Team Project Report

—

"Localizr"

Course Semantic Web Technologies, HWS 2014

Prof. Dr. Johanna Völker, Daniel Fleischhacker

presented by
Anna Primpeli
Martin Pfannemüller
Maximilian Böhm

Contents

1 Introduction

Localizr is a web application which displays available information of a specific city depending on a user's context. Users decide on a category, for example "Tourist" or "New citizen", or set up an advanced search by defining what they are looking for, e.g. airports or libraries. Afterwards they set the city by entering the name or they allow the application to use their GPS coordinates.

As a result of this search, our application displays information which are aligned to the users needs. There will be general information, point of interests, events and photos. There is also a map which shows the selected city and markers corresponding to the retrieved information.

1.1 Project Motivation

In recent years a lot of companies and public authorities published their data. If we speak about this data, we refer to it as Open Data. There are several portals [7, 11] sharing these information. Most often, these portals are data silos with no inter-linking. A lot of information is available, but the datasets do not have a connection to others. Linked Open Data, abbreviated with LOD, solves this topic. But there is still one major issue, it's hard for users to browse this data because it is not human friendly readable.

There are programs called Semantic Web Browser [38] which solve this readability issue. But they aim to be a general problem solver. Therefore they cannot address the needs of specific domains. We concluded, that each domain needs an own implementation of a reader. Localizr is such an implementation.

Localizr uses the very specific domain of geo exploring. This means, somebody provides a location and Localizr will exploit the available information and show it to the user. The way the data is presented should be as coherent and reasonable as possible. The aim is to understand the semantic character of data and use it for appropriate presentation.

1.2 Usage Scenario

1.2.1 Tourist

Think of a tourist. Maybe this tourist is already in a specific city and wants some information about this place. Let's call this tourist Johanna. She has heard of Localizr through another traveler via Facebook. She visits the homepage of Localizr (figure 3) where she can choose of several categories. As she is a Tourist, she chooses the corresponding "Tourist" category. The application detects the availability of the GPS functionality of her mobile phone and asks if it is allowed to use it. Because she mainly wants a fast overview, she permits the access. All in all, she just has to choose a category and release her GPS coordinates. Localizr only needs this data to create a comprehensive overview.

After submitting the search, Johanna will get a nicely designed result page, see figure 5. On the left side a map is shown which contains a lot of markers which refer to information displayed on the right side. These informations are split on four tabs: "General Information", "Point of Interests", "Events" and "Photos". She can just click on a tab and the content will adapt according to her choice. The information retrieved in Point of Interests and Events is aligned to her previous choice

"Tourist".

Johanna is now able to explore some facts about the city as well as events, photos and a lot of different points of interest, for example museums, libraries and much more. All these information are clickable, so it will be either opened a page at wikipedia explaining the entity or the map will scroll to the according location of the entity.

Additionally, if Johanna opens a specific point of interest, Localizr will look for other, related point of interests which could also be interesting for her, and show them to her.

1.2.2 Student

We also can think of a use scenario for a student. Imagine a student called Daniel. He deliberates about whether to study abroad or not. Therefore he has a list of possible universities. He doesn't know each city and its location. Instead of just searching for each city, he uses the advanced search of Localizr . He chooses Airports (He is afraid of being homesick), Shopping Centers (It's always good to know where to shop), Universities (To locate the university), Nightclubs (He likes to party) and Lighthouses (His dream, he loves lighthouses). He went through his list and finally explored Plymouth as his favorite because of its lighthouse.

2 Application Design

2.1 Software Architecture

Our application follows the Model-View-Controller pattern as it is shown in figure 1. The controller processes events, typically user actions like requesting a specific page on the web. It does so by extracting relevant parameters from an incoming HTTP request and apply this information to the required models. Afterwards the view renders the result in a suitable form of presentation, for example JSON or HTML [40].

Figure 2 shows the concrete implementation of the Model-View-Controller as it is done by Play. Play is the framework we use for our application. There is a file called "routes" which connects the controllers with incoming requests. Usually this is done via the pattern of the URL. An example could be that "/products/computer" would be mapped to the controller via this file. The controller extracts the name of the product ("computer") and retrieves the model which belongs to a computer. After the model is retrieved, it will be put into the view which will generate an appropriate view [40].

2.2 Use of Semantic Web Technologies

In order to enrich our application with structured information we make use of semantic web technologies by querying data with the SPARQL language from different SPARQL endpoints, exploiting the structure of the data and combining information from multiple datasets.

More specifically, as far as the SPARQL endpoints are concerned, it is required to embed two of them in our project in order to retrieve the necessary range of information. We extensively utilize Virtuoso SPARQL endpoint (see [9]) from which we are able to access DBPedia and EventMedia datasets as well as FactForge SPARQL service (see [14]), which allows us to query the GeoNames (see [36]) dataset. For the purpose of creating our Linked Data Java application, it was necessary to use a suitable Java library. We decided to work with the Apache Jena library (see [2]) and its SPARQL processor, ARQ, "a query engine for Jena that supports the SPARQL RDF Query language". The above mentioned processor simplifies the transformation operation of RDF resources and literals into user presentable information, by mapping every SPARQL query result into a model of resources as subject, properties or object and literals as objects which can be then further parsed.

Additionally, taking advantage of the possibilities semantic data does offer, we exploit their structure in two different ways. Firstly, we link at most cases every URI to its relevant Wikipedia article which is offered by the foaf:isPrimaryTopicOf or prov:wasDerivedFrom properties. In this way we do not only present to the user facts about a real world entity but we also give them the possibility to navigate through more information concerning it. Secondly, we utilize the hierarchy of types of resources so as to create a recommendation system for points of interests. More concretely, we focus on the second hierarchical level by querying for the super type of each entity that describes a point of interest. After considering that transitivity of the property rdfs:subClassOf cannot be avoided we developed an algorithm to prevent type duplicates, ending up with all distinct super types of level two. These super types are the foundation for our recommendation system by providing the user points of interest that are more general compared with their initial choice but

that are also grouped into the same super type, thus ensuring some degree of relevancy.

