Visualization Framework

Deliverable D4.2: Public

Keywords: OGC, RDF, Geo-Spatial, SPARQL, Visualization

Linked Open Data for environment protection in Smart Regions
# Table of Contents

Table of Contents ................................................................................................................. 2
List of Figures ......................................................................................................................... 4
List of Tables ........................................................................................................................ 5
Executive Summary ................................................................................................................. 7
1 Introduction ........................................................................................................................ 8
   1.1 Document goals ........................................................................................................... 8
   1.2 Document structure .................................................................................................... 8
2 Enabling Technologies ........................................................................................................ 9
   2.1 Overview ..................................................................................................................... 9
   2.2 Linked Data in Nutshell ............................................................................................. 10
   2.3 Sefarad ..................................................................................................................... 12
   2.4 MongoDB: a NoSQL Database .................................................................................... 15
   2.5 GeoServer ................................................................................................................ 16
   2.6 OpenLayers ................................................................................................................ 17
   2.7 Grunt: The JavaScript Task Runner ............................................................................. 18
3 Requirement Analysis ......................................................................................................... 20
   3.1 Overview ..................................................................................................................... 20
   3.2 Actors dictionary ......................................................................................................... 20
   3.3 Use cases .................................................................................................................... 20
     3.3.1 Portal users use cases ............................................................................................ 21
     3.3.2 Admin use cases .................................................................................................... 27
4 Architecture ......................................................................................................................... 29
   4.1 Introduction ................................................................................................................ 29
   4.2 Architecture ................................................................................................................ 29
   4.3 SPARQL Engine .......................................................................................................... 30
     4.3.1 SPARQL Editor ...................................................................................................... 31
     4.3.2 SPARQL queries executor .................................................................................... 32
   4.4 Geo Proxy ................................................................................................................... 33
   4.5 Local Server ................................................................................................................. 36
     4.5.1 Fuseki and Virtuoso .............................................................................................. 36
     4.5.2 Geo Server ............................................................................................................ 36
   4.6 Search and filtering module ......................................................................................... 37
     4.6.1 Keyword search .................................................................................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.2</td>
<td>Faceted search</td>
<td>38</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Geo filtering</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Model View View-Model</td>
<td>39</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Data Model</td>
<td>41</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Widgets layout</td>
<td>41</td>
</tr>
<tr>
<td>4.8</td>
<td>User management module</td>
<td>44</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Security: authentication and authorization</td>
<td>44</td>
</tr>
<tr>
<td>4.8.2</td>
<td>MongoDB: settings and preferences</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1 Linked Open Data cloud diagram example, in LOD project, Cyganiak R. and Jentzsch A. ................................................................. 9
Figure 2 Main Layout - Dashboard. .................................................................................................................................................. 12
Figure 3 Comparison between relational and MongoDB data models................................................................. 15
Figure 4 A collection of MongoDB documents. ......................................................................................................................... 16
Figure 5 Geoserver administration UI. ........................................................................................................................................ 17
Figure 6 OpenLayers map example with multiple layers. ................................................................................................. 18
Figure 7 Use Cases UML diagram. .............................................................................................................................................. 21
Figure 8 General Architecture. ..................................................................................................................................................... 29
Figure 9 SPARQL Editor. .............................................................................................................................................................. 32
Figure 10 SPARQL query sequence diagram. ............................................................................................................................. 33
Figure 11 Fuseki/Virtuoso and GeoServer. ................................................................................................................................. 36
Figure 12 Keyword search example. ........................................................................................................................................ 38
Figure 13 Faceted search example. ........................................................................................................................................... 38
Figure 14 Model View ViewModel diagram............................................................................................................................ 40
Figure 15 Example results table. ................................................................................................................................................. 42
Figure 16 Example tagcloud widget. ......................................................................................................................................... 42
Figure 17 Example map GeoJSON representation. .................................................................................................................. 42
Figure 18 Map for GeoServer data........................................................................................................................................... 42
Figure 19 Linear Layout............................................................................................................................................................... 43
Figure 20 Accordion Layout....................................................................................................................................................... 44
List of Tables

Table 1 Actor list.......................................................................................................................... 20
Table 2 Edit a SPARQL query....................................................................................................... 22
Table 3 Run a SPARQL query....................................................................................................... 22
Table 4 Visual display of the information. ...................................................................................... 23
Table 5 Keyword search.................................................................................................................. 23
Table 6 Faceted search ................................................................................................................... 24
Table 7 Log-in/Log-out.................................................................................................................... 25
Table 8 Customize Sefarad ............................................................................................................ 26
Table 9 Save own configuration...................................................................................................... 26
Table 10 Reset own configuration .................................................................................................. 27
Table 11 Security and users management ...................................................................................... 28
Table 12 Local datasets management ............................................................................................ 28
Executive Summary

This deliverable describes the design, architecture and use of a semantic visualization tool for representing linked geospatial data for the SmartOpenData (SmOD) project. A semantic visualization linked data tool named SEFARAD has been proposed. This tool provides mechanisms for enabling non skilled GIS users to explore and visualize different kinds of linked data resources with a high level description of classes, properties and relations. SEFARAD enables GIS users to apply faceted browsing features to go beyond the data exploration process to apply specialized linked data visualization features. Furthermore, SEFARAD provides a utility for creating and processing human-readable SPARQL queries as a part of the exploration and visualization process. Functionalities provided by the SEFARAD has been evaluated by using geospatial data from Slovak Environment Agency (SAZP). In this evaluation, about 42770 RDF descriptions that contain information about protected parcels in Slovakia, have been processed, analysed and represented.
1 Introduction

One of the most promising fields for Geographical Information Systems (GIS) is the use of Linked Open Data Visualization Tools. This approach refers to a technique for visualizing published, connected and structured information on the web according to the linked data principles (Benedetti, 2014). The benefits of using linked data visualization tools on geographical information systems are highly valued. Visualization tools enable suitable standardized mechanisms for exploring and representing large amounts of geo-spatial information by giving an overview of the datasets, their types, properties and relationships.

Despite of linked data visualization tools helps users to explore large amounts of data and interact with them, visualization tools are currently facing some problems. Representation and visualization of data is performed in ad-hoc model. Moreover, technical and conceptual knowledge are required to take advantage of the benefits provided by the linked data (Peña, 2012). Many linked data visualization tools are exclusively used for custom tailored applications and they cannot be parameterized in order to adapt them to other cases beyond the representation of the studied case. Thus, they are inaccessible for users with more general requirement needs. Many visualization tools are often accompanied by underlying assumptions that unknown to the domain experts or cannot be explicitly characterized for a particular model.

To contribute to improve the above problems, a semantic visualization tool named SEFARAD has been developed. This tool provides mechanisms for enabling non skilled users to visualize linked data sources with a high level description of classes, properties and relations. By using a map faceted navigation capability users are able to go beyond data exploration to map visualization filtering features. Furthermore, SEFARAD provides an utility for creating and processing human-readable SPARQL queries which can be used for representing linked data information.

1.1 Document goals

The aim of this document is to describe the design, architecture and implementation of a semantic visualization tool for representing linked geo spatial data. This document provides an explanation about the operational model as well as components of the framework and the interaction between them. Furthermore, the document provides a developer and user manual for explaining how end users and developers can use and extend the tool.

