Intelligent Web Applications

Final project
Group 2

Authors:
Laurens Verspeek
Student number 2562508

Tom Peerdeman
Student number 2559584

David Bernal Hoyo
Student number VU:dbo220
(UvA: 10860703)

Date: March 20, 2015

1 The Application

A person can come to a point in it's live where he has to choose a new or even an initial career. Finding a job you like however can be quite hard. Still many people managed to choose their jobs. Why not use their profile to choose for you?
This brings us to the web based application presented in this report. The initial idea of the app was to create a job recommender based on information from DBpedia. DBPedia contains at them moment of writing 50489 unique persons for which the job is correctly defined. Using this data the application creates a profile for each job.
The application can then be used by the user by simply logging in to Facebook. The app can then access some personal data from the users Facebook profile to create a user profile. Of course it is possible that some data can not be accessed from the app. This can be due to either a permission error, or the data not being available. In such a case the user is prompted to insert the missing data manually. Using semantic linking this profile can then be linked to the profiles created for the jobs. The user is then presented with the 5 top most matching job's. He can then click one of them and immediately be presented with some jobs available in his chosen category, around the users current location. These jobs can be fetched from LinkedIn or any other API providing this data.

After a few development days this idea unfortunately failed. The DBpedia entries of persons having a job defined were not as complete as assumed. For example 49958 of the 50489 people do not have a gender defined! The most reliable data properties were birth date and name. Both properties of course not very suitable for matching jobs. Another property was religion, which could be useful, but was unfortunately also not always available. The ones there were defined, were not standardized. For example "Christian" and "Christianity". A human would see that the same religion is meant, but teaching this to a computer is very hard, or even impossible. Due to this lack of information the app idea had to be changed.

The idea changed to give the user two possibilities. One of them being a list of the job clusters, as explained in the next section. When clicking on one of those cluster the user is presented with information about it. The other option is to choose one of the different categories of jobs like: awesome, crazy, fun, stupid, adventurous, chill or pays a lot. Then upon choosing a category the user will be presented with the available information of the jobs within these categories.

2 Datasets and Services

2.1 RDF store

GraphDB was chosen for storing all the information collected and generated in the application. Triples pushed into the store include the processed data from DBpedia and Twitter regarding job properties. It also contains the triples related to the user profile pulled from Facebook, and the additional info the user decided to provide at the login. If useful, the choices made regarding the choice on the job profile can also be saved.

GraphDB, also known as OWLIM, in particular was chosen since it was the only RDF store all team members had any experience with. It was also assumed that this store had OWL-FULL reasoning capabilities. GraphDB does have reasoning capabilities, but unfortunately does not support the full OWL-FULL set. The problem encountered is explained further in section 6.

2.2 DBpedia

The library sparqllib includes the necessary functions to make a request to DBpedia's SPARQL endpoint. The requested data was obtained through a CONSTRUCT query that returned triples with the job clusters, jobs and number of people by job. The job clusters are groups that contain similar occupations as defined by the *broader* property from SKOS over the value *Category:Occupations_by_type*, it was an important decision to use these clusters instead of the jobs by themselves, because it would be unpractical for a user to review the huge number of different jobs. Other desired information was the birth dates, gender, religion, nationality, education,... unfortunately, as it has already been mentioned there is no much information on DBpedia, basically most people only had few of the attributes needed, specially the ones that would be more useful like educational background, previous jobs and experience. The data requested was then limited to birth date, religion, birthplace and gender. Although still some of these properties where not available for many people. The information pulled out from DBpedia is used to generate some triples that relates the jobs with a certain value for properties we defined like iwa:popular. If the appropriate data was available this properties would have been the foundation for creating the job profiles. Another use of DBpedia data was to generate a graph that shows to the user the amount of people that has been working in a certain job cluster based on the birth date.

2.3 Twitter API

The app also makes use of Twitter API to get positive and negative tweets that mentions the different occupations. Another use of the API was to get tweets that mentioned certain keywords (like danger, fun, boring, death, stress, happy) on the same text that the job's name is mentioned. These with the purpose of providing more information related to each job. This information was also integrated with the different jobs by means of triples that linked the class of the job to the number of tweets by a property described by the keyword.