Making use of semantic web technologies evoked certain problems that are worth mentioning in this point. A major problematic issue was the frequent unavailability of Virtuoso SPARQL endpoint due to maintenance reasons. As a consequence, during those periods of time our application was not able to retrieve and thus to provide information. In addition, we were also faced with the situation where the utilized SPARQL endpoints could not supply us with the requested data within a reasonable time limit. This happened mostly when the processed query had to be long and parse different types of semantic data. We gave a solution to the above mentioned issue by splitting the "harder" queries into simplified ones which we then parsed separately.

2.3 Datasets and Ontologies

Localizr uses the following four semantic datasets in order to serve its purposes.

1. DBPedia
2. EventMedia
3. GeoNames/Freebase

Below follows a description of the used datasets and ontologies involved.

2.3.1 DBPedia

The DBPedia dataset is the result of extraction of structured information from the infoboxes of Wikipedia thus being a very rich dataset able to give information on different domains [5]. We use DBPedia in order to retrieve information about the city the user is located in and also to provide the available Wikipedia links for most real world entities used in our application. This applies e.g. for points of interest. In Figure 6 you can see the different information extracted from DBPedia concerning each city which is shown in the General Info tab of Localizr. The different ontologies used are marked with separate colors.

As it can also be inferred from the graph, four different ontologies used by the DBPedia dataset are relevant for our application. RDF Schema provides mechanisms for describing groups of related resources and the relationships between these resources (see [37]) and in our case is used for retrieving information about the labels of the different resources as well as a short description of them. The DBPedia Ontology, abbreviated with dbpedia-owl, holds cross-domain information that derives from the Wikipedia infoboxes (see [8]).

For our specific purpose, we focus on the Place and Person domain of the ontology. In addition, the GeoNames Ontology (see [20]), abbreviated with geo, is used so as to get geospatial semantic information of a place, i.e.: latitude and longitude. Finally, the usage of the DBPedia dataset for general information extraction required the consideration of the FOAF Ontology (see [6]) of which we utilized the core terms depiction and primaryTopic as well as the social web term homepage.

It is worth mentioning that not all of the above dbpedia-owl properties were available for every city which was the reason why we use similar properties in a complementary way. The result presented at the user is based only at the available information and for this reason the general information presented for different cities may vary in terms of the number of attributes provided.

2.3.2 EventMedia

We select the EventMedia dataset in order to extract relevant information about events that take place in the city the user has selected or has been located via GPS. As mentioned before, five categories of different types of events are constructed for the purpose of tailoring the information based on the category group the user belongs. In the case that the user selects the advanced search, they receive by default information on events mapped to the tourist profile, as it is considered to include the widest variety of event information. Table 1 shows which event categories are mapped to the five user profiles.

Figure 7 depicts the information extracted by our query for each event. The different ontologies used are marked with separate colors.

Our interaction with the EventMedia dataset requires the comprehension of three additional ontologies. Firstly, LODE Ontology (see [31]) is an ontology for linking descriptions of events. Events in LODE Ontology are reported past or future occurrences. For our purposes we focus on four properties that an instance of the class lode:Event may have, as depicted in the graph above. The range of the property lode:atPlace is dul:Place, while dul is the abbreviated prefix for the DOLCE+DnS Ultralite Ontology
(see [19]). Finally the vCard Ontology (see [39]) describes people and organizations while in our case it gives us access to location information about a place where an event occurs.

A defined problem with the EventMedia dataset is that the events were mostly outdated. More specifically from the 147.403 total events of the dataset only 168 future events were found. This leads to a percentage of 99% outdated events, based on our query at the time being. Even though it is considered unlikely that a user would search for past events, we decide to include the EventMedia dataset in our application as it may be updated in the future.

2.3.3 GeoNames/Freebase

We use collaboratively the GeoNames and Freebase datasets as it is possible to query both of them from the same SPARQL endpoint (See [13]). From these two datasets we extract information relevant to points of interest located in a certain radius around the user's location. GeoNames dataset offers location data concerning the retrieved points of interest while freebase is used to categorize them. Based on this, we are able firstly to map those locations into our embedded map and secondly to present tailored information based on the profile or the advanced search criteria that the user has chosen. As mentioned in the previous section we exploit the super types of the point of interest the user chooses in order to offer recommended locations grouped under the same super categories. Supplementary, the DBpedia dataset is used to link each point of interest to the relevant Wikipedia link. Figure 8 depicts the information queried and presented for each point of interest.

The additional ontologies that are utilized in this case are the Freebase Ontology, abbreviated with fb and some extensions of the GeoNames ontology, abbreviated with omgeo. The Freebase Ontology in our case helps us categorize the points of interest into multiple information groups. The nearby feature from the GeoNames ontology gives us access to location in proximity of a specified geospatial point (see [20]).

Finally, we dealt with an additional problem while retrieving information about places of interest. The problem is focused on the fact that same real world entities describing a point of interest may have different geospatial information assigned, thus returning more than one pair of values for latitude and longitude. However as we wanted to avoid taking a sample value out of the returned pairs and after taking into consideration that our results are not majorly affected, we chose to leave this problematic issue for possible future work.