1.2 Document structure

Section 2 provides an overview of the most important concepts around the Linked Open Data Visualization approach. This section describes some of the most important technologies and tools used for the development process of the SEFARAD tool. Section 3 enumerates the requirements that framework must fulfil in order to reach the goals of the SmartOpenData project. Section 4 introduces the proposed architecture and its functional design by describing some of the most important components. Section 5 illustrates the advantages of the developed tool by explaining a case study. Finally, section 6 summarizes the results and points out the possible future research directions on the proposed framework.
2 Enabling Technologies

2.1 Overview

Linked Data is a technological innovation that transforms the way we think about information and its role in society, in our case geographic information. Linked Data has been recently suggested as one of the best alternatives for creating these shared information spaces (C. Bizer, 2009). Linked Data describes a method of publishing structured and related data so that it can be interlinked and become more useful, which results in the Semantic Web¹ (or Web of Data). It is built upon standard Web technologies such as HTTP, RDF and URIs, but rather than using them to serve web pages for human readers, it extends them to share information in a way that can be read automatically by computers. This enables data from different sources to be connected and queried using SPARQL standard. This is especially important for sophisticated types of information, in particular information with spatial and temporal components.

![Linked Open Data cloud diagram example](http://lod-cloud.net/)

Figure 1 Linked Open Data cloud diagram example, in LOD project, Cyganiak R. and Jentzsch A.².

With the adoption of Linked Data, the traditional complexities of conceptual database schemata for spatial data can safely remain internal to organizations. Their externally relevant contents get streamlined into the open and more manageable form of vocabulary definitions. Users of Linked Data do not need to be aware of complex schema information to use data adequately, but "only" of the semantics and predicates (such as isLocatedIn) occurring in the data. While many questions remain to be answered about how to produce

---

¹ http://www.w3.org/standards/semanticweb/
² http://lod-cloud.net/
and maintain vocabulary specifications, the elaborate layering of syntactic, schematic, and semantic interoperability issues has simplified to a single common syntax (RDF\(^3\)), the irrelevance of traditional schema information outside a database, and a focus on specifying and sharing vocabularies.

This simplification is more dramatic for spatially and temporally referenced data (with their complexities in the form of geometries and scale hierarchies). The resulting paradigm shift, from distributed complex databases accessed through web services that expose schemata to knowledge represented as graphs, whose links can be given well-defined meaning, radically changes some of the long-standing problems of GIScience and GIS practice. Everything said is a way to facilitate analysis and integration of all geographic information available worldwide.

### 2.2 Linked Data in Nutshell

The rise of the Open Data Movement has led to the Web of Data and has grown significantly over the last years. This Web of Data has started to span data sources from a wide range of domains such as people, companies, music, scientific publications, etc. The principles of Linked Data were first outlined by Berners-Lee in 2006 (Berners-Lee, 2006):

- Use URI as names for things.
- Use HTTP URI so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URI so that they can discover more things.

Linked Data is the name for a collection of design principles, practices and technologies centred around a novel paradigm to expose, publish, retrieve, reuse, and integrate data on the Web. In summary, that is simply about using the Web to create typed links between data from different sources. In contrast to the Document Web, the Semantic Web aims at establishing named and directed links between typed data.

For example, a normal Web page about Portsmouth (such as http://en.wikipedia.org/wiki/Portsmouth) may link to another page about Hampshire (such as http://en.wikipedia.org/wiki/Hampshire).

For a machine, the intended meaning of such links is difficult to interpret and the Web pages can only be consumed as integral units of text or other media. On the Linked Data Web, by contrast, the link between Portsmouth and Hampshire would be directed and labelled, for example, forming the statement that Portsmouth is located in Hampshire. Additionally, the two places would be typed, e.g., as city and county, jointly leading to the statement that the city of Portsmouth is located in the county of Hampshire. Finally, the predicate isLocatedIn could be defined as a transitive relation in an ontology. Thus, in conjunction with a statement that Hampshire County is located in the UK, one could automatically derive the new statement that Portsmouth is located in the UK.

\(^3\) http://www.w3.org/RDF/
Technically, Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked from external data sets. That three given elements constitute each piece of information in Linked Data, one refers to such statements as triples, consisting of a subject (Portsmouth), a predicate (isLocatedIn), and an object (Hampshire).

This syntax, which happens to be the simplest form in which statements can be made in natural language, has thus been carried over to the world of data. The data model for triples is the so-called Resource Description Framework. Every entity in the physical world (even a subject, a predicate or an object) should be identified by a global unique URI, and all the information should be provided by using W3C standards such as mentioned RDF or OWL. Linked Data can be queried using SPARQL\(^4\) (an acronym for SPARQL Protocol and RDF Query Language), a query language for RDF which became an official W3C Recommendation\(^5\). The SPARQL query language consists of the syntax and semantics for asking and answering queries against RDF graphs and contains capabilities for querying by triple patterns, conjunctions, disjunctions, and optional patterns. It also supports constraining queries by source RDF graph and extensible value testing. Results of SPARQL queries can be ordered, limited and offset in number, and presented in several different forms, such as JSON, RDF/XML, among others.

**Listing 2 SPARQL query example.**

```sparql
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person a foaf:Person.
  ?person foaf:name ?name.
}
```

\(^4\) [http://www.w3.org/TR/rdf-sparql-query](http://www.w3.org/TR/rdf-sparql-query)

\(^5\) [http://www.w3.org/blog/SW/2008/01/15/](http://www.w3.org/blog/SW/2008/01/15/)
In our particular case of dealing with geographic information, GeoSPARQL\(^6\) enriches SPARQL by quantitative reasoning. Linked Data is usually stored and accessed via SPARQL endpoints (e.g., DBpedia\(^7\) or GeoNames\(^8\)). The ontologies that allow human users and machines to understand which concepts and predicates can be queried, and how they are formally defined, are described using languages such as the Web Ontology Language (OWL\(^9\)).

In the geospatial context, GeoLinked Data\(^10\) is an open initiative whose aim is to enrich the Semantic Web with geospatial data in the context of INSPIRE\(^11\) (Infrastructure for Spatial Information in Europe) Directive. This initiative focuses its efforts on collecting, processing and publishing geographic information from different organizations around the world and providing the suitable tools for handing all the data. Further efforts will have to be done in order to deal with data quality and responsibility, data privacy and data ownership issues within Internet contexts.

### 2.3 Sefarad

Sefarad is a web application developed to explore linked data by executing SPARQL queries to a chosen endpoint without writing code. Thus, it provides a semantic front-end to Linked Open Data (Bermejo, 2014). It allows the user to configure his/her own dashboard with many different widgets to visualize, explore and analyse graphically the different characteristics, attributes and relationships of the queried data. Sefarad is developed in HTML5\(^12\) and follows a Model View-View Model (MVVM) pattern performed with the Knockout framework\(^13\). This JavaScript library allows us to create responsive and dynamic interfaces which automatically is updated when the data changes. The different parts of the UI are connected to the data model by declarative bindings.

Sefarad consists of two different tabs: dashboard and control panel. The first tab allows the user to perform faceted search on the data accessed, so the users can explore a collection of information by applying multiple filters. In the control panel tab statistics about the dataset are visualized.

