Administration and Notification Service

Deliverable D4.3: Public

Keywords: W3C, OGC, RDF, OWL, Pub-Sub, Notification

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

Executive Summary ........................................................................................................ 7

1 Introduction .................................................................................................................. 8  
   1.1 Document goals .................................................................................................. 8  
   1.2 Document structure ......................................................................................... 8  

2 Semantic Front-end Facilities .................................................................................. 9  
   2.1 Administration and Notification Service goals ................................................... 9  
   2.2 Relation to other front-end tools ...................................................................... 9  
      2.2.1 Distributed Semantic Indexing Infrastructure ............................................. 9  
      2.2.2 Visualization framework .......................................................................... 10  
   2.3 Relation to other WPs ...................................................................................... 10  
      2.3.1 WP2 - Requirements and Architecture ....................................................... 10  
      2.3.2 WP3 - Data modelling and Linked Open Data (LOD) alignment ................ 11  
      2.3.3 WP4 - Demonstration Pilots ....................................................................... 11  

3 Technology review ................................................................................................... 13  
   3.1 Related to Big Data ............................................................................................. 13  
      3.1.1 CouchDB .................................................................................................... 13  
      3.1.2 MongoDB ................................................................................................... 14  
   3.2 Related to Web technologies .............................................................................. 15  
      3.2.1 HTML5 ....................................................................................................... 15  
      3.2.2 Bootstrap .................................................................................................... 16  
      3.2.3 Node.js ........................................................................................................ 17  
      3.2.4 Express ....................................................................................................... 18  
   3.3 Related to Communication models .................................................................... 19  
      3.3.1 Pub/Sub model ............................................................................................ 19  
      3.3.2 MQTT .......................................................................................................... 20  
      3.3.3 AMQP .......................................................................................................... 21  
      3.3.4 Mosquitto ..................................................................................................... 21  
      3.3.5 REST ............................................................................................................ 22  
   3.4 Related to access control .................................................................................. 23  
      3.4.1 Passport ....................................................................................................... 23  
      3.4.2 CKAN .......................................................................................................... 23  
      3.4.3 Socrata ........................................................................................................ 24  
      3.4.4 GeoKnow ...................................................................................................... 25
3.5 Conclusions............................................................................................................... 25

4 Administration and Notification Service ................................................................. 27

4.1 Requirements ......................................................................................................... 27
  4.1.1 Requirements imposed by the project nature..................................................... 27
  4.1.2 Requirements imposed by pilots ..................................................................... 29
  4.1.3 Requirements imposed by technology ............................................................. 30

4.2 Functional design and general architecture ......................................................... 33
  4.2.1 Introduction ...................................................................................................... 33
  4.2.2 Functional design ............................................................................................ 33
  4.2.3 Architecture ..................................................................................................... 34

4.3 Notification Provider ............................................................................................. 38
  4.3.1 Express framework .......................................................................................... 38
  4.3.2 User management ............................................................................................ 39
  4.3.3 Pub-Sub data handler ...................................................................................... 40
  4.3.4 Query Administrator ....................................................................................... 41
  4.3.5 Real time events .............................................................................................. 44

4.4 DB operator ............................................................................................................ 46
  4.4.1 Query management ......................................................................................... 46
  4.4.2 Query Executor ............................................................................................... 47

4.5 Usage and user interface ....................................................................................... 48

5 Pilot exploitation ........................................................................................................ 54

5.1 Agroforestry Management (Spain & Portugal) ..................................................... 54
  5.1.1 Overview ......................................................................................................... 54
  5.1.2 ANS usage ....................................................................................................... 56
  5.1.3 Evaluation ....................................................................................................... 57

5.2 Environmental Data reuse .................................................................................... 59
  5.2.1 Overview ......................................................................................................... 59
  5.2.2 ANS usage ....................................................................................................... 59
  5.2.3 Evaluation ....................................................................................................... 60

6 Conclusions............................................................................................................... 62

References..................................................................................................................... 64
List of Figures

Figure 1 Class diagram of MongoDB data model ........................................................................ 14
Figure 2 Bootstrap design example .......................................................................................... 17
Figure 3 File and folder structure in the server ........................................................................ 18
Figure 4 Characteristics of a Publish/Subscribe-based system ................................................ 20
Figure 5 The publish/subscribe communication model ................................................................ 21
Figure 6 Mosquitto MQTT broker – Websocket support .......................................................... 22
Figure 7 Passport implementation of the login screen .............................................................. 23
Figure 8 Passport implementation of the login screen .............................................................. 24
Figure 9 Data transformation to RDF ....................................................................................... 31
Figure 10 Functional description of relative systems .............................................................. 34
Figure 11 System distribution with pub/sub mesh ..................................................................... 35
Figure 12 Information paths between pub/sub systems ............................................................ 36
Figure 13 Diagram of notifications & administration system modules ...................................... 37
Figure 14 Diagram of DB operator modules ............................................................................ 38
Figure 15 Data flow adding a new notification query .............................................................. 43
Figure 16 Login page ................................................................................................................ 48
Figure 17 Home page – User logged ........................................................................................ 49
Figure 18 Navigation bar .......................................................................................................... 49
Figure 19 New query management page ................................................................................. 50
Figure 20 Status of added notifications ................................................................................... 51
Figure 21 Un-seen changes ...................................................................................................... 51
Figure 22 Results display .......................................................................................................... 52
Figure 23 Edit profile page ....................................................................................................... 53
Figure 24 Users administration ............................................................................................... 53
Figure 25 Single user administration ....................................................................................... 53
Figure 26 Geographic information retrieval and recommendation result generation .............. 55
Figure 27 User administration panel and visualization of associated notifications .................. 56
Figure 28 Process for notification of changes in requested data .............................................. 56
Figure 29 Location of the Macera and Allariz area ................................................................. 57
Figure 30 Elements considered in the application scenario .................................................... 58
List of Tables