3 Implementation

3.1 Backend

As already mentioned in Section 2.1 we use the Play framework (see [41]). Play is a Java framework for creating web applications. Since it uses the MVC design pattern in the following we describe the controllers and the model. The view is described in the next section.
Localizr uses two controllers. One controller managing the start and the results page, and another one that creates JSON output for the recommendation system. The application controller serves either the start page or the results page. If the (get) parameters we use are empty, the start page is served. Otherwise we call the datasets and serve the results page. The result page can either be opened by entering a string (e.g. a city name) or by using the browser based localization. This is done by creating so called promises for every list of result type. This means that the application runs asynchronous. Thus, we have no problems with a blocked user interface. The controller also features error handling. If a dataset returns an error (for the problems see Section 2.3), the result of this dataset is initialized as an empty object. Additionally an error flag is set to true, so we can show an error message in the results page. As of the high number of requests we have (Google Geocoding API Query, SPARQL requests, Panoramio API request, translation of event addresses into coordinates) the response time is relatively high. For this reason we use the caching functionality provided by the Play Framework. This enables us to save key/value pairs containing the requests as keys, and the results as value speeding up the response time on equivalent requests. The caches are saved as long as the application runs.
When we come to the data model, we created a model class for every result type. This means, we have image, poi and event classes. These models store the data we want to display and the data we need for processing. E.g. the poi object has properties for the resource URI, the label, the link to wikipedia, latitude and longitude. Based on these model classes we have custom list objects for each result type. We need these custom list objects since a promise in the play framework has to be of a generic type. So we have basically three different promises: one promise of type "POIList", one promise of type "PhotoList" and one promise of type "EventList". In order to populate these we use an additional class for every result type. These classes populate the list object and return them to the application controller. This means these classes create and run the result specific SPARQL queries to populate the lists. For parsing the RDF results of the SPARQL queries we use Apache Jena (see [2]). Since we use the Panoramio API for retrieving images, there is no SPARQL query in the class populating the promise of type "PhotoList".
Dependency management in the backend is done with the Scala Build Tool (see [35]) which is the dependency manager used by the Play Framework. The used dependencies are Apache Jena and Gson (see [22]) for serializing java objects into JSON. How exactly Gson is used is described in the following section.
We have two views in the application: the start page view, and a results view. The views are represented as Scala/HTML-Templates. The frontend containing these views is described in the following section.

3.2 Frontend

For the frontend we use several open source dependencies. At first this section outlines the used dependencies following the order of occurence when using Localizr.
The start screen seen in Figure 3 mainly uses bootstrap, font-awesome and custom CSS. Additionally, we use Fancybox (see [33]) for the "Advanced Search" button (see Figure 4 for the result). This opens a model view containing all possible categories that can be used as search parameters in case you do not want to select a predefined group of categories represented by the presets. These categories directly represent the names of the freebase categories (see section 2.3). Then, you can either enter the name of the place you are right now (e.g. the city name and the name of the district) or use the automatic browser localization based on HTML5. When entering a name we use the Google Geocoding API [21] in the backend to transform the name into coordinates. After sending the request to the backend we get a response containing the results page.
As seen in the screenshots in the appendix the main part of the results page is used to display a map view. We decided to use leafletjs (see [1]) here. Leafletjs is a map view that is based on javascript. Leafletjs is only the framework to create a map view. Thus, you can freely choose an available map layer. We use Openstreetmap (see [12]) as map layer. The map is useless without markers corresponding to results. In Leafletjs it is easily possible to add markers based on their GPS coordinates. Additionally, there are several plugins available. In Localizr we use the Leaflet.markercluster (see [27]) plugin to achieve clustering of the markers. With custom code we also make it possible to filter the three types of markers we display in the map. All this helps to improve performance and usability. As you can see in Figures 9, 10 and 11, we created custom marker images for every type of marker. Since points of interest and images already have information about coordinates, it was easily possible to add corresponding markers of these result types to the map view. The location of events is only represented by an address. At first we tried an own approach translating addresses to coordinates but we ended up using geocoder-js (see [29]) for this purpose. Again, you can freely choose a service provider for this translation. Here, again, we use Openstreetmap as translation service. Openstreetmap was easily integrable since they do not have a threshold determining a maximum number of requests per second. Since we have to translate the addresses of all events we have to create a request resulting in the corresponding coordinates for every event. This results in many requests. In order to synchronize these asynchronous request we use AsyncJS (see [28]). This enables us to have on central callback that gets executed when all translation requests are done. We create an array of prepared request functions that get executed in parallel. For displaying the images we use Justified.js (see [23]) which is a JQuery Plugin. Additional general javascript libraries used here are, obviously, JQuery (see [26]) and underscorejs (see [3]). All frontend dependencies are managed with Bower (see [34]).
The templates are mainly based on HTML with Scala placeholders. The Play Framework utilizes Scala as template engine. Thus we use Scala variables as placeholders in the templates. Scala enables us to use for-each statements to present all results conveniently in the resulting HTML site. Also, you can use control structures like ifs in Scala templates. We also use this e.g. to show possible errors when querying the datasets. We also use the placeholders to pass JSON encoded results to the javascript code. For the serialization we use Gson (see [22]). Gson serializes all kinds of java objects into the JSON format. Thus, we can easily access the results in the javascript code.

Inside the javascript we use these arrays of the result types to create a marker for each result of every result type. We add the created marker to each corresponding result. This makes it possible to refer to each marker of each result. Event markers are added after the address translation for each event is finished. All markers are added to one unique set of markers in the map view representing the set that gets clustered. This means, all markers together are clustered. In the javascript we also set up the method for resizing the sidebar, scrolling the map, and initializing the tabs and the map. When you click on one point of interest, an address of an event or the "Show in Map"-Link of an image in the sidebar of the results page, the index of the clicked element is passed to a javascript function. This function scrolls the map to the marker determined by the index and opens the marker popup.

4 Project Management

4.1 Work Packages

Table 2 shows the work packages together with subtasks and developers working on the package. As you can see we evenly divided the tasks. In the following we describe the work packages in more detail.