![Figure 2 Main Layout - Dashboard.](image)
The great potential of Sefarad for SmartOpenData project lies in the capability to create its own widgets really easily. There should not worry about obtaining the filtered data and updating the widget when a new facet is selected thanks to Knockout framework. For this purpose, the application specifies how to create a new JavaScript file in which it should be placed a JavaScript object using D3.js framework\(^\text{14}\). We will take advantage of this feature to develop geographic widgets. The widget template is as follows:

\(^{14}\) [http://d3js.org/](http://d3js.org/)
// New widget
var newWidget = {
    // Widget name.
    name: "Name",
    // Widget description.
    description: "description",
    // Path to the image of the widget.
    img: "path/to/image",
    // Type of the widget.
    type: "type",
    // Help display on the widget
    help: "help",
    // Category of the widget (1: textFilter, 2: numericFilter, 3: graph, 5: results, 4: other, 6: map)
    cat: X,
    render: function() {
        var id = 'A' + Math.Floor(Math.random() * 10001);
        var field = newWidget.field || "";
        vm.activeWidgetsRight.push({
            "id": ko.observable(id),
            "title": ko.observable(newWidget.name),
            "type": ko.observable(newWidget.type),
            "field": ko.observable(field),
            "collapsed": ko.observable(false),
            "showWidgetHelp": ko.observable(false),
            "help": ko.observable(newWidget.help)
        });
        newWidget.paint(id);
    },
    paint: function(id) {
        d3.select(’div’ + id).selectAll(’div’).remove();
        var div = d3.select(’div’ + id);
        div.attr("align", "center");

        // Code to paint
    }
};

Listing 3 Sefarad widget template.
2.4 MongoDB: a NoSQL Database

MongoDB\textsuperscript{15} is an open-source document-oriented NoSQL database distributed under the GNU Affero General Public License\textsuperscript{16} and the Apache License\textsuperscript{17}, written in the programming language C. As a NoSQL database, instead of traditional table-based relational database MongoDB is structured into collections. Those collections are a set of BSON (Binary JSON) documents containing a set of fields or key-value pairs: keys are string and value can be of so many types (string, integer, oat, timestamp, etc.). That provides high performance, high availability, and automatic scaling. Figure 3 shows the possible similarities or equivalences between MongoDB and traditional relational databases.

![Figure 3 Comparison between relational and MongoDB data models.](image)

In MongoDB, the basic piece of data is called a document. See figure 3. A major advantage in MongoDB is that documents do not have a predefined schema (flexible schema). We can think of a document as a multidimensional array whose values could themselves be another array. In practical matters, MongoDB documents have a JSON array structure.

Furthermore, MongoDB is optimized for CRUD operations. You can store as much information as you need in a document without first defining its structure, and the data that will be able to be queried. In order to retrieve one or more documents, you may run your own query specifying some criteria or conditions. A query may support search by field, range or conditional statements such as the existence or not of a key. These make the system highly scalable.

\textsuperscript{15} https://www.mongodb.org/
\textsuperscript{16} http://www.gnu.org/licenses/agpl-3.0.html
\textsuperscript{17} http://www.apache.org/licenses/LICENSE-2.0.html
MongoDB also offers the possibility of replication into two or more replica sets, providing high availability. Every moment one of the replica sets works as the primary and replaces and updates the data of the replicas. When the primary fails, the secondary replica becomes principal. Additionally, MongoDB can run simultaneously over multiple servers, balancing the load between them and keeping those security replica sets and the system running in case of hardware failure.

MongoDB supports drivers for most common programming languages. Due to the fact that the structure of a document is similar to a JSON object and most of programming languages drivers support the management and conversion of JSON datatypes to language-specific structures, it is too easy to communicate and manipulate the data. In the case of this project, we use the PHP\textsuperscript{19} driver for MongoDB.

2.5 GeoServer

GeoServer\textsuperscript{20} (B. Youngblood and S. Iacovella, 2013) is an open source software server written in Java that allows users to view and edit geospatial data. GeoServer functions as the reference implementation of the Open Geospatial Consortium Web Feature Service\textsuperscript{21} standard, and also implements the Web Map Service\textsuperscript{22}, Web Coverage Service\textsuperscript{23} and Web Processing Service specifications\textsuperscript{24}.

Some of the main features of GeoServer are:

- Full compatible with the OGC specifications.
- Easy installation and configuration (no large configuration files needed)
- Multiple formats supported, such as PostGIS or Shapefile.
- Multiple output formats supported, such as JPEG, GIF, PNG, SVG y GML.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{mongo.png}
\caption{A collection of MongoDB documents\textsuperscript{18}.}
\end{figure}

\begin{itemize}
\item \url{http://docs.mongodb.org/manual/core/crud-introduction/}
\item \url{http://php.net/manual/es/book.mongo.php}
\item \url{http://geoserver.org/}
\item \url{http://www.opengeospatial.org/standards/wfs}
\item \url{http://www.opengeospatial.org/standards/wms}
\item \url{http://www.opengeospatial.org/standards/wcs}
\item \url{http://www.opengeospatial.org/standards/wps}
\end{itemize}
ECQL query language support.

GeoServer also includes an administration UI from which users can manage the stored data, observe and analyse the different attributes of the information as well as a preview of the different layers for what GeoServer includes an integrated OpenLayers client. In our project we save our geospatial dataset into a GeoServer installation and display the information with maps integrated into Sefarad as a new widget developed with OpenLayers.

![Figure 5 Geoserver administration UI.](image)

### 2.6 OpenLayers

OpenLayers\(^\text{25}\) is the most complete and powerful open source JavaScript library to create any kind of web mapping application. OpenLayers was originally developed and released by MetaCarta under a BSD license.

In addition to offering a great set of components, such as maps, layers, or controls, OpenLayers offers access to a great number of data sources using many different data formats, implements many standards from Open Geospatial Consortium (OGC), and is compatible with almost all browsers.

Thus, OpenLayers allows us to display the geographical information stored in all major and common data servers into functional and interactive maps. This means the users can connect your client application to web services spread, add data from a bunch of raster and vector file formats such as GeoJSON and GML, and organize them in layers to create original web mapping applications.

OpenLayers provides lots of controls such as pan, zoom, and query the map to build interactive maps which give users the possibility to actually explore the maps and the geospatial data display on them. OpenLayers allows you to include as many layers as you want, each representing a piece of information. Each layer can be customized with different

\(^{25}\) http://openlayers.org/
colours, transparency, shadows, alive and clicking information, etc., and can be shown or hidden every moment. There are two kinds of layers in OpenLayers: base and non-base. Base layers are always visible and determines some of the essential properties of the map (zoom, centre, etc.). A map can have more than one base layer but only one of them can be active at a time.

In the case of GeoLinked Data, OpenLayers provides us the necessary tools to represent geographical information stored in GeoJSON in a map. GeoJSON is a format for encoding a variety of geographic data structures and supports multiple geometry types, such as Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon.

![OpenLayers map example with multiple layers.](image)

**2.7 Grunt: The JavaScript Task Runner**

GruntJS (Hogan, 2014) is a JavaScript task runner written with Node.js used to automate predefined tasks to ease the development and integration of our project and to save time automating repetitive tasks.

Grunt provides lots of plugins that are installed and manage via npm, Node.js package manager, which allows us to automate some manual repetitive tasks we run as part of our development or deployment process. Those plugins are labelled contrib packages, which mean they are branded as officially maintained and stable. Moreover, users share their own plugins and everyone can easily create their own user-defined task plugin if there is no one for the task they want to automate.

To automate your project with Grunt, you must implement your Gruntfile.js configuration file and a package.json file. In the configuration file we indicate the tasks we want to automate and load the corresponding plugins with a simple command and configure them.

