more restricted by providing the number of days to look in the past. The returned tweets are then stripped of unnecessary terms and characters (such as the RT tag, any hashtags, URL and screenames) and a count is performed to compute the number of occurrences of each word in that tweet. This information is then stored in the database. To display the top keywords, a query is performed on the MySQL database.

For part **1.1.2.b** the form takes as input a user id and how far back in the past to look. If the number of days is greater than zero, the system queries the Twitter API and retrieves the user's timeline. It then goes through the timeline and expands all the links in search for any Foursquare link. When a Foursquare link is found, the system call the Foursquare API [2] and extracts data about that checkin. In the end, all the data is displayed both as a list on on a map. In case the number of days is 0, a connection established with the Twitter Streaming API [2] and a listener is created which waits for any checkins performed by that user. All the results are then displayed using both a list and a map. [2]

For part **1.1.2.c**, if the name of the location is provided then a search is done with the Twitter API for tweets that match the exact input provided. This was chosen after a series of tests were performed and we have noticed that Foursquare automatically add the name of checked in venue to the text of the tweet. Otherwise, if the geographic coordinates are provided, the Foursquare API is called using those coordinates in order to get the top 10 most relevant venues closest to that point. After that, for each of the ten venues, an exact Twitter search is performed to get the tweets containing the name of the venues. (Similar to the case when the name is provided) This only done when the number of days if greater than zero. If the number of days is zero, then a connection to the Twitter Streaming API is established. Similar to the case when the system is looking back for checkins, now the stream is listening for tweets containing the venue names.

Requirements: all the requirements regarding what data is being returned have been, however, our solution could be improved by firstly searching through the system's database before querying Twitter or Foursquare.

Limitations: depending on how active a user is, we might or might not be able to return all the requested information. This also due to the fact that you can only retrieve a maximum of 3200 tweets from a user's timeline. Also, since there is not direct method of looking at who checked in in certain venue, just by looking for tweets that match that venue's name does not guarantee that we are getting all the results. **Extensibility:** since all of our functionalities were split into smaller tasks, these could easily used to implement additional features. For example, as we know where certain people have been at a specific time, our solution can be

extended to display data of how close certain people have been to each other at a given point in time.

1.2 Storing information

Issues

As requested in the assignment, all the storing has been done using RDF/RDFS and Jena for their manipulation. Most of the issues were related to the way Jena works, specifically the implementation of multiple namespaces and retrieving the data from the triple store. The issues were generally easily solvable, but very time-consuming due to the lack of examples found on the internet.

Design choices

The design choices for the storing part of the assignment are expressed through the ontology, which consists of three classes, `TwitterUser`, `FoursquareVenue`, and `Tweet`, which are enough in order to store the information required in the assignment. Due to the very specific classes in the ontology, external ontologies were not used extensively.

However, **`TwitterUser`** is a subclass of **`foaf:Agent`** and uses two of its superclass's properties: `foaf:name` and `foaf:depiction`. The choice for the **`foaf:Agent`**, and not **`foaf:Person`**, class was motivated by the fact that a twitter user is not necessarily a human being, it can also be a bot or organisation and the FOAF specifications specifically mention this case. Another external class used was the W3C's **`SpatialThing`** in **`FoursquareVenue`** and handles the latitude and longitude.

In terms of the code, the RDF layer is composed of two separate models, one for the ontology and one for the triples, joined in a union for the purposes of querying. Consequently, the model is split into two files, which is mean to keep the semantics and the actual data separate.

Requirements

The RDFS model is a good enough framework in order to contain all the information required in the assignment specifications, as we have tested most of it in the implementation.

Limitations

There is a possibility for the ontology to be able to include more external sources with which it can be linked. However, we were not able to find more appropriate superclasses than the ones that we have already implemented in our ontology. Furthermore, some of the the fields were named differently just in order to avoid a naming clash. For example, all the three classes have an id. A solution would have been for all them to subclass a class containing an id, but for such a small-scale project, we considered this was not necessary.