Since we freely could decide what to develop, clearly the first step was to find a good idea. It was our intention to develop something useful which does not exist. There should always be the opportunity to publish our work. Also, we all agreed on developing something location based for the reason that it is exciting and interesting. Another reason is that nobody has ever worked before with location based development. We also decided to build a web application to have the possibility to use it on every device.

When we found an idea, we had a look at several (web) frameworks. One requirement was the availability of a library that handles RDF and SPARQL. Apache Jena [2] seemed to be the most sophisticated solution. This made it necessary to use a framework that is Java based. We did not want to use Java EE for the reason that it would be definitely too much of a framework for the expected size of the project. In the end we decided to use the Play Framework [41].

Package 3 was to setup a project environment including IDE integration (we used Eclipse [10] and IntelliJ [25] here) and a repository (we used Bitbucket, see [4]).

Then we built the basic data model and tested it with simple SPARQL queries.

The next package describes the development of the controller. This made it possible to fill the data model on behalf of an HTTP request.

Since for the moment we only got all data based on very simple SPARQL queries, the next step was to refine the queries. The idea here also was to have several presets of categories to customize the search query. In the end this work package contained the idea to have some kind of simple recommendation system based on SPARQL queries.

Package 6 is about the map in the frontend. As of Localizr being location based, a map is an important user interface element. Here we had to add (custom) markers to the map view. Additionally it should be possible to click on a link in the result list to scroll to the corresponding marker in the map view.

The last package determines the general frontend design and the image view based on Justified.js.

4.2 Work Schedule

When working on the project we used a very simplified SCRUM development approach. Thus, we worked on work package 4 to 7 almost the whole time, refining every package with every version. We met weekly to talk about our last accomplishments and what was to do next. Everybody got a new task from the work packages every week.

5 Conclusion

Localizr has evolved to a great product. Although we do not see a possibility to charge money, we are proud for what we have created. During development, we focused on conceptualisation and implementation. What to do now, what to do next, how to satisfy the requirements?

But after having the jobs done, Localizr grew to something great. You want to go abroad? You want a lot of information? And not just displayed on your screen (as many others do it) but also linked to a map. You are not sure in which part of a city you should rent your flat? Localizr can help you in your decision process.

Beside creating an innovative product which is a lot of fun, we have learned a lot. Evaluation and decision on a framework is not that easy. Also you have to stick to your decisions and live with the consequences you have made. The 'Play' framework was a great choice as it simplifies a lot of tasks from development to deployment.

By using linked open data you get access to a huge datasets with meaningful data. Sites like Datahub (see [30]) provide an overview of a big number of datasets. Our choices regarding datasets are well-founded but there is still room for improvement. Generally we love the idea of linked open data but we also experienced some drawbacks. Endpoints are often not reliable (DBPedia) or the dataset is not up to date (EventMedia). We were aware that EventMedia is out of date but because of a lack of alternatives we still have chosen it.

Despite of the problems we faced, we were capable of creating a great project.

A Appendix

A.1 Tables

User Profile	Preset Categories
Tourist (Default)	Musical Concert, Visual and Performing Arts, Festivals, Other and Miscellaneous, Comedy, Performing Arts, Media and Literary, Technology, Nightlife, Museums and Attractions, Food
Family with Kids	Family and Kids, Education, Animals
New Citizen	Musical, Social Gathering, Visual and Performing Arts, Concert, Festivals, Community, Other and Miscellaneous, Media and Literary, Comedy, Performing Arts, Nightlife, Fundraising and Charity, Food
On Business Trip	Commercial and Sales, Politics, Technology, Professional, Conferences and Tradeshows, Organizations and Meetups, Business and Networking Science
Sport Fanatic	Sports, Outdoors, Recreation, Health and Wellness

Table 1: Mapping between user profile and categories

#	Work Package	Subtasks	Developers
1	Finding an idea	-	Anna, Max, Martin
2	Evaluating possible frameworks	Try some frameworks, check documentation	Max, Martin
3	Project setup	Setting up Play project, git repository and IDE integration	Anna, Max, Martin
4	Model	Creating data model, fill data model with results of SPARQL query	Anna, Max, Martin
4	Controller	Create controller, add error handling, start requests	Max, Martin
5	Advanced SPARQL queries	Improve SPARQL queries, add categories, recommendations based on clicked points of interest	Anna
6	Frontend: Map and markers	Add map view, add markers to map, scroll to markers and open popup	Martin
7	Frontend: General design and images	Creation of the general template design, create imageview	Max

Table 2: Work packages, subtasks and developers

A.2 Images

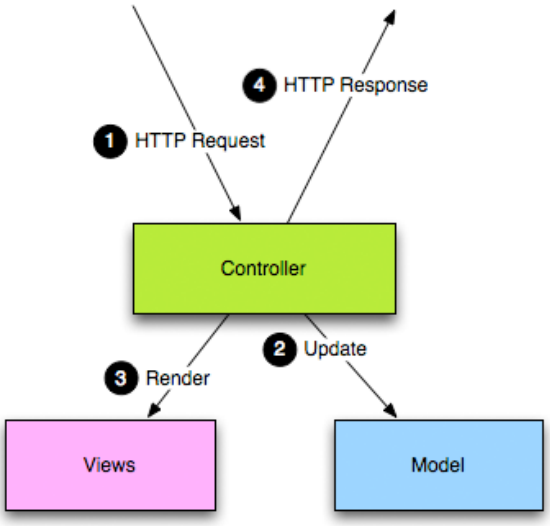

Figure 1: Model View Controller Pattern [40]