---

26 http://geojson.org/
27 http://gruntjs.com/
28 http://nodejs.org/
29 https://www.npmjs.org/
with JSON format. The package.json file is used to list grunt and the Grunt plugins your project needs as npm devDependencies.

Once the two mentioned files have been created, each time grunt is run it looks for a locally installed Grunt. When it is found, the CLI (Grunt’s Command Line Interface) loads the local installation of the Grunt library, applies the configuration from your Gruntfile, and executes any tasks you've requested for it to run.
3 Requirement Analysis

3.1 Overview

This section describes a complete specification of the requirements, which will be matched by each module in the design stage. This helps us also to focus on key aspects and take apart other less important functionalities that could be implemented in future works.

3.2 Actors dictionary

The list of primary and secondary actors is presented in table 1. These actors participate in the different use cases, which are presented later.

<table>
<thead>
<tr>
<th>Actor identifier</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT-1</td>
<td>Portal User</td>
<td>End user without technical knowledge on Semantic Technologies that uses Sefarad to query a SPARQL endpoint, display the retrieved information and use the faceted search. These users need an intuitive and clear interface and an appropriate help section.</td>
</tr>
<tr>
<td>ACT-2</td>
<td>Advance User</td>
<td>End user with technical knowledge on Semantic Technologies. These users can edit their own SPARQL queries and use more complex configurations because of their knowledge.</td>
</tr>
<tr>
<td>ACT-3</td>
<td>Admin</td>
<td>Administrator of Sefarad, in charge of tasks such as security management, inserting and deleting datasets in local database, etc.</td>
</tr>
</tbody>
</table>

Table 1 Actor list.

3.3 Use cases

This section identifies the use cases of the system. This helps us to obtain a complete specification of the uses of the system, and therefore define the complete list of requisites to match. First, we will present a list of the actors in the system and a UML diagram (Figure 7) representing all the actors participating in the different use cases. This representation allows us to specify the actors that interact in the system and the relationships between them.

These use cases will be described in the next sections, including each one a table with their complete specification. Using these tables, we will be able to define the requirements to be established.

The next graphic represents all the use cases involved in this project in a UML diagram.
3.3.1 Portal users use cases

The use cases presented in this section are those related to all the portal users. These are:

- Edit SPARQL queries detailed in (Section 3.3.1.1).
- Run SPARQL queries detailed in (Section 3.3.1.2).
- Visual display of the information detailed in (Section 3.3.1.3).
- Keyword search detailed in (Section 3.3.1.4).
- Faceted search detailed in (Section 3.3.1.5).
- Log-in/Log-out detailed in sub-section 3.3.1.6).
- Customize Sefarad detailed in (Section 3.3.1.7).
- Save own configuration detailed in (Section 3.3.1.8).
- Reset own configuration detailed in (Section 3.3.1.9).
### Table 2 Edit a SPARQL query.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Edit SPARQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.1</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Advanced User</td>
</tr>
<tr>
<td>Flow of Events</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>The user selects the SPARQL Editor to write his own query</td>
</tr>
<tr>
<td></td>
<td>A new text-box is opened for the user to write the query</td>
</tr>
<tr>
<td>2</td>
<td>The user writes the query</td>
</tr>
<tr>
<td></td>
<td>The SPARQL Editor assists the user by displaying different valid options for the query or highlighting errors</td>
</tr>
<tr>
<td>3</td>
<td>The user saves the query</td>
</tr>
<tr>
<td></td>
<td>The new query is saved for later execution</td>
</tr>
</tbody>
</table>

### Table 3 Run a SPARQL query.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Run SPARQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.2</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal User</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The user has selected or edit a SPARQL query</td>
</tr>
<tr>
<td>Flow of Events</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>The user selects a SPARQL endpoint</td>
</tr>
<tr>
<td></td>
<td>The configuration of the application is updated so the following queries will be run against the selected endpoint</td>
</tr>
<tr>
<td>2.1</td>
<td>The user executes the query</td>
</tr>
<tr>
<td></td>
<td>The SPARQL query is executed against the selected endpoint</td>
</tr>
<tr>
<td>2.2</td>
<td>The server responds with results data. The data available is updated with the information retrieved and all the widgets are automatically updated</td>
</tr>
</tbody>
</table>
### Table 4 Visual display of the information.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Visual display of the information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.3</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal User</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The application has received response data from a SPARQL query</td>
</tr>
<tr>
<td>Flow of Events</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Actor Input</td>
</tr>
<tr>
<td></td>
<td>The user adds a new widget selecting which facet or facets to show</td>
</tr>
</tbody>
</table>

### Table 5 Keyword search.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Keyword Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.4</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal User</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The application has received response data from a SPARQL query and it has been indexed</td>
</tr>
<tr>
<td>Flow of Events</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Actor Input</td>
</tr>
<tr>
<td></td>
<td>The user expresses her goals by typing textual keywords into the search box</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Use Case Name</td>
<td>Faceted Search</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Use Case ID</td>
<td>UC1.5</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal End-User</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The application has received response data from a SPARQL query and it has been indexed</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Actor Input</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The user adds a new faceted search widget selecting which facet to filter</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>The user selects a new values for this facet to filter by</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6 Faceted search.
<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Log-in/Log-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.6</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal User &amp; Admin</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The application has been initialized without any user logged in</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>The user can log out any moment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow of Events</th>
<th>Actor Input</th>
<th>System Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The user introduces his credentials (username and password)</td>
<td>On the server-side the application finds matches for the username/password introduced in MongoDB users database</td>
</tr>
<tr>
<td>2.a</td>
<td>The user introduces valid credentials</td>
<td>The user is authenticated. The interface of the application is updated to the admin interface, showing hidden buttons such as 'add new widget', 'delete widget', 'save configuration', etc.</td>
</tr>
<tr>
<td>2.b</td>
<td>The user introduces no valid credentials</td>
<td>An alert message reports that authentication has failed.</td>
</tr>
<tr>
<td>3</td>
<td>The user clicks 'logout' button</td>
<td>The session is closed and the application turns back to no-admin interface</td>
</tr>
</tbody>
</table>

Table 7 Log-in/Log-out.
### Table 8 Customize Sefarad.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Customize Sefarad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.7</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal User</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The application has been initialized and the user has logged in</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Actor Input</td>
</tr>
<tr>
<td></td>
<td>1.1 The user adds a new widget</td>
</tr>
<tr>
<td></td>
<td>1.2 The user removes a widget</td>
</tr>
<tr>
<td></td>
<td>1.3 The user configures a widget</td>
</tr>
<tr>
<td></td>
<td>1.4 The user reorders the widget layout</td>
</tr>
<tr>
<td></td>
<td>1.5 The user changes the global configuration</td>
</tr>
</tbody>
</table>

### Table 9 Save own configuration.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Save configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC1.8</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Portal User</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The application has been initialized and the user has logged in</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Actor Input</td>
</tr>
<tr>
<td></td>
<td>1 The user clicks ‘save configuration’ button</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
3.3.2 Admin use cases

This use case package collects the use cases related to admin users. The use cases presented in this section are:

