MAPPING ASSISTED BY THE SEMANTIC WEB AND SOCIAL MEDIA INTEGRATION

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Jose Andres Bolaños Perera

SPRING
2015
MAPPING ASSISTED BY THE SEMANTIC WEB AND SOCIAL MEDIA INTEGRATION

A Project

by

Jose Andres Bolaños Perera

Approved by:

__________________________________, Committee Chair
Du Zhang, Ph.D.

__________________________________, Second Reader
Meiliu Lu, Ph.D.

____________________________
Date
Student: Jose Andres Bolaños Perera

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

____________________________, Graduate Coordinator
Jinsong Ouyang, Ph.D. ______________________

Date

Department of Computer Science
Abstract

of

MAPPING ASSISTED BY THE SEMANTIC WEB AND SOCIAL MEDIA INTEGRATION

by

Jose Andres Bolaños Perera

This project makes use of several knowledge base sources and the structured data advantages of the Semantic Web to assist users in the creation of rich and dynamic maps for business or leisure activities. Our smart search will be capable of recognizing entities that, combined with Text/Sentiment Analysis tools, will speed up and simplify how we populate our maps. Each entity type has its own action, and allows complex queries such as upcoming events with your favorite band or the best restaurants/hotels in your vicinity. For each entity I added social media integration, allowing users to define their own user experience by selecting specific properties they want to display on their maps (ratings, comments, pictures and other specific business information).

The project has built-in capabilities to share and embed user maps in any website. PHP, AngularJS and services such as AlchemyAPI, Factual and DBpedia are at the core of all functionalities. This project aims to change the way we map the world by providing a meaningful context, which is augmented with the semantic web and sentiment analysis.

_____________________, Committee Chair
Du Zhang, Ph.D.

_____________________
Date
ACKNOWLEDGEMENTS

I would like to express my gratitude to my family, girlfriend and co-workers who supported me throughout the development of this project. Their feedback and constructive criticism allowed me to steer the project in the right direction.

Special thanks to my brother Javier Bolaños, whose design guidance helped to create a unique, user-friendly interface.

I would also like to thank my project advisor Dr. Du Zhang for his advice, and all companies that made their data available to the public through APIs that made this project possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Project Overview</td>
<td>2</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Semantic Web</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 DBpedia</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Application Programming Interface (API)</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Sentiment Analysis</td>
<td>7</td>
</tr>
<tr>
<td>3. RELATED WORK</td>
<td>9</td>
</tr>
<tr>
<td>4. STRUCTURE</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Builder</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Viewer</td>
<td>12</td>
</tr>
<tr>
<td>5. TECHNOLOGIES</td>
<td>14</td>
</tr>
<tr>
<td>5.1 Languages</td>
<td>14</td>
</tr>
<tr>
<td>5.2 Frameworks &amp; Tools</td>
<td>16</td>
</tr>
<tr>
<td>6. BUILDING THE APPLICATION</td>
<td>17</td>
</tr>
<tr>
<td>6.1 Smart Search using the Semantic Web</td>
<td>17</td>
</tr>
<tr>
<td>6.1.1 DBpedia Resources</td>
<td>18</td>
</tr>
<tr>
<td>6.1.2 Entity Extraction Implementation</td>
<td>19</td>
</tr>
</tbody>
</table>
6.2 Sources of Information ................................................................. 22
  6.2.1 Factual ................................................................. 22
  6.2.2 Songkick ................................................................. 26
  6.2.3 Google Geocoding ....................................................... 29
6.3 Social Media Integration ................................................................. 31
  6.3.1 Factual Places Crosswalk .................................................. 32
  6.3.2 Social Networks .......................................................... 34
6.4 Sentiment Analysis ................................................................. 38
  6.4.1 Alchemy API ............................................................ 39
  6.4.2 Implementation ........................................................ 40
6.5 Caching ................................................................................. 41
7. MAPPING ........................................................................ 42
  7.1 Markers ........................................................................ 43
  7.2 Layers ........................................................................... 45
  7.3 Timeframes ................................................................... 47
8. STORAGE ........................................................................ 49
  8.1 JSON ............................................................................ 50
  8.2 SQL Server .................................................................... 51
9. PERFORMANCE & RESULTS ......................................................... 54
10. CONCLUSIONS AND FUTURE WORK ........................................ 56
Appendix A. APIs References ......................................................... 58
Bibliography ............................................................................. 60
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1. Components of the Builder</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2. Viewer interaction with users and devices</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3. Relationship among Builder, Viewer and Storage</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4. AngularJS code to call DBpedia’s API</td>
<td>15</td>
</tr>
<tr>
<td>Figure 5. AngularJS code to display suggestions</td>
<td>15</td>
</tr>
<tr>
<td>Figure 6. Accepted classes for each entity type</td>
<td>19</td>
</tr>
<tr>
<td>Figure 7. JQuery code to filter results by type</td>
<td>20</td>
</tr>
<tr>
<td>Figure 8. Example of a suggestion for the string “Costa Rica”</td>
<td>21</td>
</tr>
<tr>
<td>Figure 9. List of accepted entity types</td>
<td>22</td>
</tr>
<tr>
<td>Figure 10. List view overlay for restaurants</td>
<td>23</td>
</tr>
<tr>
<td>Figure 11. Factual API calls</td>
<td>24</td>
</tr>
<tr>
<td>Figure 12. HTTP request to the Factual API</td>
<td>24</td>
</tr>
<tr>
<td>Figure 13. List view overlay for hotels</td>
<td>25</td>
</tr>
<tr>
<td>Figure 14. AngularJS call to Songkick Artist Search API</td>
<td>27</td>
</tr>
<tr>
<td>Figure 15. Description of an artist on the list view overlay</td>
<td>27</td>
</tr>
<tr>
<td>Figure 16. AngularJS call to Songkick Artist Calendar API</td>
<td>28</td>
</tr>
<tr>
<td>Figure 17. Notification System</td>
<td>28</td>
</tr>
<tr>
<td>Figure 18. List view overlay for artists</td>
<td>29</td>
</tr>
<tr>
<td>Figure 19. Usage of Google’s geocoding functions</td>
<td>30</td>
</tr>
<tr>
<td>Figure 20. Presentation of a country in the map</td>
<td>30</td>
</tr>
<tr>
<td>Figure 21. Presentation of a landmark in the map</td>
<td>31</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

This project is based on the integration of efforts from many companies that made their information available through APIs. In a world where new knowledge bases become available to the public every day, the aim here is to use existing sources of information to guide users through the creation of rich and meaningful maps unique for them and their experiences.

This project was born from the idea of a suggestion system that requires minimal input from the user. By simply typing in the name of what the user is looking for, they would be able to find an abundance of information on that subject — no matter if the entity is a person, company or place.

Thanks to the Semantic Web, data is now widely interconnected and structured. These relationships between data become knowledge bases that machines read, and help them understand context. Through the collaboration of the community, these knowledge bases continuously grow, improve their accuracy and stay better updated.

If we combine these new technologies with social media, we have a powerful data source that can not only provide factual information about a person or place, but can also show recommendations, tips or photos of other human interactions with the entity. The combination of this data can provide users with real and current information about any entity they’re interested in.

This project applies this knowledge into the creation of maps, in which all entities have information gathered from several data sources. This allows for a richer and more dynamic experience for users.
1.1 Project Overview

Think of the amount of steps you would have to take and how many different data sources you would have to consult for information to be able to create a map of all city landmarks, or one with your favorite restaurants; now, how about a map with all of the places you would like to visit in a foreign city? Or, imagine that a concert or event is coming up and you would like to share a map of recommended hotels and restaurants around the venue to your friends. What if the event goes on for several days? This project integrates all of this information into a single map, and even includes special offers featured by different places, depending on the day.