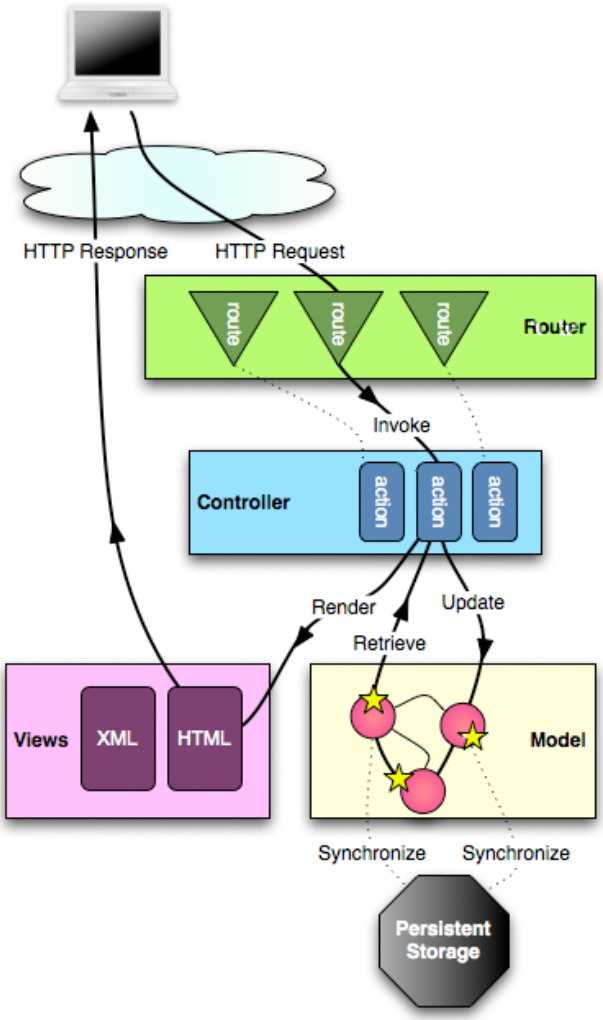

Figure 2: Play: Request Life Cycle [40]