- Security/Users Management detailed in (Section 3.3.2.1).
- Local Datasets Management detailed in (Section 3.3.2.2).
### Table 11 Security and users management.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Security/Users Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC2.1</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Admin</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The administrator of the system has logged-in as admin into the MongoDB database</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Actor Input</td>
</tr>
<tr>
<td>1.1</td>
<td>The admin inserts a new user into the database</td>
</tr>
<tr>
<td>1.2</td>
<td>The admin removes a user from the database</td>
</tr>
<tr>
<td>1.3</td>
<td>The admin edits user permissions or information</td>
</tr>
</tbody>
</table>

### Table 12 Local datasets management.

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Local datasets management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case ID</td>
<td>UC2.2</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>Admin</td>
</tr>
<tr>
<td>Pre-Condition</td>
<td>The administrator of the system has logged-in as admin into the local database</td>
</tr>
<tr>
<td>Flow of Events</td>
<td>Actor Input</td>
</tr>
<tr>
<td>1.1</td>
<td>The admin inserts data into the local dataset</td>
</tr>
<tr>
<td>1.2</td>
<td>The admin removes data from the dataset</td>
</tr>
<tr>
<td>1.3</td>
<td>The admin updates the local dataset</td>
</tr>
</tbody>
</table>
4 Architecture

4.1 Introduction

In this chapter we show a detailed diagram about the complete architecture of Sefarad (see figure 8). In the first section we introduce the scheme, the behaviour and the main function of each of the modules and components. After this, in the following subsections we describe each module in depth showing specific diagrams, screenshots and detailing their particular operation.

4.2 Architecture

A diagram of the architecture is shown in Figure 8. Each module is detailed in the following sections.

Since the main purpose of this master thesis is to develop a HTML5 Framework to query, manage and represent Geo Linked Data, we need a SPARQL Engine to edit and execute the queries to the endpoint we want and retrieve the data. Non-technical users can edit their queries by using a SPARQL editor which will help them with recommendations and corrections while editing the query and advanced users can edit their own queries. The application allows the users to query any semantic repository with a corresponding SPARQL endpoint or any local dataset within a local database such as Fuseki and Virtuoso or Geo
Server. In order to get the data in a proper format for Sefarad, the Geo Proxy module handles the conversion of the data when it is needed.

Once the data is retrieved, the Search and filtering module provides us the necessary tools to manage the queried data enabling Faceted search, Keyword search and Geo filtering. The application automatically indexes, sorts and classifies the information, obtaining the different facets and values of the data. All the filtering services are provided to the user in an intuitive graphical interface by providing multiple widgets, a search box and a sortable table of results. To handle the changes in the filtered data due to the different filters selected by the user, the Model View View-Model module uses the features offered by Knockout framework. Every time a new search or filter criteria is included, the data model will be automatically updated with the results that meet the new conditions and all the widgets displayed in the layout will be redrawn using updated final results.

For the security and administration tasks, we need a User management module. This module includes a Security: authentication and authorization sub-module based on PHP and MongoDB. The user's username and password (encoded in MD5 hash) are stored in a MongoDB collection named users. When a user wants to log in, the application checks his user credentials with a PHP5 script. In case of success, depending on the users permissions (admin, basic user, etc.), the different tools and options are shown or hidden (i.e. add, configure and delete widgets). Furthermore, the user preferences and settings are stored in another MongoDB collection, so that when a user logs in the application is configured using the last configuration saved by the user. This is managed by MongoDB: settings and preferences sub-module.

Finally, the Setup module provides a graphical installer for an easy deployment of the application in any computer running a Linux operating system, installing anything needed to run Sefarad. This module includes two sub-modules: a Custom installer, which allows the user to select which modules to include in its installation; and an automation module, to automate certain repetitive tasks with a single command.

4.3 SPARQL Engine

The SPARQL engine is the main module for managing the SPARQL queries. It provides the necessary tools for editing and running our own queries to the selected dataset, what is the first step in the indexing and processing of the information requested.

Sefarad allows the user to query different kinds of datasets. The user can query a local dataset stored into a geographic server such as GeoServer or a local RDF database such as Fuseki or Virtuoso. The user can also select a SPARQL endpoint provided by any website to query their public datasets. All these options can be configured in the SPARQL tab in configuration window. After selecting the endpoint, the user can write his/her own query and execute it. Data retrieved in response to the query will be stored in the application as a JSON object. This data will be managed by both the Search and filtering module and the Model View View-Model in order to allow all the filtering features, the display and the update of the final results and widgets.
The SPARQL Engine module consists of two submodules: SPARQL Editor, which supports the user to write his own query by showing errors in the query language or proposing valid options; and the SPARQL queries executor, which allows the user to run the query and retrieve the data.

### 4.3.1 SPARQL Editor

This sub-module allows users to write and edit their own query. Those basic users with no full technical knowledge about SPARQL query language can make use of the SPARQL Editor based on Yasqe\(^\text{30}\) provided by Sefarad. They can access it in one of the principal tabs of the application. This service provides a simple syntax highlighted text area, bundled with features such as auto completion, and the option to query SPARQL endpoints. So, a non-advanced user can edit his own query with the assistance of the application. The complete list of features is presented below:

- Query syntax highlighting and checking
- Accessing -all- endpoints (including CORS-disabled ones, or endpoints on your localhost)
- Prefix auto completion (using prefix.cc\(^\text{31}\))
- Endpoint search and auto completion
- Endpoint search and auto completion (Using CKAN\(^\text{32}\) and the Mondeca Endpoint Status Catalogue\(^\text{33}\))
- Query permanent links
- Persistent application states between user sessions
- SNORQL-type navigation

The following screenshot shows the SPARQL Editor in Sefarad.

\(^{30}\) http://yasqe.yasgui.org/
\(^{31}\) http://prefix.cc/
\(^{32}\) http://datahub.io/
\(^{33}\) http://labs.mondeca.com/sparqlEndpointsStatus/
On the other hand, those advanced users with technical knowledge about SPARQL syntax can write their queries without the help of the editor directly in the SPARQL section of the configuration window.

### 4.3.2 SPARQL queries executor

This module is responsible for executing the query and store the information retrieved. The users can select the information source (local dataset: Fuseki or GeoServer; or any SPARQL endpoint) and run the query. After receiving the response from the server, this module will index the retrieved data and will work with it using the rest modules.
4.4 Geo Proxy

The Geo Proxy Module is responsible for processing the data and converting it to a proper format suitable for its management with Sefarad. In the case of geographic information, it is particularly important the conversion from SPARQL response data to GeoJSON data, the proper format for representing geographic information in an Openlayers map. For this purpose, we have developed the next converter module shown in listing 4. The function takes a SPARQLJSON object and processes it to return a GeoJSON object, which can be easily read and represented with an Openlayers map.
function sparqlToGeoJSON(sparqlJSON) {
  'use strict';
  var bindingindex, varindex, geometryType, wkt, coordinates, property;
  var geojson = {
    "type": "FeatureCollection",
    "features": []
  };

  for (bindingindex = 0; bindingindex < sparqlJSON.length; ++
    bindingindex) {

    for (var key in sparqlJSON[bindingindex]) {

      if ((sparqlJSON[bindingindex][key].datatype != undefined) &&
          (sparqlJSON[bindingindex][key].datatype() == "http://www.
            opengis.net/ont/geo sparql# wktLiteral" ||
          sparqlJSON[bindingindex][key].datatype() == "http://www.
            openlinksw.com/schemas/virt rdf# Geometry")) {
        // assumes the well-known text is valid!
        wkt = sparqlJSON[bindingindex].value();