The amount of search methods and data sources people need to consult every day keeps growing, and it can get overwhelming for people. We’re used to having to get our information from many different sources, even if we’re just trying to find a good place to eat. It is extremely common to look at Yelp reviews and ratings and/or Foursquare tips, and we have a certain dependency upon their accuracy. Similar situations occur when we’re looking for a hotel, store, landmark or artist information. The list is endless.

All of this data is already publicly available, but as of yet there is no way to integrate it into a single application – one that allows the creation of meaningful, personalized maps for users.

The idea of an application that integrates all of these information sources with centralized data sources and expansion of maps capabilities is what guided the project and made it a reality.
The rest of this report is organized as follows:

- Chapter 2 introduces all the concepts and terminology needed to understand the technologies used for the project implementation.
- Chapter 3 compares existing tools and applications with the concepts behind our application.
- Chapter 4 specifies the structure and features of the project.
- Chapter 5 describes the technologies used.
- Chapter 6 explains how the concepts of the Semantic Web, Social Media & Sentiment Analysis come together, digging into the code and sources of information used.
- Chapter 7 describes how all these concepts are presented to users on a map.
- Chapter 8 goes into the backbone of the project, specifying how data is stored and which structures are used to rebuild maps.
- Chapter 9 collects all results and measures the performance of the application.
- Chapter 10 presents our conclusions and future plans.
Chapter 2

BACKGROUND

This chapter focuses on providing the reader with all of the background knowledge needed to understand the processes and technologies that are used and implemented throughout the project.

2.1 Semantic Web

The Semantic Web [1] was created with the vision of a web of linked data, where each entity can have properties that represent relationships with other data. Ultimately, this would allow computers to have the knowledge required to perform more powerful and trusted tasks on the web. To reach this goal, the creation of several technologies was required:

- **Resource Description Framework (RDF)**
  RDF is a data model that can store data about data (metadata). RDF provides the specifications, data formats and protocols needed to represent web resources.

- **SPARQL**
  SPARQL [2] is a query language for RDF that defines the syntax to retrieve and manipulate data stored in RDF format.

- **Web Ontology Language (OWL)**
  A family of knowledge representation languages for the creation of vocabularies (ontologies), which define the structure of knowledge for various domains: the nouns represent classes of objects and the verbs represent relations between the objects. [3]
These technologies allow for the creation of data stores on the web, ways of organizing data (building vocabularies) and writing rules to handle data. Essentially, this allows the creation of knowledge bases that enhance the intelligence of web search and information integration.

This project uses a very powerful knowledge base (DBpedia) and SPARQL to perform entity recognition.

2.1.1 DBpedia
DBpedia [4] differs from other knowledge bases since it covers more than one domain and has a large community of contributors – they have chosen Wikipedia as their data source, which is one of the largest knowledge sources created by and available to mankind.

This crowd-sourced effort to extract structured information from Wikipedia and make it publicly available is the reason why DBpedia was chosen as the knowledge base for this project. DBpedia provides its own SPARQL endpoint and lookup API service, which allows the project to query for entity information. This interaction will be covered further in this document.

2.2 Application Programming Interface (API)
APIs are sets of standards, routines and protocols for accessing software applications. Companies release their own APIs to the public so that other software developers can create applications based on their service.

All of the APIs used in the project are REST APIs [5], which are based on the REST software architecture that provides the guidelines for scalable web services. Without them this project would not be possible, since they are the key feature that allows for the retrieval of information from artists, companies and places.
Since the REST API is based on open standards, it is possible to use any web development language to access it. For this project I am using AngularJS.

In order to use any REST API, the application needs to make an HTTP request and parse the response. All APIs used for this application support JSON, and this was chosen as the preferred response format.

**Authentication**

Most of the APIs used in this project require an authentication key, tokens and secret keys. Each API provider needs the developer to register their application on their website, requiring information such as application name, description and a brief paragraph explaining how the API is going to be used throughout the application.

Once the application is accepted, you’re provided with all necessary tokens to request information about their products. Some of these tokens are merely used for tracking purposes. Certain APIs allow only a fixed amount of free requests per day. The purpose of this can either be to protect their servers from receiving more requests than what they can handle, or to make a profit for sharing those services by charging extra for higher limits in usage.

Some requests require a higher amount of permissions by the user, hence a better authentication than just providing a key. This is where an open standard for authorization such as OAuth [6] comes in, specifying a process for resource owners to authorize third-party access to their server resources without sharing their credentials. It was designed to work with HTTP, and essentially allows access tokens to be issued to third-party clients by an authorization server with the approval of the resource owner (or end-user).

Each API used in the project will be explained in detail further along in the document, but to get a better understanding, here is an example of a Factual API call used to retrieve
locations of Chipotle restaurants located within a radius of 5,000 meters of a specified position in the map:

**Prefix:** http://api.v3.factual.com/t/places

**Suffix:** ?q=chipotle&geo={"$circle":{"$center":[34.0602,-118.4182],"$meters":5000}}

The prefix section provides the base URL of the source and the suffix provides the parameters necessary to interact with it. In this case only two are used: 1) the query text identified by the statement “q=chipotle”, which according to its Factual documentation specifies that the text to search is “chipotle,” and 2) the second parameter is “geo”, which provides geographic information about the user.

This is then wrapped into an HTTP Request and sent to the API, which will check whether the request is valid and build a response according to the sent specifications.

### 2.3 Sentiment Analysis

Making a machine understand that a sentiment is the attitude, opinion or feeling toward an entity is a very complex task, usually involving the use of natural language processing, text analysis and computational linguistics to identify and extract subjective information.

In this project, sentiment analysis will be used to detect the contextual polarity of a given text at document, sentence and feature level. We don’t use our own algorithms – instead we rely on the Alchemy API [7] to identify positive or negative sentiment within reviews and tweets. These algorithms classify sentences into two principal classes with regard to subjectivity:

- **Objective sentences:** Contain factual information.
- **Subjective sentences:** Contain explicit opinions, beliefs and views about specific entities.

AlchemyAPI uses unsupervised approaches to document-level sentiment analysis, which are based on determining the semantic orientation (SO) of specific phrases within the document. If
the average SO of these phrases is above some predefined threshold, the document is classified as positive. Otherwise, it is labeled negative.

There are two main approaches to the selection of the phrases: either 1) a set of predefined POS patterns can be used to select these phrases, or 2) a lexicon of sentiment words and phrases can be used.

Now that we have a better understanding about the core technologies used in the project, we’ll move into details about how the different technologies interact with each other to perform tasks such as entity recognition, social media extraction, sentiment analysis and their integration into a mapping application.
Chapter 3

RELATED WORK

Designing maps and adding locations to them is not a new science. We continuously find new ways to map and discover venues around us in any device – we see it every day in most mobile applications, with built-in sections where we can see venue and landmark information placed on a map. Let’s start by looking at several tools that allow for the creation of maps, such as:

- SmartMap (http://smartmap.com/)
- Google Maps Engine (https://mapsengine.google.com/map/)
- MapBox (https://www.mapbox.com/)

These tools allow for the creation and design of maps based on datafiles; but they lack the social integration and simplicity provided by a semantic search. These tools allow for a very specific design of map features, with controls to change road colors and labels. For example, Google can receive a datafile with several coordinates and will add a marker for each of them.

Each tool covers a specific area – either map design or addition of locations to the map – but none of the existing tools use the advantages of the Semantic Web. They lack the ability to conduct entity extraction of any text that isn’t in their database, e.g., Google uses Google Places to display results in their search, but if what we’re looking for is an artist instead of a venue we won’t get any results.