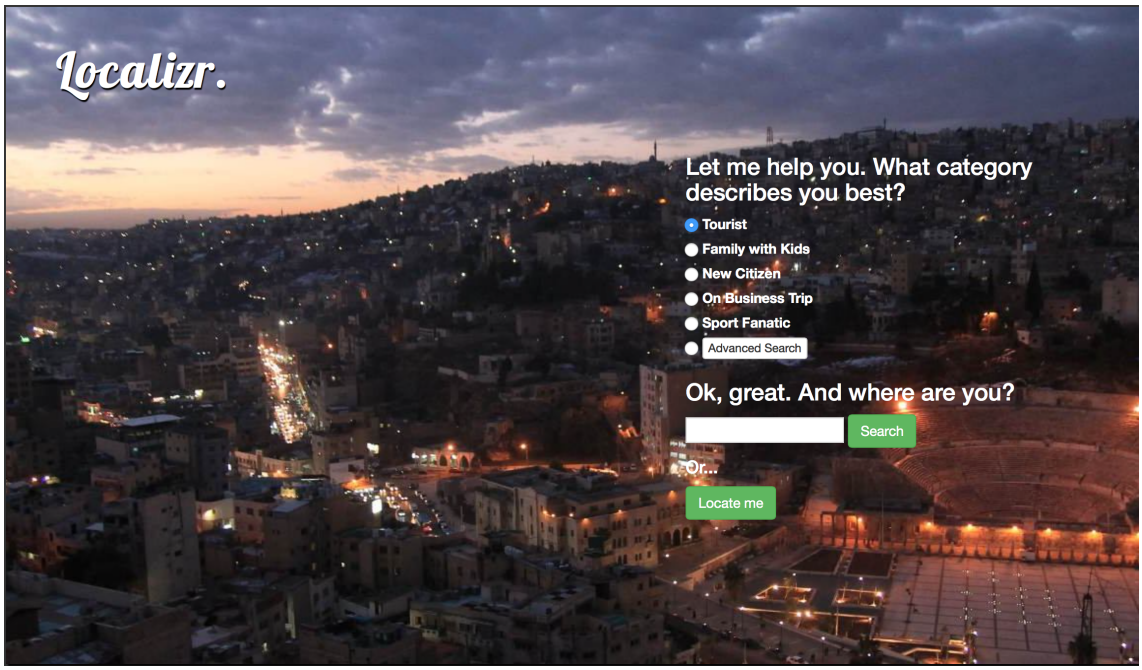

Figure 3: Screenshot: Startscreen

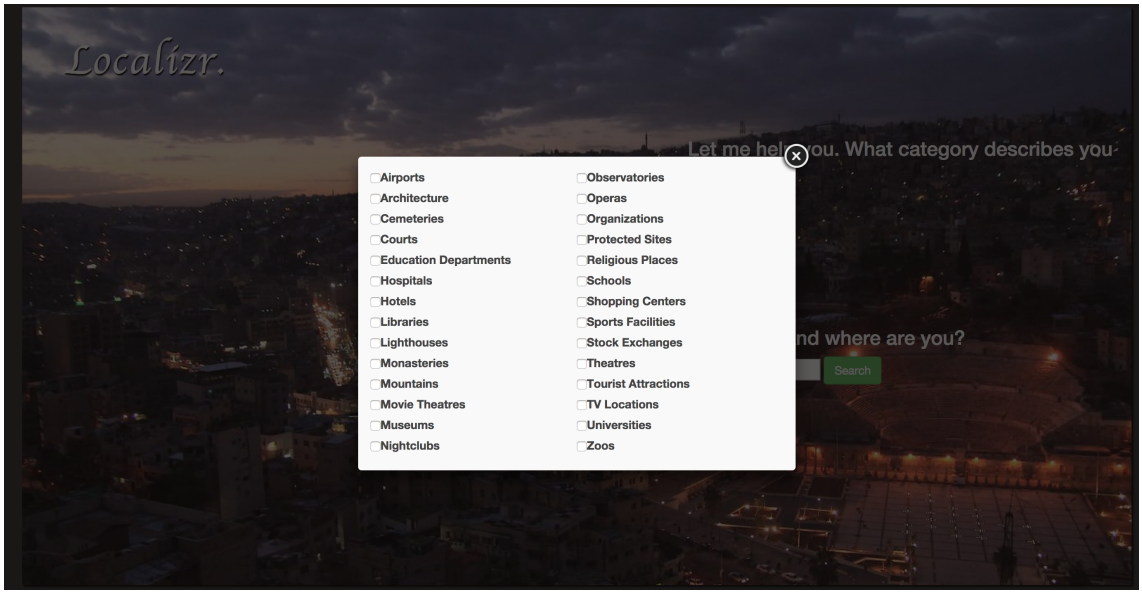

Figure 4: Screenshot: Advanced Search

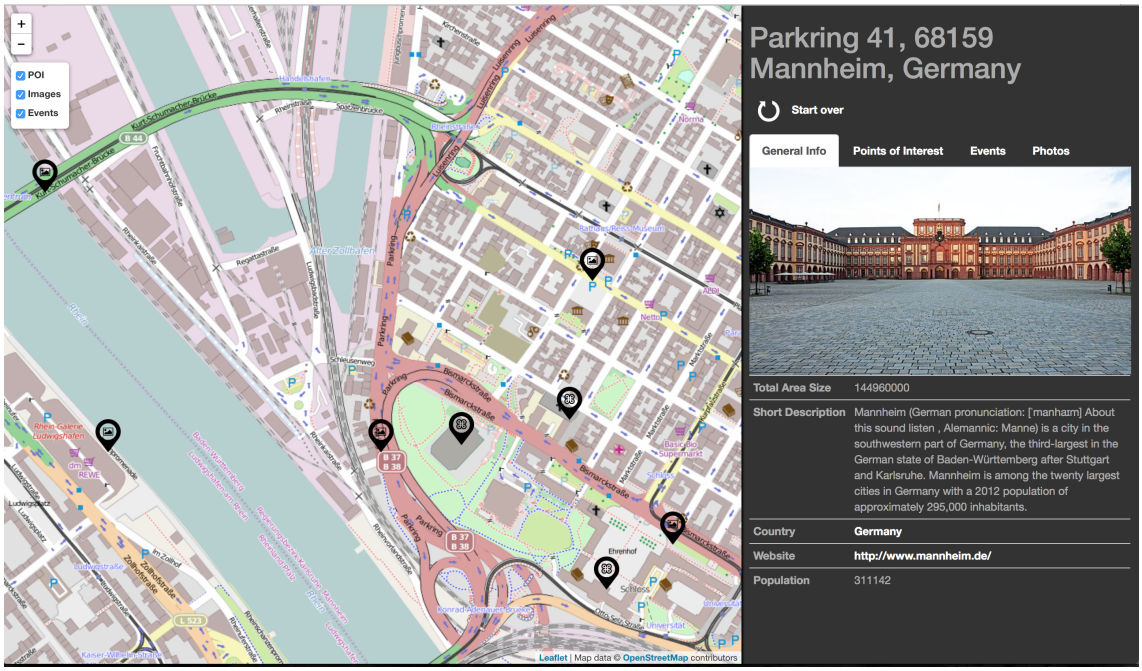

Figure 5: Screenshot: Results

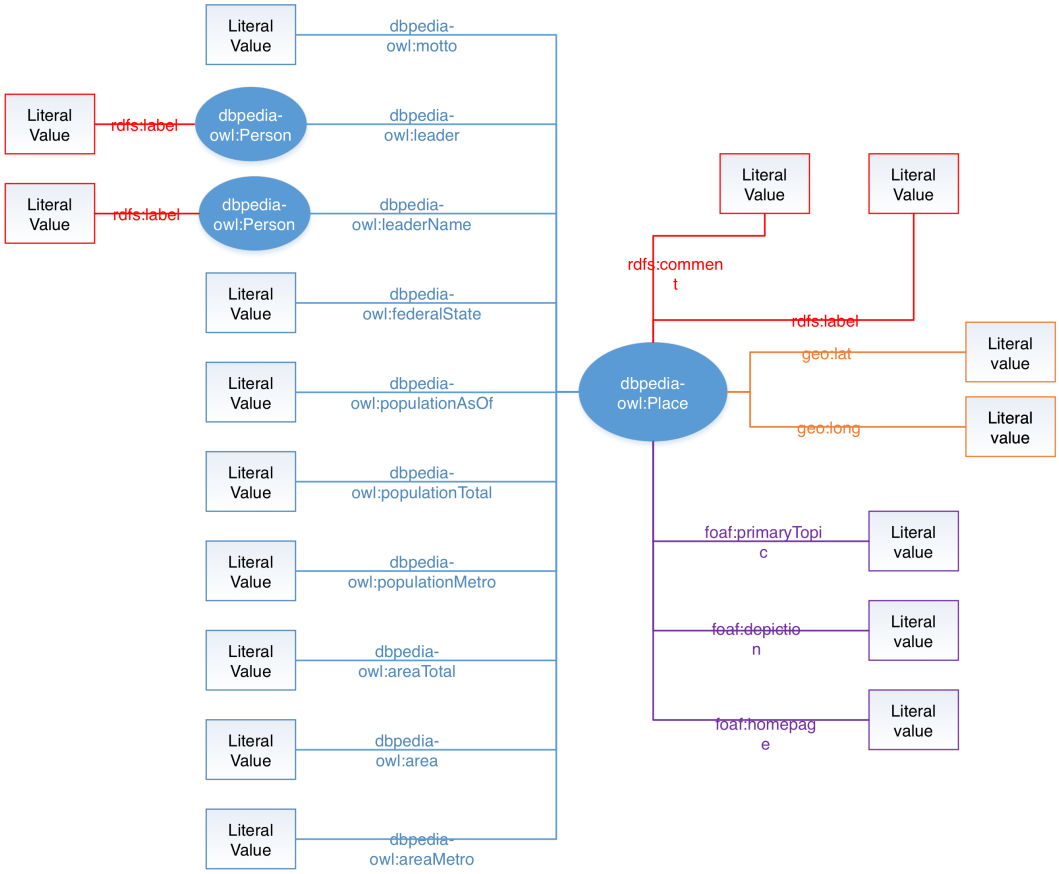

Figure 6: Semantic information extracted from the DBPedia dataset

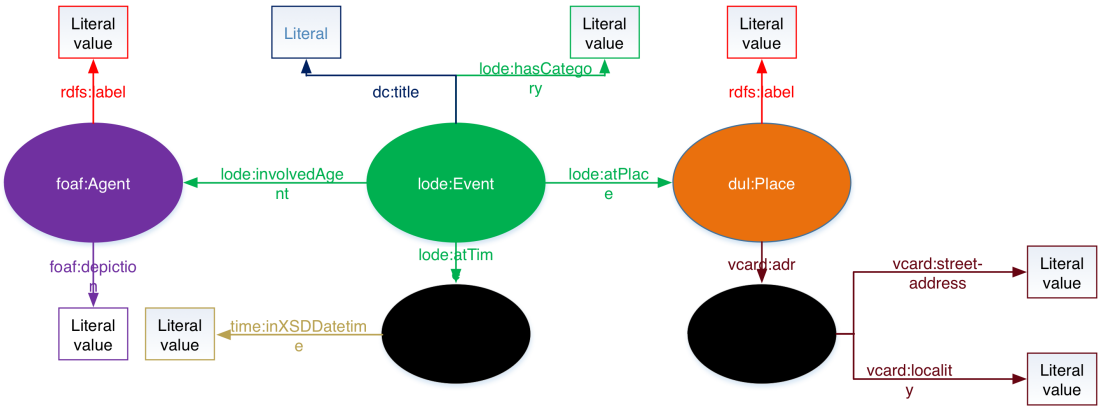

Figure 7: Semantic information extracted from the EventMedia dataset