        // find substring left of first "(" occurrence for geometry
type
        switch (true) {
          case /POINT*/.test(wkt.substr(0, wkt.indexOf("("))):
            geometryType = "Point";
            coordinates = coordinates.substr(1, coordinates.length
              - 2); // remove redundant [ and ] at beginning and
            end
            break;
          case /MULTIPOINT*/.test(wkt.substr(0, wkt.indexOf("("))):
            geometryType = "MultiPoint";
            break;
          case /LINESTRING*/.test(wkt.substr(0, wkt.indexOf("("))):
            geometryType = "LineString";
            break;
        }
      }
    }
  }
}
```javascript
var feature = {
    "type": "Feature",
    "geometry": {
        "type": geometryType,
        "coordinates": eval('(' + coordinates + ')')
    },
    "properties": sparqlJSON[bindingindex]
};
geojson.features.push(feature);
```

Listing 4 SPARQL to GeoJSON conversion.
4.5 Local Server

The main feature of this module is to provide a local database to store our datasets. The user can upload his data to a local database to query and manage it with Sefarad, configuring it in the SPARQL tab in the configuration section, instead of using an external endpoint. It can be RDF data, for whose storage we use Fuseki and Virtuoso, or shapefiles data, for which we use GeoServer. Figure 11 shows the different ways to access these databases and represent the results with Openlayers.

![Diagram of Fuseki/Virtuoso and GeoServer](http://geoserver.org/)

**Figure 11 Fuseki/Virtuoso and GeoServer.**

4.5.1 Fuseki and Virtuoso

In either of these two databases we can store our datasets in RDF format. They both provide us an API REST to perform the queries and retrieve the data. Before storing the information in Sefarad, it should be treated in the Geo Proxy.

4.5.2 Geo Server

GeoServer (B. Youngblood and S. Iacovella, 2013) is an open source software server written in Java that allows users to view and edit geospatial data. Some of the main features of GeoServer are:

- Full compatible with the OGC specifications.
- Easy installation and configuration (no large configuration _les needed)
- Multiple formats supported, such as PostGIS or Shapefile.
- Multiple output formats supported, such as JPEG, GIF, PNG, SVG y GML.

---

34 http://geoserver.org/
- ECQL query language support.

GeoServer also includes an administration UI from which users can manage the stored data, observe and analyse the different attributes of the information as well as a preview of the different layers for what GeoServer includes an integrated OpenLayers client.

The user can save his geospatial dataset into a GeoServer installation and display the information with maps integrated into Sefarad as a new widget developed with OpenLayers.js. Into this server the user can upload geographic dataset in specific GIS formats such as DBF or Shapefile.

4.6 Search and filtering module

Once the information is retrieved and stored in a proper format, the 'Search and Filtering' module provides the necessary tools to categorize, filter and sort that information. For this purpose, this module provides different features:

- Back-end technology to manage and index the data, its different facets and the filtering features, updating the filtered results when the user selects new filtering criteria.
- A results table which shows the final filtered results and all the filters selected by the user, allowing him to manage both the filters and the results.
- Keyword search box, in which the user can introduce the words he wants to search by. This feature is fully detailed in subsection 4.6.1.
- 'Tag cloud' and 'vertical layout' filtering widgets, which show the different values for a particular facet, allowing the faceted search detailed in subsection 4.6.2. The user can configure all these widgets.
- Different many widgets to filter by numeric or geographic facets.

To manage and update the filtered data and the widgets in the layout when the user selects new filters, this module makes use of the advantages of Knockout.js. When a new filter is selected (either in a widget or in the search box), it is added to a collection that contains all the filters selected. When this collection changes, due to the knockout bindings, the filtered data is updated with the data that satisfy the new criteria. After that, all the active widgets are redrawn with the new filtered data available. Figure 12 and Figure 13 show examples of filtering.

4.6.1 Keyword search

This sub-module allows the user to filter the information by a keyword criterion. The user can introduce his search in a text-box shown in the top right corner of the application. This criterion will be used by the application to find results that contain the entered text in any of its fields.
4.6.2 Faceted search

This feature (also known as faceted browsing) allows the user to access the information according to a faceted classification system, applying multiple filters. A faceted classification system classifies each information element along multiple explicit dimensions, called facets, enabling the classifications to be accessed and ordered in multiple ways. Facets correspond to properties of the information elements. They are derived by analysis of the different keys of the retrieved data (in JSON format).

4.6.3 Geo filtering

Due to the geographical purpose of this master thesis, a geo filtering module is needed. This module works in a similar way of the two modules above, but focusing on geographical terms. Thus, this sub-module allows the management and filtering of the information on aspects such as latitude, longitude, area, etc.
With respect to the Openlayers maps, this module includes features such as layers classification, so that the different information can be categorized into different layers that can be shown or hidden on these maps (restaurants, hospitals, urban areas, and etcetera).

Regarding to the data stored in GeoServer, this module includes a tool that allows the user to filter the information available on the server by using CQL or ECQL\(^35\) query language. CQL (Common Query Language) is a query language created by the OGC for the Catalogue Web Services specification\(^36\). Unlike the XML-based Filter Encoding language, CQL is written using a familiar text-based syntax. It is thus more readable and better suited for manual authoring. GeoServer provides an extended version of CQL called ECQL, which removes the limitations of CQL, providing a more flexible language with stronger similarities with SQL. The user can include in his query any of the attributes included in the layer, both text and numeric fields. Also, the user may employ any of the comparison operators (=, <>, >, <, >=, <=), filter by range (BETWEEN X AND Y). Comparisons can be established between an attribute and a value (numeric or textual), or between two attributes of the layer (e.g., ATTRIBUTE1 > ATTRIBUTE2). Furthermore, this query language includes geometric filters capabilities, such as intersections or crosses. The full list of geometric predicates is: EQUALS, DISJOINT, INTERSECTS, TOUCHES, CROSSES, WITHIN, CONTAINS, OVERLAPS, RELATE, DWITHIN, BEYOND.

4.7 Model View View-Model

As we detailed before, we use the Knockout.js framework, which is a JavaScript library that helps us to create rich, responsive display and editor user interfaces with a clean underlying data model. Any time we have sections of UI that update dynamically, KO helps us implement it more simply and maintainable.

The Model View ViewModel (MVVM\(^37\)) is an architectural pattern used in software engineering. Largely based on the model-view-controller pattern (MVC), MVVM is a specific implementation targeted at UI development platforms which support event-driven programming. The main features of this module are: dependency tracking, automatically updates the right parts of our UI (widgets layout) whenever our data model changes; declarative bindings, a simple way to connect parts of our UI to our data model.

\(^35\) http://docs.geoserver.org/2.5.x/en/user/_liter/index.html
\(^36\) http://www.opengeospatial.org/standards/cat
\(^37\) http://en.wikipedia.org/wiki/Model View ViewModel
Figure 14 Model View ViewModel diagram

MVVM facilitates a clear separation of the development of the graphical user interface (as markup language or GUI code) from the development of the back end logic known as the model (also known as the data model to distinguish it from the view model). The view model of MVVM is a value converter, meaning that the view model is responsible for exposing the data objects from the model in such a way that those objects are easily managed and consumed. In this respect, the view model is more model than view, and handles most if not all of the view's display logic (though the demarcation between what functions are handled by which layer is a subject of ongoing discussion and exploration). The view model may also implement a mediator pattern organising access to the backend logic around the set of use cases supported by the view.