Every crowd-sourced application, such as Yelp and Foursquare, provides its own maps – but they are limited to their own information. This gives my application an advantage, since I am integrating information from various sources, giving the user different options to choose from.

Many applications apply Sentiment Analysis to Amazon products reviews [8], and also use tweets to retrieve the overall sentiment about a company or political figure [9]. But no
existing platforms or tools use it for mapping purposes, e.g., retrieving tweets regarding venues around the user and sorting them by sentiment. Yelp reviews have also gone through many text analysis tools to remove advertising and other undesired content, but the reviews are not usually taken into consideration in order to decide which venue has a better perception by users than others. I have not yet found any application that sorts and filters results on a map depending upon the final sentiment retrieved from a combination of sources.

A different approach to mapping has been taken by the University of Maryland in a tool named NewsStand [10], which retrieves news articles from hundreds of RSS news sources and displays them on a map. This map-query interface allows users to navigate to the specific areas of the map where they want to get news from. It has some similarities with my application on the sense that it retrieves information from different sources, places them on a map and allows users to interact with it. No sharing or customization capabilities are included but it’s based on similar concepts.

There are very useful tools available to the public, but each has a very specific purpose. What we wanted to do here was to allow users to get all of the information they need from a single source, and provide them with detailed information to help them decide which places should or should not be included on their maps – allowing informed decisions to be made upon a set of complete information from one source.
This project is built with the idea of users sharing and embedding their maps all over the web, built through the collaboration of many social media APIs. We’d also like to be able to contribute back to this community by providing users with all the necessary means to make this social propagation possible.

For this reason the project was separated into two sections: a builder and a viewer.

4.1 Builder

As illustrated in Figure 1, the builder is the core of all functionalities, the project’s custom design and user-friendly interface allows users to authenticate and add venues, events and businesses to the map – which can be enriched by adding information gathered from social media. Users can personalize what data is displayed or not in their location by selecting which social media sources will be used, e.g., for each restaurant that is displayed, the user can choose to display tips from Foursquare, reviews from Yelp and pictures from Instagram – all based on personal preference.

The builder takes care of all of the construction stages of the map, from adding and grouping locations in the map to selecting their presentation. It also collects and converts all data from the new map into JSON format. Which is then sent to the application database in an external
server, where it is stored and becomes instantly accessible for other users to see (depending on the privacy settings on the map, which are covered in Chapter 8).

By itself the builder has no storage unit – it depends on the temporary JavaScript objects created at the clients browser level, and this lightweight feature is what makes it so fast.

Every time users want to access their maps, no matter from which location, the builder will retrieve its data from the application database stored in a remote server. All the user needs to provide is their credentials (username and password), and a request will be created and sent to the server. The server will then validate the authentication, and the map data stored in JSON format will be sent back on the HTTP response to the builder (browser) on the user’s machine. This data allows the builder to reconstruct the map locally and grants the user the ability to continue updating it.

4.2 Viewer

Once a map is stored in the database, all we need is a platform for external users to be able to interact with it. This is where the Viewer comes in; its interactions are illustrated in Figure 2.

As the name implies, this is specifically for viewing – no map editing or construction is done in this section. When a user allows sharing on their map, a link and embedding code will automatically be created. The user can then share this through social media or simply by sending the link to any interested source.
Any external user can open the link and gain access to the viewer, which contains all grouping and presentation options selected by the map creator. This includes all venues and locations added to the map.

It is dynamic and shows all specifications the map designer wants to display. The only difference is that it’s not editable and users don’t need to authenticate to be able to see it.

Figure 3 provides an overall view of the structure of the project. The builder has a two-way communication with the back-end storage, allowing maps to be saved and retrieved at any time. The viewer can only retrieve maps to allow interaction with external users.

![Figure 3 - Relationship among Builder, Viewer and Storage](image-url)
This project is based on the retrieval of external information and its customized presentation. That is why AngularJS, jQuery, PHP and CSS3 were the languages selected for its implementation.

5.1 Languages

AngularJS [11] carries all the main functionalities of the application by making all external calls to the APIs. It provides a client side model-view-controller (MVC) architecture that allows for a seamless integration of data from external sources and its logical and continuous presentation to the users.

Since it is embedded in the HTML by the use of custom tags attributes, we are able to bind the response of the API calls into the presentation layer through JavaScript variables. This is possible thanks to Angular’s capability to interpret those HTML custom attributes as directives, which bind those parts of the code to a model that is represented by standard JavaScript variables. These variables are manually set within the code to store JSON formatted responses received from external APIs.

For a better understanding I will present sections of the code that highlight how AngularJS helps us integrate DBpedia’s API responses into our HTML. First we need to create an HTTP request to ask DBpedia’s API for information. In Figure 4 we are asking for resources that match a given search text.
You will notice the use of a variable named scope. In Angular this is an object that refers to the application model, where the variable searchText had already been defined. For the purpose of the example, let’s focus on what happens if the request is successful. As illustrated in Figure 5, a variable named “suggestions” receives the response of the API in JSON format, and this variable is used in the HTML custom tag attributes to bind its values within the HTML.

Angular’s directive ng-repeat acts as a loop to traverse the suggestions of JSON. For each object on the JSON, it will create an HTML table using each object’s attributes, such as label, description and type.

This capability to add more logic into the HTML code is a cleaner approach to display client-side variables in the presentation layer, and is the logic followed by each communication with external APIs throughout the project.
**PHP** has a much smaller role in the project than Angular. It is used to be able to reuse code (import files) and to communicate with Twitter’s API through a PHP library.

**Javascript** and libraries such as **JQuery** and **Google Maps** are mainly used to take care of animations of UI elements and map functionalities, such as adding points, layers and displaying InfoWindows. Most of the mapping structure is created through JavaScript objects. This will be expanded upon in Chapter 6.

### 5.2 Frameworks & Tools

The front-end framework **Bootstrap**, which is based on CSS, HTML and JavaScript, is used for the responsive design and customized UI elements of the project.

This presentation framework is used in SASS format. This is a CSS extension language, with capabilities to create variables, use operators and inheritance within CSS classes.

This helped organize all the project’s presentation layer code, which was designed to be able to make UI changes faster, e.g., instead of changing a color or width for all div elements within a part of the HTML, we can just change one variable that affects all of those elements. Instead of assigning fixed values to CSS elements, we use variables.

A very helpful tool used for the development of the project is Prepros [12], which allows the compilation of SASS code into CSS, provides auto-compilation, live browser refresh and minification of JavaScript files (removing all unnecessary characters from the source code without changing its functionality). SASS is only used during development stages as a way to simplify the styling of the project. Once deployed, its code is compiled into CSS and those files become the presentation styles used by the project.

**SQL** Server was used for the creation and management of the database. A detailed description of the storage of the project is covered in Chapter 8.
Chapter 6
BUILDING THE APPLICATION

This project consists of the integration of several knowledge bases to assist users through the creation of interactive maps. Its challenge is to provide a seamless experience through the combination of all used data sources.

The project reaches its goals by the use of these main components:

1. Smart Search using the Semantic Web.
2. Social Media Integration.
3. Sentiment Analysis.
4. Creation of Interactive Maps.

6.1 Smart Search using the Semantic Web

The first step to start creating maps is to select an entity you would like to get information about or select a place in the map. With the suggestion system, we’re able to assist users in finding what they are looking for.

This system is based on the use of the DBpedia Lookup service to perform entity extraction tasks.

What is the DBpedia Lookup Service?