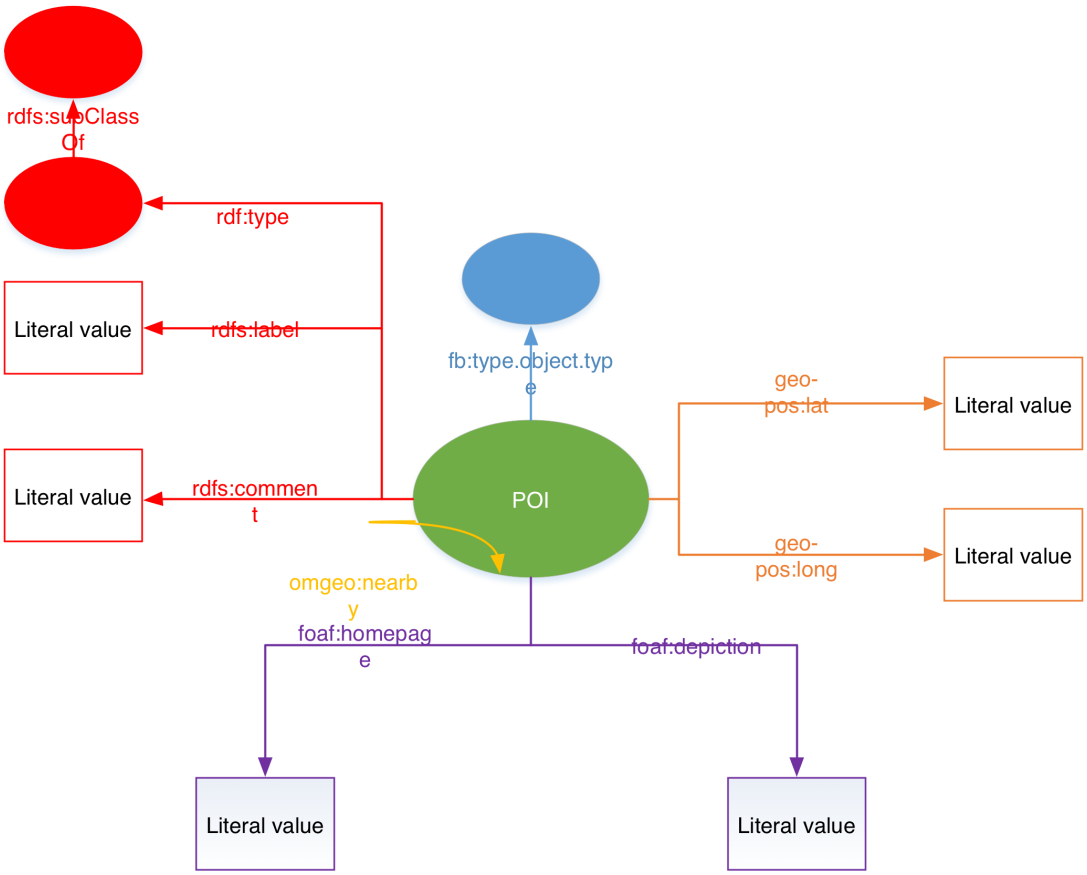

Figure 8: Semantic information extracted from the Freebase and GeoNames datasets

Figure 9: Custom POI Marker [24, 32]

Figure 10: Custom Image Marker [18, 32]

Figure 11: Custom Event Marker [15, 32]

Figure 12: Start Over Icon [16]

Figure 13: Wikipedia Icon [17]

Figure 14: Wikipedia Icon [17]

References

[1] Vladimir Agafonkin. *LefletJS*. URL: http://leafletjs.com/ (visited on 12/22/2014).