1.3 Web interface implementation as a Java Servlet

Issues

At the beginning we were confused as to where PHP could fit in the whole interface - servlet - database picture. We managed to solve this issue by using AJAX to perform requests from the Web interface to the Java Servlet and to receive responses from the Servlets. The responses were then processed using JavaScript. [2]

Design choices

All the forms we used are stored in the 'queryInterface.html' file as requested in the assignment. Each form is handled using AJAX which sends POST requests to the Java Servlets. All the requests and responses are encoded using JSON. The whole interface is in one file and the forms are managed using tabs. This resulted in a very easy to use and consistent interface.

The Java Servlets decode the JSON requests, process the data either by querying one of the APIs or by accessing the MySQL database and then encodes the information in JSON.

For the User Interface we also use the jQuery framework to process AJAX requests and to handle the validations on the forms. All the forms we have included contain validations.

1.4 Quality of the solution

Issues

Probably the biggest issue that we had was receiving a constant stream of information from the servlet to the interface. At first we did not quite know a proper way to have continuous flow of data from the Servlet.

Design choices

The issue mentioned above was dealt with by creating an AJAX request to the servlet every 5 seconds and displaying the data as soon as it is received.

We decided not to use PHP but rather a more modern technique (as presented in the lectures) using AJAX. The data exchange between the client and the server has been performed exclusively through JSON, except for when displaying information about a user from the database (i.e. when a user screen name is clicked a GET request is performed). Finally, the forms are validated using Javascript (jQuery).

Requirements: all the requirements mentioned in the assignment have been met.

1.5 Additional features

In order to enrich the user experience we decided to add two more features to the system. One of them allows the user to search for a venue using the Foursquare API, select the venue and look for nearby venues within a given radius. The second feature extracts images and additional information from tweets which contain a link to an Instagram post.

Issues

One of the biggest issues encountered when implementing the ‘Nearby’ functionality was selecting the right starting venue from the query results and display its neighbouring venues. This issue was raised by the fact that the same venue name can be found in multiple cities and/or countries. To solve the issue the system allows the user to type the name of the venue and the city, it queries Foursquare and returns a list of venues along with their address. The user then chooses the actual venue he meant and the radius after which the neighbouring venues are displayed.

Design choices

For the ‘Nearby’ functionality, the system is using the Foursquare API to find any venues within a radius of the given venue. The input given to the API is the location of the starting venue by providing its latitude and longitude. The API returns the neighbouring venues within that radius and the system displays the required information (name, photo, category, URL and a description) both on a list and on a map.

The requirements are being met as the venue details are being displayed and the venues themselves are being displayed on both a map and a list. **The limitation** is that the user can type a name which might not return the desired result from the Foursquare API.

For the second functionality which extracts Instagram links from tweets and displays the associated pictures, we have used the Instagram Oembed API. The system sends through a cross-domain request using JSONP the URL extracted from the tweet and the API returns a JSON object which contains the username of the person who posted the picture along with a link to the picture. Both are being displayed below of the tweet. This features **will not work** for posts which have privacy settings. This feature can be **extended** by getting the full details of the post which also include location and displaying the pictures on a map.

Extra information

We do not have any special requirements for our code to run.

Bibliography

1. Lecture from Week 2

2. Lecture from Week 3
3. jQuery Framework - <http://jquery.com/>
4. CSS Reset file (source at the top of the file)
5. Stoplist - http://nlp.cs.nyu.edu/GMA_files/resources/english.stoplist

Code usage

Within the implementation of our system we have used part of the code presented in the lectures. We have implemented the basic Twitter and Foursquare APIs connections using the code explained during the lectures. Also, the link expander and Google Maps API showed in the lectures have been used as well as the Gson encoder/decoder.

We have also used some external libraries like jQuery, Twitter4j and Foursquare-api.