DBpedia Lookup Service is a web service that allows the retrieval of DBpedia URIs (resources) by comparing a given text with related keywords. This means that either the label of the resource matches, or an anchor text that was frequently used in Wikipedia to refer to a specific resource matches.
I used the PrefixSearch API to create an autocomplete search box. For any given partial keyword such as “Sacr,” the API will return URIs of related DBpedia resources, such as http://dbpedia.org/resource/Sacramento,_California.

How does this integrate into our suggestion system?

With this API we are able to obtain related resources that we can organize into categories, allowing us to display to the user different suggestions depending on the text they are entering.

This gives the application an entity extraction capability; if a user is entering Sacramento, thanks to the DBpedia resource suggested, we will know that it refers to a Populated Place – in this case a city, which will allow us to perform a different action for each entity type.

### 6.1.1 DBpedia Resources

A DBpedia resource consists of a number of properties that contain factual information and relationships with other data. The following are 3 properties that make the suggestion system possible:

- **Classes**: Contains the type of the resource, e.g., populated place, company, music group or architectural structure. All of these examples are part of an extensive class vocabulary created by DBpedia.

- **Categories**: Holds a number of lists that the resource is part of, e.g., Mexican restaurants, companies based in Sacramento, American female guitarists and more. Each of these categories helps to identify and filter the resource.

- **Abstract**: Provides a brief description of the resource.
I have separated the different classes and categories returned by a DBpedia resource into five types of objects, which will have their own actions when selected:

1. Restaurants.
2. Hotels.
3. Stores.
4. Artists.
5. Cities & Landmarks.

Now let’s connect all the puzzle pieces.

When you type a keyword into the DBpedia PrefixSearch, the API returns a number of matching resources. These are filtered by my “Entity Filtering” function, which traverses all of the results in search of accepted entities (restaurants, hotels, stores, artists and landmarks).

### 6.1.2 Entity Extraction Implementation

Figure 6 displays a portion of the JQuery code that looks for accepted sources. For each type I created a list of strings that contains classes that would match a specific type:

```javascript
var artistLabels = ["band", "music group", "musical artist", "artist"];
var companyLabels = ["company", "organisation"];
var placeLabels = ["place", "populated place", "country", "lake"];
```

Figure 6 - Accepted classes for each entity type

As we can see, the strings “band” & “music group” are part of the type “Artist”. Now, in Figure 7 let’s apply this logic to all of the results retrieved from the API.
This code loops through each entity class, retrieves the label (name) of the class and compares it our specific list of classes for each type.

But sometimes, this logic will not be enough to filter accepted sources, as is the case of the types Restaurants, Hotels and Companies; hence, the “searchInCategories = true” statement. These types share similar classes, such as “company” and “organization”, which makes it very complicated to differentiate only using an entity’s class.

This is where the categories are used. They provide specific information about the company, which allows the program to differentiate types, e.g., hotels are part of categories such as hotels in the United States, and restaurants belong to categories such as restaurant chains in California.

Now that we know the type of each suggestion, we can add specific styling and actions associated to each entity’s type. To enrich the suggestion we make use of the abstract property to display a brief description of the resource, as shown in Figure 8.
This is how the project’s entity extraction module and suggestion system is implemented.

So far we have created a smart search that assists the user with suggestions as they type, using data coming from a semantic database – building in essence an entity extraction system.

A user does not need to specify that what they are looking for is a restaurant or an artist. They can simply start typing what they are looking for, and the application will recognize the type of the entity.

What if no suggestions can be found?

Since DBpedia is based on data from Wikipedia, there is a chance that the searched resource does not have an entry in the semantic database. In this case, we would have no way of identifying the type of the resource.

This is why we implemented a backup system that allows the user to specify what type of result they expect to receive. Figure 9 illustrates this functionality.
The user simply selects the type of resource they are looking for from the list, then adds the text of the resource and performs the search. This will have the same functionality as if the user had selected a suggestion for that resource.

Now that the user has successfully found the type of data they are looking for, it is time to apply the appropriate actions and use new sources of information to retrieve data.

6.2 Sources of Information

Each accepted entity type has a specific action (described in next section) that is performed right after the user selects a resource. Each of them requires the use of an external API to reach its goal.

This section contains a detailed description of each action and its respective source of information.

6.2.1 Factual

Factual [13] is an API that contains data on over 65 million places and points of interests in 50 countries, all of which are constantly updated and improved as a crowd-sourcing effort. It caught
my attention since it integrates inputs from many social media sources, collecting and making publicly available most of the important attributes of each business.

Its **Global Places API** makes it possible to access detailed information about local businesses (restaurants, hotels, stores etc.) and points of interest (museums, airports, landmarks etc.). We are able to query to specific Factual data extensions for restaurants and hotels, which provide detailed attributes for each type. Thanks to this source of information, we were able to define specific actions for our entity types.

**Restaurants**

When the user selects a resource identified as a restaurant, the application will use the Restaurants Data extension of the Factual API to retrieve the business locations around the current map center.

As evidenced in Figure 10, besides nearby locations, the application provides a brief description of the restaurant along with detailed attributes such as restaurant rating, address, phone number, hours of operation, cuisine, price range, website and more.

![Figure 10 - List view overlay for restaurants](image-url)
But how is this information being requested internally?

Based on the Factual API specifications, an HTTP Request is built and sent to the API. Figure 11 shows the AngularJS code that creates this communication.

```javascript
$scope.getPlaces = function (searchText) {
}

$scope.getRestaurants = function (searchText) {
}

$scope.getHotels = function (searchText) {
}
```

Figure 11 - Factual API calls

In this case, we can see that each entity builds a different call to access specific extensions of the Factual API, e.g., restaurants are retrieving information from the extension “restaurants-us”, which translates into restaurants in the United States.

Once the API base call is created, it is time to add authentication and location parameters to the request, as illustrated in Figure 12.

```javascript
var searchQuery = (searchText != "") ? '&q=' + searchText : "");
$http.get(apiUrl + '&KEY=' + factualKey + searchQuery
  + '&geo="Scircle:"{Center:"' + mapCenter.lat() + ",", Lon:"' + mapCenter.lng() + ",",Radius:"' + radius + "")
```

Figure 12 - HTTP request to the Factual API

The factual key is added for authentication purposes, and the geo parameter holds all of the information about where in the world the user would like to get information from (latitude, longitude and radius around this location).

A response is sent containing all information requested in a JSON format. Later in this document, I will explain how this data is used to add these locations to the map.
Built in with Factual’s restaurants API is the ability to send types of cuisines as the search text, e.g., if we send “Indian” on the query parameter, a list of nearby restaurants serving Indian food will be returned. This simplifies the search of restaurants on any location for all users.

Hotels

All entity types have a similar methodology regarding the action implemented after selecting a resource, which can be resumed with the retrieval of the next information:

1. Nearby locations of the resource based on current settings (e.g., radius).
2. Specific attributes about the entity.
3. Social Media integration (Covered in section 3 of this chapter).

The only difference between hotels, companies and restaurants is the set of attributes displayed, as shown in Figure 13.

Figure 13 - List view overlay for hotels

We can now see the similarities about the structure in which data is displayed for each entity. Hotels vary, in the sense that under the rating it has data only applicable to hotels. From left to right, each of those icons represents internet access, restaurant inside the hotel, fitness facilities,
pool and spa services. Other categories, such as pet friendly, will also be displayed when available.

**Companies (Stores)**

This is the third and last entity type that is retrieved from Factual. It is also set as the default API in case no suggestions or categories are selected. It uses the sent query parameter text to match any nearby companies, returning results from any type of company. For this project it is used specifically to look for stores, e.g., Best Buy, Target, Home Depot or any other company that provides a service to customers.

The action implemented for this entity type is the same as for restaurants and hotels – once results are found, they are displayed on a list view overlay on the map. The only differences are the attributes displayed.