[2] Apache Software Foundation. *Apache Jena*. URL: https://jena.apache.org/ (visited on 12/22/2014).

[3] Jeremy Ashkenas. *UnderscoreJS*. URL: http://underscorejs.org/ (visited on 12/22/2014).

[4] Atlassian. *Bitbucket*. URL: https://bitbucket.org/ (visited on 12/22/2014).

[5] S Auer, Christian Bizer, Georgi Kobilarov, and Jens Lehmann. *Dbpedia: A nucleus for a web of open data*. 2007. URL: http://link.springer.com/chapter/10.1007/978-3-540-76298-0_52.

[6] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification 0.99*. URL: http://xmlns.com/foaf/spec/ (visited on 12/22/2014).

[7] Bundesministerium des Innern. *GOVDATA*. 2014. URL: https://www.govdata.de/ (visited on 12/22/2014).

[8] DBPedia. *DBPedia Ontology*. URL: http://wiki.dbpedia.org/Ontology (visited on 12/22/2014).

[9] *DBPedia SPARQL Endpoint*. URL: http://dbpedia.org/sparql (visited on 12/22/2014).

[10] Eclipse Foundation. *Eclipse*. URL: http://eclipse.org/home/index.php (visited on 12/22/2014).

[11] European Union. *European Union Open Data Portal*. 2014. URL: http://open-data.europa.eu/en/data/ (visited on 12/22/2014).

[12] FOSSGIS e.V. *Openstreetmap*. URL: http://www.openstreetmap.de/ (visited on 12/22/2014).

[13] FactForge. *FactForge SPARQL-Endpoint*. URL: http://factforge.net/sparql (visited on 12/22/2014).

[14] *Factforge SPARQL Endpoint*. URL: http://factforge.net/sparql (visited on 12/22/2014).

[15] Freepik. *Event Icon*. URL: http://www.flaticon.com/free-icon/election-event-on-a-calendar-with-star-symbol_48732 (visited on 12/22/2014).

[16] Freepik. *Start over Icon*. URL: http://www.flaticon.com/free-icon/restart_2046 (visited on 12/22/2014).

[17] Freepik. *Wikipedia Icons*. URL: http://www.flaticon.com/free-icon/wikipedia-logotype-of-earth-puzzle_48930,http://www.flaticon.com/free-icon/wikipedia-logo_49360 (visited on 12/22/2014).

[18] Dave Gandy. *Image Icon*. URL: http://fortawesome.github.io/Font-Awesome/ (visited on 12/22/2014).

[19] Aldo Gangemi. *DOLCE+DnS Ultralite Ontology*. URL: http://ontologydesignpatterns.org/wiki/Ontology:DOLCE+DnS_Ultralite (visited on 12/22/2014).

[20] GeoNames. *GeoNames Ontology*. URL: http://www.geonames.org/ontology/documentation.html (visited on 12/22/2014).

[21] Google Inc. *Google Geocoding API*. URL: https://developers.google.com/maps/documentation/geocoding/ (visited on 12/22/2014).

[22] Google Inc. *Gson*. URL: https://code.google.com/p/google-gson/ (visited on 12/22/2014).

[23] Nitin Hayaran. *Justified.js*. URL: https://github.com/nitinhayaran/Justified.js (visited on 12/22/2014).

[24] Stephen Hutchings. *POI Icon*. URL: http://www.flaticon.com/free-icon/point-of-interest-outline_2985 (visited on 12/22/2014).

[25] Jetbrains. *IntelliJ*. URL: https://www.jetbrains.com/idea/ (visited on 12/22/2014).

[26] JQuery Foundation. *JQuery*. URL: https://jquery.org/ (visited on 12/22/2014).

[27] Dave Leaver. *Leaflet.markercluster*. URL: https://github.com/Leaflet/Leaflet.markercluster (visited on 12/22/2014).

[28] Caolan McMahon. *AsyncJS*. URL: https://github.com/caolan/async (visited on 12/22/2014).

[29] Brandon Morrison. *GeocoderJS*. URL: https://github.com/geocoder-php/geocoder-js (visited on 12/22/2014).

[30] Open Knowledge Foundation. *Datahub*. 2014. URL: http://datahub.io/ (visited on 12/22/2014).

[31] Ryan Shaw. *LODE: An ontology for Linking Open Descriptions of Events*. URL: http://linkedevents.org/ontology/ (visited on 12/22/2014).

[32] Simpleicon. *Map Marker*. URL: http://www.flaticon.com/free-icon/map-marker_34468 (visited on 12/22/2014).

[33] Janis Skarnelis. *Fancybox*. URL: https://github.com/fancyapps/fancyBox (visited on 12/22/2014).

[34] Sindre Sorhus. *Bower*. URL: http://bower.io/ (visited on 12/22/2014).

[35] Typesafe Inc. *SBT*. URL: http://www.scala-sbt.org/ (visited on 12/22/2014).

[36] Unxos GmbH. *Geonames*. 2014. URL: http://www.geonames.org/ (visited on 12/22/2014).

[37] W3C. *RDF Schema 1.1*. URL: http://www.w3.org/TR/rdf-schema/ (visited on 12/22/2014).

[38] W3C. *Semantic Web Browser*. 2014. URL: http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/SemWebClients (visited on 12/22/2014).

[39] W3C Interest Group. *vCard Ontology*. 2014. URL: http://www.w3.org/TR/vcard-rdf/ (visited on 12/22/2014).

[40] Zengularity. *Documentation Play Framework*. 2014. URL: https://www.playframework.com/documentation (visited on 12/22/2014).

[41] Zengularity. *Play*. URL: https://playframework.com/ (visited on 12/22/2014).