2.4 Facebook Graph API

To get personal details from the user some data of the Facebook profile information is requested by means of the Graph API. This data includes the name, educational background, Political views, Relationship status, Religion, Employment history, Interests (from likes), Inspirational people (from likes). These data will be used to match the user to some profile in one of the job categories. It turns out that Facebook also limits the amount of information a user can accept to provide from a Facebook account, unless a special Facebook permission is granted or the user is defined as a developer or tester by the creator of the app. This permission is only granted after a extensive review of the app by a Facebook employee. Since we did not have a working app requesting this review would be usesless. So the information retrieved from Facebook was only name, ID, location and birthday.

2.5 Data registered by the user

The user is able to expand the information on its profile by filling in some fields where he can specify which categories he or she find more related to.

3 Functionality of the application

Upon an entry of a new user the application asks to get permission to the Facebook profile information. If the user accepts to disclose the information, it is pulled through the Facebook API and displayed for the user to check it and fill in any more details that could help the app find a better match for a job profile. After submitting the personal profile the data is processed to generate some triples on the user that are stored in the RDF store. Meanwhile the application displays on a page links on each of the job cluster for the user to explore. Upon clicking on a job different information about the people that work on those occupations appear on the page including name, date of birth and a plot on the number of people that worked on the professions through time. The different categories the cluster belongs to and the similarity ranking with the user profile are also shown.

Another usual work-flow would consist on a user browsing through the descriptive categories. Upon choosing one, the user would get the job clusters belonging to that category ordered by the matching punctuation with regard to the users' Facebook profile.

4 Inferencing

The first semantic rules are over the property coworker. This property is used to relate all users that choose the same job profile, so that they can know what other people are interested in the same field. Two rules are defined for this property, that it is an owl:symetricProperty and owl:transitiveProperty.

There are several categories that describe the job clusters. The other semantic rules were defined referring to the relationships among these categories and to the memberships that the different occupations/users have in them. For example a restriction property was used so jobs in the intersection of Fun and Rich are also members of Awesome.

Another rule was defined to work the other way around, so if a job was Awesome then it should also be Fun and Rich. This was accomplish by defining Awesome as a subclass of Fun and Rich.

Furthermore one of the basic reasons for a job belonging to one of the categories is by means of an inference using a property restriction, for example, a job that has a value of 100 for the property iwa:danger then it belongs to the class Danger.

The inferred information is presented to the user by displaying the jobs that belong to the different categories.

5 Implementation

The language used to program this application was PHP, since it is popular among web programmers, easy to use and there are lots of built-in functions and libraries available. Also because, except for David, the team already was familiarised with it.

EasyRdf library

The app interacts with a remote SPARQL endpoint, namely DBPedia, and the local RDF store, therefore a method had to be created to interact with both of them via PHP. Interacting with the endpoints themselves is quite easy, a simple HTTP call with a text based SPARQL query is enough. This can be done in PHP via the cURL bindings [1]. The problem is the return values, or better said the format of the return values. There is no widely accepted universal standard for serializing RDF triples. Instead a couple of standards exists, namely turtle, RDF/XML JSON-LD and many more. Creating a parser for all these formats would be wasting time, and therefore a library was used. Initially a library called sparqllib [2] was used, as mentioned before in section 2.2. This library has capabilities of executing SPARQL queries and fetching the result as a simple PHP associative array object. The downside of this that it cannot be used to execute INSERT, UPDATE or DELETE SPARQL queries, as it is a very simple library and therefore does not know what to do with these queries.
Since we did want to insert data, otherwise a local RDF stores would not be needed, a different library was used. This library is called EasyRdf [3]. The EasyRdf library has a SPARQL client that can send query's to a server, and an interface for a remote graph store like the local RDF store. The results of both are parsed and returned as a Graph, a very basic RDF triples representation than has convenient methods for retrieving the triples. Both interfaces also have the required methods for inserting data.
One can see that EasyRdf can do everything that sparqllib can do, and more. However due to time constraints sparqllib was never replaced by EasyRdf in the at the time existing code.

Unfortunately using EasyRDF also has its downsides. One problem we encountered was the use of the JSON-LD format. When querying the DBpedia endpoint the EasyRdf SPARQL client gives the DBPedia endpoint the RDF formats it can understand, with a given priority. The top priority it gives is JSON-LD. The DBPedia endpoint of course can understand JSON-LD as well, so happily returns the results in JSON-LD. The EasyRdf library itself can however not parse JSON-LD at all, even though it claims so. It requires a secondary library to parse JSON-LD, and either this library does a bad job, or the JSON-LD standard is flawed, as incorrect results were returned. Some URI's were converted into string literals, which of course is terrible of you want to use those URI's to infer some other data.