### 6.2.2 Songkick

Songkick [14] has one of the largest live music databases in the world, with over 3 million upcoming and past concerts and over 100,000 setlists. This makes it an ideal API to allow users to find their favorite artists’ upcoming events.

Our project will access two of their API sections: Artist search and Artist calendar.

There are two ways to access these APIs within the project:

- Selection of an artist suggestion from the smart search.
- Performing any search when the “artists” category is selected.

The project will first look for the artist and then retrieve their upcoming concerts. Figure 14 shows the AngularJS code section that looks for an artist by name.
We are using the API base URL “api.songkick.com/api/3.0/search/artists.json”. This takes the sent query parameter value and looks for matches with any artist names in their database. If the request is successful and a match was found, we automatically make another API call to get the artist’s calendar.

While this second API call is sent, the project displays information about the artist to the user. Figure 15 is an example of how this information is displayed if the searched artist is U2.
For example, as illustrated in Figure 16, to retrieve a band’s calendar, we create another HTTP request to the respective artist calendar API section.

```javascript
$scope.getSongkickArtistCalendar = function (artistId) {
    var endPoint = 'http://api.songkick.com/api/3.0/artists/' + artistId
                   + '/calendar.json?apikey=' + songkickKey + '&callback=?';

    $http.get(endPoint)
        .success(function (res) {
            mapConcerts(res.resultsPage.results.event);
        });
```

Figure 16 - AngularJS call to Songkick Artist Calendar API

If the request is successful, we use the returned JSON results to display a detailed list of the artist’s concerts and festivals.

An artist entity type action is described by the following steps:

- Look for artist on Songkick’s database.
- On success, retrieve artist’s calendar using Songkick’s calendar API.
- Display events for the user to navigate through, and add to map if wanted.

In case an artist cannot be found or has no upcoming events, we will use our notification system to alert the user of this fact. This system displays a modal window, shown in Figure 17, on the top right corner that will disappear in five seconds if not discarded by the user.

Figure 17 - Notification system

Some of the information displayed for concerts involves attributes such as type, lineup, venue, city and date, as illustrated in Figure 18.
The mapping of these events and how they are saved and grouped will be covered in the next chapter.

6.2.3 Google Geocoding

Using Google Maps as the map provider has many advantages thanks to the wide number of built-in features in their JavaScript library. The Geocoder class contains a geocode function that receives an address as a parameter and returns the position in the map where this address is located, as evidenced in Figure 19.
When a user starts typing the name of any landmark or city, the project’s suggestion system will recognize its type and relocate the map position to fit the city or landmark location.

As shown in Figures 20 and 21, JavaScript functions are able to determine the required zoom levels on the map to fit the new location on the map correctly.
Landmarks can be saved on the map, along with all grouping functions that are also applied to them.

6.3 Social Media Integration

Social media integration is what gives the application the ability to display real time and dynamic information. To make this possible, we need to communicate directly with each of the social media sources used (Twitter, Foursquare, Instagram, Flickr and Yelp).

Finding each location in all databases seems to be a very time consuming process with large overhead. Thankfully, Factual provides a service that makes this social media integration feasible.

Let’s keep in mind that our goal is to be able to display meaningful information to the user about each place or event they are adding to their maps. This will include tips, reviews, photos and detailed attributes of each place.
6.3.1 Factual Places Crosswalk

The factual places crosswalk service centralizes all resources retrieved from over 50 different sources, and becomes the easiest way to bridge multiple location APIs [15].

One of our biggest issues with integrating so many social media sources was to identify the same place entity in each of the different services. Factual makes it possible to access all of those identifiers by only providing one out of the fifty services it contains. Figure 22 is a representation of this functionality.

![Figure 22 - Factual Places Crosswalk overview](image)

The project already uses Factual to retrieve companies, hotels and restaurants. Each of those entities retrieved contain their own Factual ID, which can then be used with the Places Crosswalk to retrieve all of the other social media identifiers.

We only used the identifiers from the sources we need. But, since we already have access to so many different sources, the project gains a large range of growth. We could start displaying
menus and allowing users to make dinner reservations, for example, but for now we will focus on a limited number of sources.

The communication with the Crosswalk API is no different from the previously covered APIs: as shown in Figure 23, an HTTP request is sent to the respective API address containing a Factual place ID, and in return the user receives a result in JSON format containing all third party IDs found for the respective entity.

```javascript
$scope.getCrosswalkEntries = function (factualId) {
  $http.get('http://api.v3.factual.com/t/crosswalk?KEY=' + factualKey
  + '&filters={"factual_id":"' + factualId + '"}')
  .success(function (res) {
    if (res.response.data.length > 0) {
      linkFactualSocialMediaData(factualId, res.response.data);
    }
  })
}
```

Figure 23 - AngularJS call to the Factual Places Crosswalk API

It is important to notice that requests to this API are not made unless the user wants to see more information about a place. Initially, results will only be posted on the list view overlay – each location will already have detailed factual information, but no dynamic or crowd-sourced data will be on it.

Once a user clicks on any location from the list view, the Crosswalk API will be used, and all the results will go through a filtering process. This filtering process checks whether the namespace corresponds to one of the social media sources used in the project. If a match is found, a direct communication with this source will start. These extra communications retrieve the dynamic data the user is looking for.

The Crosswalk API saves the project at least one API call each time we retrieve data from any social media source. Instead of sending a request to find the entity, we directly ask for its details since Factual already provided us with all of the identifiers we need.
Figure 24 contains one of the code sections that filters through the Crosswalk API results. It also illustrates how each matched namespace has its own action, e.g., if the Foursquare identifier is found, we will communicate directly with Foursquare to retrieve tips.

![jQuery function that filters the Crosswalk API results](image)

We now move to the specific actions that are performed after an accepted social media resource is found.

### 6.3.2 Social Networks

**Foursquare**

I decided to retrieve tips from Foursquare, since this crowd-sourcing service is used worldwide, constantly updated and contains personalized and real-time recommendations from users that already experienced that place. Figure 25 shows an example of how tips from Foursquare are displayed to the user.
Each of the selected social media sources was chosen with the goal of providing meaningful information to the user. Another example of this is the reviews featured on Yelp.

For these communications we use the same format as previously described to make an API call. A list of references to the specifications of each API service will be provided at the end of the document.

**Twitter**

We all know the amount of useful comments, complaints and recommendations coming from this giant service. In the project, an area right next to the Foursquare tips was reserved as the location for a live Twitter feed from the current place the user is visiting on the application.

Authentication with this API is different, which led us to the use of a PHP library that assists with the interaction of the Twitter OAuth REST API.

The application-only authentication flow follows these steps:
- Encodes our project’s consumer key and secret into a specially encoded set of credentials.

- Makes a request to the POST oauth2 / token endpoint to exchange these credentials for a bearer token.

- When accessing the REST API, we use the bearer token to authenticate.

Once authenticated, we use the Search API – specifically the GET method search/tweets, identified by the resource URL https://api.twitter.com/1.1/search/tweets.json.

This returns a collection of relevant Tweets matching the specified query parameter. This information is not only important to display more useful information about the place, but it is the text source used for the sentiment analysis methods, which retrieve the overall sentiment that crowd-sourced platforms such as Twitter have about that specific location.

**Instagram**

Once all reviews, tweets and tips are selected, it is time to show some visual feedback to the user. Factual Crosswalk returns the “instagram_place” identifier of any place selected, which can be used directly with the Instagram location endpoint to retrieve recent media objects (images and videos).

The resource URL used is:


The results include several sizes for each image and are sent to a jQuery function that retrieves them and creates a UI carousel element for the user to be able to navigate through.

This function is reused to display Flickr images as well. Most of the functions throughout the project are designed to be reused by similar objects.