MVVM was designed to make use of data binding functions in to better facilitate the separation of view layer development from the rest of the pattern by removing virtually all GUI code ("code-behind") from the view layer. Instead of requiring user experience (UX) developers to write GUI code, they can use the framework mark-up language and create bindings to the view model, which is written and maintained by application developers. This separation of roles allows interactive designers to focus on UX needs rather than programming of business logic, allowing for the layers of an application to be developed in multiple work streams for higher productivity.

By using knockout bindings in our HTML document, this module automatically updates the DOM elements whenever the vm.filteredData() array changes. An example of a knockout binding in Sefarad is show in listing 5.

---

The code above shows the data-binding for all the widgets at right column. As you can see at the `foreach: activeWidgetsRight binding`, the `widgets-template` is filled with every active widget. When the user adds a new widget, a new object is pushed. Due to the data-binding, the DOM object is automatically updated and a new HTML DIV for the new widget is added. That is how the model view-model works.

### 4.7.1 Data Model

The view model is defined in the file `js/mvvm.js`. It is a "model of the view", meaning it is an abstraction of the view that also serves in mediating between the view and the model which is the target of the view data bindings. It could be seen as a specialized aspect of what would be a controller (in the MVC pattern) that acts as a converter that changes model information into view information and passes commands from the view into the model. The view model exposes public properties, commands, and abstractions.

Thus, the view model contains of the public properties (functions and ko.observable) that will be accessed from the other parts of the project, directly or through data bindings.

### 4.7.2 Widgets layout

One of the main purposes of this project is to show the queried semantic data. To this end Sefarad includes a large library of widgets to display many kinds of information: filtering widgets, slider widget, graphic widgets (bars, wheels, donuts, etc.). To visualize geographic information, the most important widgets are: results table, filtering widgets (tagcloud and selector), Openlayers map for GeoJSON and Openlayers map for GeoServer shapefiles. We describe their main features below.
Results table.

This widget shows the total filtered results. The user can show or hide the different columns of the data, sort the elements by any of the attributes and search any result. If any result contains a link to the URI of its web resource, it can be directly accessed by clicking on it, so it will open a new tab.

Tagcloud widget.

This widget shows the total different values for the selected facet. It allows the user to select one or more values and filter the results, what we presented as faceted filtering or faceted browsing.

Map for GeoJSON data.

This is an Openlayers map which can represent any geospatial and geometric information in GeoJSON format. It includes POINT, MULTIPOLYGON, POLYGON, MULTIPOLYGON, LINE, etc... For this purpose, this widget uses the `geojson_format.read()` function provided by Openlayers.js. This map also includes the possibility of group the different types of data into different layers, so the user can show or hide the different kinds of results.

Map for GeoServer data.

This map allows the user to display the different layers (file formats .dbf and .shp) stored on GeoServer. An important feature is that this widget includes a configuration section to manage ECQL queries in which the user can edit and run his own queries and display the results into the map.
Furthermore, Sefarad offers two different layout templates that the user can select according to his needs: a linear layout and an accordion layout.

The linear layout is the basic style layout for Sefarad. In this template the application is divided in two columns, with the right column width bigger than the left one. Each widget consists on its own division and item in the screen. If the sortable widgets option is enabled, any of the widgets can be placed either in both columns. As the right column is bigger, it is the best option to place the more visual widgets such as the different maps or the results table. In the other hand, the left column is better option for simpler widgets such as filtering widgets, which not require a large field of view.

The accordion layout is a special style layout for Sefarad. It is also divided into two columns, each one with the same dimensions as in the previous case. The difference is that all the widgets in the left column are packed into an accordion widget. This option is specially designed to simplify the layout when the user wants to display a large amount of filtering widgets, so that the user can compress all these widgets into an accordion on the left of the screen, while the right column shows the results and map widgets.
4.8 User management module

This module provides all the necessary features for users control and security. It includes an installation of a MongoDB database in which the application stores user authentications and authorization information (username, password, role and permissions) and user preferences and settings. This module also includes all the server-side technology necessary to manage user login/logout, load saved settings and saving new preferences. For this purpose, the system uses the PHP driver for MongoDB databases management.

4.8.1 Security: authentication and authorization

As we introduced above, this module contains PHP sub-modules and a MongoDB collection in the Sefarad MongoDB database. For security reasons, the passwords of users are stored in the database md5 hashed.

For session handling, we have developed a Session Manager as explained in (Islam, 2011). The security module provides a session manager in PHP, a module that will handle the HTTP session of a user in Sefarad, using MongoDB for storing the session data. It provides us important functionalities such as signing in a user (authentication), authorizing his/her actions, and logging him/her out. We use object-oriented programming principles for building the module, so that it is easy to maintain. We also build a separate module for user authentication, which is used by the session manager for logging a user in.

We need to implement three php classes:

- dbconnection.php, detailed in section [4.8.1.1].
- session.php, detailed in section [4.8.1.2].
The class DBConnection handles the connection with the MongoDB database. You can see the code in listing 6 below.

```php
<?php

class DBConnection {
    const HOST = 'localhost';
    const PORT = 27017;
    const DATABASE = 'sefarad';
    private static $instance;
    public $connection;
    public $database;

    private function __construct() {
        $connectionString = sprintf('mongodb://%s:%s', DBConnection::HOST, DBConnection::PORT);
        try {
            $this->connection = new Mongo($connectionString);
            $this->database = $this->connection->selectDB(DBConnection::DATABASE);
        }
        catch (MongoConnectionException $e) {
            throw $e;
        }
    }

    static public function instantiate() {
        if (!isset(self::$instance)) {
            $class = __CLASS__;
            self::$instance = new $class;
        }
        return self::$instance;
    }

    public function getCollection($name) {
        return $this->database->selectCollection($name);
    }
}
</div>
```

Listing 6 dbconnection.php.

Calling the initialize() static method on this class returns an instance of it, and we can then select a collection by invoking the get Collection() method on this instance. The
DBConnection class implements the Singleton\(^\text{40}\) design pattern. This design pattern ensures that there is only a single connection open to MongoDB, within the context of a single HTTP request.

This module uses a collection in a MongoDB database for storing/retrieving/handling sessions. It extends the session handling with session_set_save_handler(), a function which allows us to define our own functions for storing and retrieving session data. The function takes six arguments, each one being the name of a callback function for handling sessions (open, close, read, write, destroy and gc). Let's see each callback in detail:

- **open():** This method is called whenever a session is initiated with session_start(). It takes two arguments, the path to where the session will be stored and the name of the session cookie. It returns TRUE to indicate successful initiation of a session.

- **close():** This is called at the successful end of a PHP script using session handling. This also needs to return TRUE.

- **read():** This method is called whenever we are trying to retrieve a variable from the \$_SESSION super global array. It takes the session ID as an argument and returns a string value of the \$_SESSION variable.

- **write():** This function is executed whenever we are trying to add or change something in \$_SESSION. This takes the session ID and the serialized representation of the data to be stored in \$_SESSION as its two arguments.

- **destroy():** This is called whenever we are trying to terminate a session by calling the built-in session_destroy() method. It takes the session ID as its only parameter and returns TRUE upon success.

- **gc():** This function is executed by the PHP session garbage collector. It takes the maximum lifetime of session cookies as its argument, and removes any session older than the specified lifetime. It also returns TRUE on success. The session.gc_probability setting in php.ini specifies the probability of the session garbage collector running.