The solution of this problem was to remove the JSON-LD support from the library by editing the supported formats. The next preferred standard, turtle, was in tests unfortunately quite slow to either fetch from the server or parse. Therefore the preferred standard was forcefully set to RDF/XML.

Another problem was caused due to the usage of GraphDB. The EasyRdf library can send an SPARQL DELETE statement via it's update method. It then sends the SPARQL query to the endpoint using an HTTP POST call. It sets the Content-type of this call to application/sparql-update to indicate to the server that this query will alter the data.
GraphDB is based on the openrdf sesame framework [4] to handle the SPARQL queries. This framework however does not understand the application/sparql-update type and simply returns: I don't know this type, and doesn't execute the alteration of the data based on the query. To solve this a small part of the EasyRdf SPARQL client was rewritten to send the query as urlencoded data instead. It sends an HTTP POST call with Content-type application/x-www-form-urlencoded, and as data it sends update={the query as urencoded data}. The openrdf sesame framework can understand this and does execute this query.

6 Results

The application basic services are working, this means that the SPARQL endpoint is working, the Twiter API can retrieve tweets, Facebook API can retrieve the profile information and the RDF store is online and getting triples from the application. The reasoner is also working, so that the inferences using OWL are realized. Although there was a problem with OWLIM restriction to support integer properties as maxInclusive, minExclusive [5], which limited us to use certain restrictions on properties regarding the number of tweets.

The processing of the information is also working, so that we gather all the info from the different data sources, create certain triples, plot a graph using DBpedia info to show the amount of people working on an occupation, displayed available photos of the people (instead of names and birthdates), show the average birth date and location, convert the JSON from Facebook to get the user info, etc.

The implementation of the matching system was not done since there are no matching properties between the data sources and the user input is also missing. The coworker property is in operation although it is quite useless since we are the only users of the app at the moment. The other semantic rules are also implemented.

The functionality and value of the information provided by our application is very questionable. As mentioned on the beginning of the report the first purpose of this application was to be a job recommender, but due to the lack of information on the DBpedia dataset that idea was doomed. Therefore we had to come up with an alternative in a small amount of time and that's how the idea came up of gathering additional information from an almost infinite source,i.e., peoples' opinion via Twitter. Naturally this decision comes along with inconsistent, ambiguous, untrustworthy, subjective and maybe ridiculous information that doesn't leave the application a chance to preserve its formality and utility as a job recommender. It has to be said though that it establishes a foundation to start building up a vocabulary around jobs and also to experiment on semantic rules.

Unfortunately a very important piece of code for the correct workings of the app could not be implemented. Each time a job is clicked the data is retrieved for that job. This is a good thing, because it keeps the data up to date. This also means that there can be duplicates, or old out of date data staying behind in the local RDF store. As written in section 5 an effort was made to execute a delete query. Code was created that could delete the data related to an occupation in the local RDF store so that new data could take its place. However due to time constraints this code was never extended to the Twitter and Facebook data. This code was also never merged with the fetching code, so it is never called, and thus the problem remains.

7 Possible improvements

Search for a more trustworthy source of info to create more useful categories. Also to get another source for information on the user side and implement the ability to let the user fill in some data upon registering to the app.

Another possibility would be to change completely the purpose of the app towards a semantic tester,letting the user test some semantic rules (maybe very difficult) and define new categories by establishing parameters for a Twitter search and for the rules used on the restriction property.

References

[1] Client url library. http://php.net/manual/en/book.curl.php. Accessed 19-03-2015.

[2] Christopher Gutteridge. Sparql rdf library for php. http://graphite.ecs.soton.ac.uk/sparqllib/. Accessed 19-03-2015.

[3] Nicholas Humfrey. Easyrdf - rdf library for php. http://www.easyrdf.org/. Accessed 19-03-2015.

[4] Sesame. http://rdf4j.org/. Accessed 19-03-2015.

[5] Does owlim supports data type reasoning? http://ontomail.semdata.org/pipermail/owlim-discussion/2010-September/000455.html. Accessed 19-03-2015.