**Flickr**
Flickr is the only social media source we use that is not part of the Factual Crosswalk third-party services. The method accessed does not require authentication, which makes it a great candidate for displaying photos of each entity – specifically in the list view description section.

The API method used is flickr.photos.search, which returns a list of photos matching the criteria sent as a parameter. Since we are not authenticated we can only access public photos, which is exactly what we need – a way to access public photos of an artist, venue or landmark. Most of the figures presented earlier in this document use Flickr as their photo provider.

Instagram “places” images are location-oriented but Flickr images are not, hence the reason why Instagram images are displayed for specific locations and Flickr images are displayed for general public photos of artists or landmarks. An example of Instagram images is displayed in Figure 26.

Figure 26 - Presentation of Instagram images for a restaurant
Settings

We want users to be able to customize the information displayed on each of their places as much as possible, which is why we created a settings section that allows users to select which social media sources they would like to display or hide.

This section simply contains checkboxes for each source, and all configurations will be saved directly into the map settings. Each source is displayed only if it is checked. This allows users to enforce their social media services preferences.

6.4 Sentiment Analysis

Teaching a machine to retrieve the sentiment of a text is one of the most complicated tasks of Machine Learning [16]. We could create a whole Masters project on the algorithms involved in this process, but this project is about collaboration which is able to display what can be done when a number of web services are integrated into a new service. And, sentiment analysis is not the exception – we aim to provide the user with more options in how they can find the best places, which is a goal that involves a crowd-sourcing effort.

All of the information needed is already publicly available. It is just a matter of using the information with the right tools to provide new search engines.

How will this process work?

Users are restricted to a stars system from Yelp to know if a place is good or not. But what if we could bring tweets and reviews into consideration? This project already has access to all of these resources, tweets and reviews. We will add a new external communication that will allow us to perform an overall sentiment analysis of the social networks, targeted to specific locations such as restaurants, hotels and nearby companies.
6.4.1 AlchemyAPI

AlchemyAPI is a pioneer in assisting computers in the understanding of human language. It uses machine-learning algorithms to perform text and sentiment analysis. These methods look for words that carry a positive or negative connotation, then figure out which person, place or thing the comments are referring to.

The sentiment analysis API works on documents, articles, blog posts, product reviews, comments and tweets. Basically, it provides all of the tools that we need for our project.

In our case we will be using the API to extract document-level sentiment from publicly-accessible webpages and given texts, which each require their own method:

- API Call: TextGetTextSentiment
  
  Used to extract positive/negative sentiment from a given text; in our case a review, comment or tweet.
  
  Endpoint: http://access.alchemyapi.com/calls/text/TextGetTextSentiment

- API Call: URLGetTextSentiment
  
  This powerful method extracts positive/negative sentiment from a given web page. AlchemyAPI will download the requested URL, extracting text from the HTML document structure (ignoring navigation links and advertisements) and performing sentiment extraction operations.
  
  Endpoint: http://access.alchemyapi.com/calls/url/URLGetTextSentiment

Extracting sentiment from a given URL allows the application to perform sentiment analysis on twitter pages, saving the time that it would take to retrieve the latest tweets of a given venue.

For example, if we want to classify the overall sentiment from Chipotle, we give the API the URL of a Twitter search page for Chipotle mentions: twitter.com/search?q=Chipotle. This will
only extract the sentiment of tweets. The API call that retrieves this sentiment is illustrated in Figure 27.

```javascript
scope.getUrlSentiment = function (sourceUrl) {
  // Example: https://twitter.com/search?q=%40chipotle
  var url = "http://access.alchemyapi.com/calls/url/URLGetTextSentiment?"
  var params = {
    url: sourceUrl,
    apiKey: alchemyKey,
    outputMode: "json",
    showSourceText: 0,
    sourceText: "cleaned_or_raw" // removes ads, navigation links, etc.
  }
  http.get(url + s.param(params))
    .success(function (res) {

Figure 27 - AngularJS call to the AlchemyAPI

The response format is JSON, and it will contain an overall type of sentiment and a score. A negative score means the sentiment is negative, zero is neutral and everything higher means the sentiment is positive.

6.4.2 Implementation

Now, let’s put this functionality together and apply it to several sources. Our smart search knows that if the search text starts with the word “Best” and is continued by an entity type, the user wants to do a sentiment analysis of the results given by this search text, e.g., if the user enters “best hotels,” automatically the application will use Factual to retrieve hotels. But, instead of filtering the results by distance or rating, they will be filtered by the highest sentiment score.

Factual Crosswalk contains the twitter entity identifiers for each result, which will help the application construct its specific twitter resource URLs. Each result will make a call to the AlchemyAPI URLGetTextSentiment method. All of these results will be stored directly into each resource object, specifically to a property named SentimentScore.

Now that each resource has its own score, the hotels are displayed on the list view from positive to negative, and on the top of the list will be the one with the highest sentiment score.
This new functionality brings the smart search to a new level, showing an apparent intelligence on the way the places are displayed and affecting how the user makes decisions about the places they select.

Thanks to the sentiment analysis, users will now be able to filter by sentiment, distance and rating. The more options we provide to users, the better decisions they are able to make about what places and information they want to save on their maps.

### 6.5 Caching

The project will be in constant communication with external resources, but we want to keep the amount of API calls as low as possible. For this purpose we will use the AngularJS built-in caching system for http requests (uses browsers cache), which will keep the latest API calls and responses cached until the cache is cleared or it reaches its maximum size limit.

In case an entity is asked to retrieve social media data that was already requested, the project will not ask an external data source anymore. Instead, it will retrieve the response from its cache, improving the application response time dramatically.
Chapter 7

MAPPING

Now that we have covered all of the main functionalities of the project, it is time to explain how all of those features come together into a map and how they are grouped, customized and displayed to the user.

Mapping would not be possible without the use of the Google Maps JavaScript API v3, which includes several Google libraries and services such as Geocoding, Directions and Street View.

The API is loaded on the HTML file by using the script tags and setting the source to Google’s JavaScript URL:

```html
<script src="http://maps.googleapis.com/maps/api/js?extension=.js&output=embed"></script>
```

This JavaScript file loads all of the symbols and definitions we need for using the Google Maps API. The map is a JavaScript class defined by Google, which can take many options, some of which are shown in Figure 28.

```javascript
var mapOptions = {
  zoom: defaultZoom,
  center: symsoftLatLng,
  panControl: false,
  mapTypeControl: true,
  mapTypeControlOptions: {
    style: google.maps.MapTypeControlStyle.HORIZONTAL_BAR,
    position: google.maps.ControlPosition.TOP_RIGHT,
    mapTypeIds: [
      google.maps.MapType.Id.ROADMAP,
      google.maps.MapType.Id.HYBRID
    ]
  },
  zoomControl: true,
  zoomControlOptions: {
    style: google.maps.ZoomControlStyle.SMALL,
    position: google.maps.ControlPosition.RIGHT_BOTTOM
  },
  mapTypeId: google.maps.MapType.Id.ROADMAP,
  styles: [{
    "featureType": "landscape",
    "stylers": [{
      "hue":
```
We can notice initialization options such as the map’s zoom level, types of maps, initial location and styles. But we can also customize its controls, e.g., we decide if we want to display the zoom or pan controls and where on the map we want them to be displayed.

Once the map is loaded, styled and using the right controls, it is time to add locations to it.

7.1 Markers

Each location on the map is identified as a Marker. They have prebuilt constructors and methods defined by Google to simplify their use, but in essence they are still JavaScript objects, which will allow us to define custom properties on them.

Figure 29 displays the JavaScript function that creates markers, and assigns to each of them several properties to help keep them organized.