The final code for the session.php class is shown in listing 7

```php
<?php
require_once ('dbconnection.php');

class SessionManager{
    const declaration;
    private $_mongo, $collection, $_currentSession;

    public function __construct() {
        $this->_mongo = DBConnection::instantiate();
        $this->_collection = $this->_mongo->getCollection(
            SessionManager::COLLECTION);
        session_set_save_handler(array(&$this,'open'), array(&$this,'close'),
            array(&$this,'read'), array(&$this,'write'),
            array(&$this,'destroy'), array(&$this,'gc'));
        ini_set('session.gc_maxlifetime', SessionManager::
            SESSION_LIFESPAN);
        session_set_cookie_params(SessionManager::SESSION_LIFESPAN,
            SessionManager::SESSION_COOKIE_PATH, SessionManager::
            SESSION.COOKIE.DOMAIN);
        session_name(SessionManager::SESSION_NAME);
        session_cache_limiter('nocache');
        session_start();
    }

    public function open($path, $name) { return true; }

    public function close() { return true; }

    public function read($sessionId) {
        $query = array(
            'session_id' => $sessionId,
            'timeout_at' => array('$gte' => time()),
            'expired_at' => array('$gte' => SessionManager::
                SESSION_LIFESPAN));
        $result = $this->_collection->findOne($query);
        $this->_currentSession = $result;
        if (!isset($result['data'])) { return ''; }
        return $result['data'];
    }
}```
public function write($sessionId, $data) {
    $expired_at = time() + self::SESSION_TIMEOUT;
    $new_obj = array(
        'data' => $data,
        'timeout_at' => time() + self::SESSION_TIMEOUT,
        'expired_at' => (empty($this->_currentSession)) ? time() +
            $this->SessionManager::SESSION_LIFESPAN : $this->_currentSession[
            'expired_at']
    );
    $query = array('session_id' => $sessionId);
    $this->_collection->update($query, array(
        '$set' => $new_obj
    ), array(
        'upsert' => true
    ));
    return true;
}

public function destroy($sessionId) {
    $this->_collection->remove(array('session_id' => $sessionId));
    return true;
}

public function gc() {
    $query = array('expired_at' => array('$lt' => time()));
    $this->_collection->remove($query);
    return true;
}

public function __destruct() {
    session_write_close();
}

Listing 7 session.php.

Finally, this class represents a user in the web application. This class can be used to log a user in (authentication), enable him to view pages that he is allowed to see (authorization), and log him out when he wishes. The php class is detailed in listing 8.
<?php
require_once ('dbconnection.php');
require_once ('session.php');

class User {
    const COLLECTION = 'users';
    private $mongo;
    private $collection;
    private $user;

    public function __construct() {
        $this->mongo = DBConnection::instantiate();
        $this->collection = $this->mongo->getCollection(User::COLLECTION);
        if ($this->isLoggedin()) $this->loadData();
    }

    public function isLoggedIn() {
        return isset($SESSION['user_id']);
    }

    public function authenticate($username, $password) {
        $query = array('username' => $username,'password' => md5($password));
        $this->user = $this->collection->findOne($query);
        if (empty($this->user)) return False;
        $SESSION['user_id'] = (string)$this->user['id'];
        return True;
    }

    public function logout() {
        unset($SESSION['user_id']);
        unset($SESSION['user_name']);
    }
}
In the constructor of this class, we obtain a database connection and select the appropriate collection. These objects are stored in private member variables of the class. The `authenticate()` method of the class is used to authenticate a valid user. The method receives the username and password as its arguments. It queries the database with the username and MD5 hash of the password. If a matching document is found, the `ObjectId` of the document is casted to string and stored in `$_SESSION` as `user_id`. The method returns TRUE to indicate that the user is successfully authenticated. Otherwise the method returns FALSE.

The `isLoggedIn()` method checks whether the user is already logged in by simply checking the existence of user id in `$_SESSION`. The `logout()` method terminates the authenticated session by unsetting the user id field. If the user is logged in, the `loadData()` private method is called within the constructor to query the database with the ID and populate the values of user attributes. Finally, the `get()` method is used to read the attributes of a User object.

### 4.8.2 MongoDB: settings and preferences

This sub-module, as the one above, is composed of two parts: a MongoDB database for storing users' settings and preferences; and the PHP server-side technology to access the data. The MongoDB collection for this purpose (named configurations) can store as many settings documents as users registered in the system, but only one document per user.

That module contains three main php classes:
- `mongo load.php`, detailed in section [4.8.2.1].
- `mongo save.php`, detailed in section [4.8.2.2].
- `mongo delete.php`, detailed in section [4.8.2.3].

```php
<?php
    // connect
    $m = new MongoClient();
    // select Sefarad Database
    $db = $m->sefarad;
    // select Configuration collection
    $collection = $db->configuration;
    // search saved configuration
    $query = array('name' => 'saved_configuration',
                   'user_id' => $_SESSION['user_id']);
    $cursor = $collection->find($query);

    // load configuration (saved or default)
    if (($cursor->count()) > 0) {
        foreach ($cursor as $doc) {
            echo json_encode((($doc));
        }
    } else {
        $query = array('name' => 'default_configuration');
        $cursor = $collection->find($query);

        if (($cursor->count()) > 0) {
            foreach ($cursor as $doc) {
                echo json_encode((($doc));
            }
        } else {
            trigger_error("No configuration found", E_USER_ERROR);
        }
    }
?>
```

Listing 9 `mongo load.php`.

First, the script establishes connection to 'sefarad' database and then, it selects the 'configuration' collection. After the connection, we query the collection is queried searching a saved configuration for the current logged in user. That query includes the user id. If any saved configuration is found, it is returned in JSON format. Otherwise, the script returns the default configuration to initialize Sefarad.
<?php

require ('../auth/session.php');

if (isset($_SESSION['user_id'])) {
    $_ac = $_REQUEST['actual_configuration'];

    // connect to Mongo
    $m = new MongoClient();
    // select Sefarad Database
    $db = $m->sefarad;
    // select Configuration collection
    $collection = $db->configuration;

    // delete old saved configuration
    $collection->remove(array(
        'name' => 'saved_configuration',
        'user_id' => $_SESSION['user_id']
    ));

    // save new configuration
    $document = json_decode($_ac, true);

    unset($document['_id']);

    $document['user_id'] = $_SESSION['user_id'];

    $collection->insert($document);
}

Listing 10 mongo save.php.

In the case of saving configuration, after connecting to the database in the same way as in the case above, the script deletes the old saved configuration for the current user (if any). After that, the script inserts the new configuration in a new document into the configuration collection.
Finally, we develop the mongo delete.php class. This script will be called when the user wants to reset his configuration. The main purpose of this script is deleting the saved configuration for the current user, so it simply connects to MongoDB and deletes this document by querying it with the user id.

```php
<?php

require ('../auth/session.php');

if (isset($_SESSION['user_id'])){

    // connect to Mongo
    $m = new MongoClient();

    // select Sefarad DataBase
    $db = $m->sefarad;

    // select Configuration collection
    $collection = $db->configuration;

    // delete old saved configuration
    $collection->remove(array(‘name’ => ‘saved_configuration’, ‘user_id’ => $_SESSION[‘user_id’]));
}
?>
```

**Listing 11 mango delete.php.**