```javascript
// Creates a marker at given position
function createMarker(position, id, type, data, provider) {
  var marker = new google.maps.Marker({
    position: position,
    animation: google.maps.Animation.DROP,
    map: map,
    icon: getMarkerIcon(type)
  });
  marker.id = id;
  marker.msg = msg;
  marker.type = type;
  marker.data = data;
  marker.provider = provider;
  marker.saved = false;
}
```

The ID property will contain the identifier of the resource, which can vary depending on the provider (Factual or Songkick). A personalized message can be added to each marker, e.g., a bar’s happy hour information, which will be stored on the “msg” property.
As illustrated in Figure 30, the type specifies which kind of entity the marker belongs to (restaurant, hotel, company, artist or landmark). The data property contains a JSON object with the detailed attributes of the specific entity, e.g., a restaurant will have attributes such as cuisine, reservations, hours of operation and more.

For organization purposes, the provider property will either have the value Factual or Songkick. And, last but not least is the saved property, initially set to “false.” When a user decides to save the marker to the map it switches to “true.” It is important to notice from the code above that each marker will have the same color icon as the entity type it represents, making them easy to differentiate once mapped together.

The combination of these properties allows all needed mapping functionalities. Each marker holds enough information to know exactly which action to perform in any given case.

**InfoWindow**

Google Maps API allows the setup of a popover window called InfoWindow. This can be set up so that a popover window is displayed on top of a marker every time a marker is clicked on. Within our project, the InfoWindow will gather important data from the marker using its custom properties and display them each time it is selected.
How do you add a marker to the map?

A button consisting of a pin icon will be displayed on the list view entry of each location and on the InfoWindow of any marker. Once selected, it will automatically change the saved property of the marker. The pin button coloring will determine if the marker is already saved on the map or not, and if clicked it will have a toggle effect. When the color is red the button will unpin the marker from the map, and if it is green the marker will be pinned to the map.

7.2 Layers

Allowing users to add unlimited markers to the map can become disorganized and unmanageable. This is why I created a JavaScript array structure called Layers, which operates under the concept of adding different layers for each type of entity on the map, as evidenced in Figure 31.

Figure 31 - Layers are groupings of markers with custom names

The layers structure allows the grouping of markers, solving the issue of dealing with hundreds of markers at the same time. Each collection can have a custom name, e.g., a user added 25 restaurants, 10 events and 15 landmarks to the map, but once they are all combined in the map it becomes difficult to manage them. In this situation the layers structure will then be used to group the markers according to their entity type, resulting in a layers structure as shown in Figure 32.
Internally a layers structure is nothing more than a multidimensional array. Each position represents a different layer on the map, and contains an array of markers.

As shown in Figure 32, each layer will have on/off capabilities. This will allow users to personalize what entities are displayed on the map. This setting does not eliminate the markers from the map, it just hides them – they can be displayed again simply by selecting the corresponding checkbox.

**How are markers added to a layer?**

When a pin button is selected for any marker or location, a small popover will be shown asking the user which layer they would like to add the marker to. By default a new layer will be created for each different type of entity, but this popover allows users to customize their map layers.

The layers concept is also helpful when a map is shared. It gives the new user an easier way to filter what is shown on the map, allowing an easier understanding of its structure.
7.3 Timeframes

This functionality is for the mapping of events that last through a period of time longer than a day. For example, let’s say we are making a map to capture all of the Sacramento Beer Week events in one easy-to-see map view. We know that for each day during an entire week, there will be several breweries around the city with different drink specials.

To be able to create these types of maps I added timeframes, which operate in a level higher than layers. Their internal structure is still a multidimensional array, in which each position represents a period of time. But instead of saving an array of markers, it will save an array of layers. Figure 33 illustrates how, when combined, layers and time periods create a timeframe.

Figure 33 - Timeframes are groupings of layers displayed on specific dates

The construction of the map is exactly the same as any other map. But once the user is ready to save the state of the map, they will need to select the time period they would like to assign this set of layers to, as shown in Figure 34.

Figure 34 - Presentation of a timeframe
Once the dates are set and the map is saved, the current markers and layers will be cleared, allowing the user to construct a new set of layers with their respective markers for another period of time.

Basically the user will be creating new maps, but instead of saving them independently, they become a set of maps where each layer represents a period of time.

Once a timeframe is created, two navigation arrows will be added to the builder and viewer, allowing the user to navigate through the timeframe’s multidimensional array. Every time a new position in the array is selected, the set of layers will be populated to the map.

A grouping of markers with a custom name is described as a layer, and a grouping of layers with a custom time range is described as a timeframe.

This concept allows for the mapping of a larger set of activities. Music festivals, including concerts in different venues or areas per day, can now be mapped. We want to be able to let the user set their own limits as to what they can map.
Now that we have covered every aspect involved in the creation of maps, it is time to specify how all of this information comes together into a database, and how the JSON format plays a very important role in gathering resources and populating maps.

The project is designed to be lightweight and very fast. Its code consists of a series of calls to external sources and integration of data into a user-friendly interface. Its structure is based in a single HTML file that uses several JavaScript files to retrieve and manage all of its data.

- **Map.js**: JavaScript file that contains all mapping functions such as the creation of markers, layers and communication with the Google Maps API.
- **Controllers.js**: AngularJS code that has a function for every API call to external sources.
- **Scripts.js**: Handlers for each UI element and their communication with the data retrieved from APIs.
- **Index.html**: Project structure, consisting of a fixed navigation bar and a full screen map with several overlays. Contains all calls to JavaScript files and AngularJS directives.

This code organization allowed us to simplify code management and optimization. But let’s talk about the data storage structure of the project.

As we mentioned in earlier chapters, the builder does not store any information locally, which improves the portability of the application. Users can easily access the builder through a webpage and gain access to their maps by a simple log-in process. A similar procedure is used for the viewer: access is granted through a browser without the need to import or upload files to the site.
When a user logs in to the application the first communication to the database occurs. This will validate the user’s credentials and return a list of maps associated with the user. When any map is selected it will request its information to our database, and a JSON-formatted structure is sent to the builder, containing all required features to rebuild the map locally.

8.1 JSON (JavaScript Object Notation)

This lightweight format is the key to the storage organization of the project. We will have two JSON objects that will hold all of the information needed to create a map.

Features JSON

During the map creation stage, global JavaScript objects such as the markers, layers and timeframe arrays will manage all storage. A number of internal JavaScript functions manages each of these objects.

When a user saves a marker or rearranges a layer or timeframe, all they are actually doing is reordering, adding and removing objects from those global arrays. The only time this information is sent to the database is when the user saves the map.

This triggers a method that converts all of this object structure into a JSON string. These three main JavaScript arrays are combined into a single array that contains markers in the first position, layers in the second and timeframes in the last position. This array is called Features, and by itself it contains all information needed to recreate a map.

This is possible thanks to the object-oriented structure of JavaScript, e.g., a timeframe has layers at each position, each layer contains markers and each marker contains enough information by itself to know what type of entity it represents. The attributes information saved in the data and provider property allow the application to know exactly how to recreate each marker. Once we know what entity it represents, we know what action needs to be performed when selected.
This logic is what gives the project a very lightweight and structured storage organization. The features array is then converted into a JSON string using the JavaScript function `JSON.stringify()`, which receives the array as a parameter and returns a string in JSON format that can be saved to the database.

This JSON string can be converted back to the original array structure with the method `JSON.parse()`. These two functions allow maps to be saved and recreated easily.

**Settings JSON**

Now that we stored all features of the map into a JSON string, we will apply the same concept to the map settings, which include: which map controls are displayed and all social media preferences. A dynamic structure such as JSON allows the application to add and remove settings without modifying database fields.

Once the user saves the map, all of the settings stored in global JavaScript variables will be put together into a single settings array. This will then be converted to a JSON string and stored in the database.

**8.2 SQL Server**

Since we’re saving most of the features into JSON strings, there is no need for a large database. The database actually does not need more than two tables to hold all of the information from users and their respective maps.

But why put JSON as text in a relational database?

- Efficiency. Table JOINs can be slow, and since we know we do not need to query for specific features of the map, we can store it in a single object.
Simple implementation. Only a JSON validation needs to be performed before saving the text to the database.

All we need now is one database table to store all of the account information for each user, and another table to save its respective maps information. The database tables are shown in Figure 35.

![Database tables](image)

Figure 35 - Database tables

The user’s table contains only the IDs of the maps created. If any extra information is needed we can reference its respective ID in the Maps table, which will hold all features and settings of the map.

The JSON strings simplify the structure of the database. These strings can be recreated into objects very easily, which allows the builder and viewer to recreate a whole map very fast.

The only thing the viewer needs in order to access a map’s features will be its ID. The JSON strings will contain all of the required information to perform specific actions for each entity type marker.

The key to all functionalities is the information saved in each marker. It is designed to be able to describe all functionalities by itself.

Unfortunately, Microsoft SQL Server has not yet added support for JSON. This is why we use the type NVARCHAR to store each JSON string. Each string can become very large,
depending on the size of the map, but the NVARCHAR can either be set to a fixed length resulting in around 8000 bytes (the setting used for our project), or it can be defined to hold its maximum value (which stores up to 2GBytes of data). This is more than enough to hold any large map. Reaching this limit would be almost impossible since our map structure is considerably small. The storage of the project is based on the JSON format, which avoids the creation of a complicated database structure and simplifies communication between external sources and UI elements. All functions and objects are designed to be reusable, and each API call is cached. All of these features result in a high performance that allows users to easily integrate social network elements, sentiment analysis and worldwide locations into their maps.
Chapter 9

PERFORMANCE & RESULTS

The main goal of the project was to allow users to create meaningful and dynamic maps, thanks to the ability to integrate data from several data sources and add customizations to it we were able to successfully create interactive and personalized maps for users.

In comparison to the tools discussed on chapter 3 we can conclude that our application stands out due to its flexibility, there’s no dependence to a single database or a specific type of entity, we expand our functionalities by basing our limits on the semantic web data, for the scope of the project we perform specific actions only for companies, artists, populated places & architectural structures, which are then separated into more classes. But the single use of a semantic database as our data source allows us to map any kind of information; it’s just a matter of adding actions to them, the backbone and information is ready in place.

Thanks to the storage structure and how we only retrieve database data the first time a map is viewed, the overall performance of the application is very fast. Instead of constantly retrieving data for each marker, the JSON format allows us to lower the database calls to the minimum, adding a low overhead to the application.

Throughout the project we implement caching, if the information of a venue has already been retrieved, then it’ll automatically save on the marker object, so next time the marker is displayed all data will be easily accessed, there’s no waste of information or duplicated API calls.

Even though the project makes use of many datasources the overall performance is high since data is retrieved efficiently. Mapping functions are completed instantly and external API results are constantly reused.
The downside of the current model of the project is the dependence of an Internet connection, we are allowed to keep local storage to a minimum thanks to the retrieval of information from other sources, but if we’re in a low speed network the application’s performance will decrease.

Even though the application can successfully sort venues by sentiment there’s still work to do on the optimization of these tasks, sentiment filtering is achieved by the retrieval of tweets for each venue and construction of a request to the Alchemy API to retrieve the sentiment of the sent URL or text. Testing was able to corroborate that sending the URL of the twitter page instead of using specific tweets separately is faster. If the amount of venues found is very high, it’ll require a couple seconds to filter all entries, which compared to other sentiment analysis tools is much faster.

The project is built with no dependencies of a specific API, there are built-in functions that will trigger the use of backup sources when needed, e.g., if Factual is not available, Yelp will be used instead. This is also applied for the images APIs, in case Flickr cannot retrieve images, Instagram will make sure images can still be displayed for users.

The application is lightweight, portable and fast; the current structure allows code reuse and caching of API calls. And when compared to existing tools it’s suggestion system has no rival due to the range of entities accepted, even though all mapping tools have their special features, none provide support for social media integration, maps can be shared but they have no capabilities to link data and expand its information.

Even though the current use cases have been successful we won’t know if the application will need modifications until we receive enough feedback from users.
Chapter 10

CONCLUSIONS AND FUTURE WORK

This project is an example of the thousands of applications that API integrations can provide. Our approach was mainly focused on mapping, but the current publicly-available resources open up the way for developers to create an unlimited amount of applications in any kind of field.

The semantic web has the potential to revolutionize the way end users capture, communicate and manage information. The integration of several APIs is just an example of how we can enrich existing information, providing users with meaningful data and the possibility to use all of these resources for personal or commercial purposes.

Interactive applications are changing the way users get their information and communicate with each other. Our project was able to create an entity extraction system with sentiment analysis capabilities over social networks, all packed together into a customizable mapping framework.

A project like this would not have been possible a few years ago, but thanks to the rate at which crowd-sourced efforts are collecting information, we can retrieve data from most parts of the world.

This real-time growth in data is what gives our project the possibility of expansion, not only for displaying more information to the user such as menus and reservation systems, but to provide a larger coverage of entity types (e.g., sports teams) and addition of actions (e.g., purchase tickets). Our way to give back to the community is the possibility to share and embed these interactive maps into any website. Eventually we would like to create crowd-sourced maps that can be maintained by the contribution of many users around the world.
Currently, the project is only optimized for desktop usage, but we want to expand it to a mobile-friendly interface where users can contribute with comments and pictures directly from their phones.

An area that can be improved upon is the way information is stored. Although, saving the map features in a JSON string allows for simple implementation, speed and easier integration of changes.

A special database like PostGIS [17] would be preferred. This object-relational database adds support for geographic objects allowing location queries to be run in SQL. This would open the possibility of retrieving information about what is contained in the map features without recreating a map.

This and many other ideas will continue to make this application grow, but for now this project aims to be a learning tool about the powers of the semantic web, sentiment analysis and social media integration.
Appendix A

APIs References

For further information about each of the APIs used on this project, we provide a list of references to its specifications:

**Entity Extraction**

DBpedia Prefix Search

http://wiki.dbpedia.org/Lookup

**Local Businesses**

Factual Global Places, Restaurants & Hotels

http://www.factual.com/data/t/places/schema

http://www.factual.com/data/t/restaurants-us/schema

http://www.factual.com/data/t/hotels-us/schema

Factual Places Crosswalk

http://developer.factual.com/places-crosswalk/

**Social Media**

Foursquare Venue Tips

https://developer.foursquare.com/docs/venues/tips

Twitter Search

https://dev.twitter.com/rest/public/search

Yelp Business Reviews
https://www.yelp.com/developers/documentation/v2/business

Artists Search & Calendar

http://www.songkick.com/developer/artist-search
http://www.songkick.com/developer/upcoming-events-for-artist

Images

Flickr Photo Search

https://www.flickr.com/services/api/flickr.photos.search.html

Instagram Locations Media

https://instagram.com/developer/endpoints/locations/

Sentiment Analysis

Alchemy API

http://www.alchemyapi.com/api/sentiment/textc.html
http://www.alchemyapi.com/api/sentiment/urls.html

Mapping

Google Maps JavaScript API v3

https://developers.google.com/maps/documentation/javascript/tutorial

Google Maps Geocoding

https://developers.google.com/maps/documentation/geocoding/
REFERENCES