No table of figures entries found.
Contractual Date of Delivery to the EC: May 2015

Actual Date of Delivery to the EC:

Editor(s): Ramón Alcarria, UPM

Contributor(s): Ramón Alcarria, Diego Sánchez, Tomás Robles (UPM), John O’Flaherty (MAC), Giovanni Tummarello (Sindice), Karel Charvat (HSRS), Dumitru Roman (SINTEF), Pēteris Brūns (IMCS), Martin Tuchyna (SAŽP), Azucena Sierra (TRAGSA)

DOCUMENT HISTORY

<table>
<thead>
<tr>
<th>Version</th>
<th>Version date</th>
<th>Responsible</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>15 October 2014</td>
<td>UPM</td>
<td>Initial draft, TOC, call for contributions from others.</td>
</tr>
<tr>
<td>0.02</td>
<td>26 January 2015</td>
<td>UPM</td>
<td>First version</td>
</tr>
<tr>
<td>0.03</td>
<td>04 February 2015</td>
<td>UPM</td>
<td>Document revision</td>
</tr>
<tr>
<td>0.04</td>
<td>23 February 2015</td>
<td>UPM</td>
<td>Released version, minor corrections still required</td>
</tr>
<tr>
<td>FINAL</td>
<td>9 March 2015</td>
<td>UPM</td>
<td>Merged with contributions from John O’Flaherty and Pēteris Brūns</td>
</tr>
<tr>
<td>FINAL_REV</td>
<td>21 April 2015</td>
<td>SAŽP-UPM</td>
<td>Integrated version with SAŽP revision</td>
</tr>
<tr>
<td>1.0</td>
<td>25 May 2015</td>
<td>TRAGSA</td>
<td>Final revision and minor corrections</td>
</tr>
</tbody>
</table>

The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Communities. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

Copyright © 2014, SmartOpenData Consortium.
Executive Summary

This deliverable defines the functional architecture, APIs and modules of the Administration and Notification Services. It includes a communication layer composed by several functional components such as: communication brokers, proxies and gateways. It also describes the Administration and Notification service developed, including the description of the implemented APIs, data format and functional components. Finally it summarises test and experimental evaluations.
1 Introduction

This deliverable defines the functional architecture, APIs and modules of the SmartOpenData Administrative and Notification Services. It includes a communication layer composed by several functional components such as: communication brokers, proxies and gateways. It also describes the Administration and Notification service developed, including the description of the implemented APIs, data format and functional components. Finally it summarises test and experimental evaluations.

1.1 Document goals

The aim of this document is to describe the definition, design and implementation of the Administration and Notification Service and to provide an analysis of how this tool can be used in each SmartOpenData pilot.

1.2 Document structure

This deliverable is structured as follows:

Section 2 provides an introduction to the Semantic Front-End facilities, from the point of view of the Administration and Notification service. Other tools to be developed in the scope of the Semantic Front-End facilities are the Distributed Semantic indexing Infrastructure and the Visualization Framework. Relations between the Administration and Notification Service and these two tools are explained. There is also described the relation between these tools and the systems and solutions provided by the other work packages, WP2 (Requirements and architecture), WP3 (Data modelling and Linked Open Data (LOD) alignment) and WP5 (Demonstration Pilots).

Section 3 presents a study of the state of the art in technologies to carry out the development, implementation and testing of this tool. It considers technologies related to Big Data, Web technologies, Communication models, and solutions in related work regarding access control.

Section 4 describes the administration and notification service. This section includes the enumeration of requirements that this service must fulfil in order to reach the goals of the SmartOpenData (SmOD) project. Requirements are imposed by the project nature, the defined pilots, and also by technology. This section also presents a general architecture and the functional design of the different component integration and also the detailed component definition, with the relation between components through defined APIs.

This service must be exploited in various SmartOpenData pilots. Section 5 analyses how pilots find this service useful and explains the process to integrate this service in their service catalogue. We particularly review the two pilots that expressed interest in the current version of the application so far: Agroforestry Management and Environment Data reuse.

Finally, some conclusions are provided in Section 6.
2 Semantic Front-end Facilities

This section provides an introduction to the semantic front-end facilities, described in SmOD. Firstly, the objectives pursued with the development of the tools are shown and, after that, descriptions of the relations between the Administration and Notification service and other activities included in the project, such as other Front-end tools (Distributed Semantic Indexing Infrastructure and Visualization framework) and also the rest of work packages, considering that the Administration and Notification Service is closely related to WP2 (Requirements and architecture), WP3 (Data modelling and Linked Open Data (LOD) alignment) and WP5 (Demonstration Pilots).

2.1 Administration and Notification Service goals

The Administration and Notification Service inherits that objectives imposed by the tools composing the Semantic Front-end Facility. These objectives are:

- Provide common facilities for exploiting environmental data. This objective refers to the fact that many solutions to the exploitation of environmental data are not interoperable and changing the data source these solutions stop working. By providing a single data processing environment the companies holding such data do not need to seek services outside of the platform for obtaining a personalized data processing.

- Improve environmental data searchability thanks to integration of big data infrastructure for structured and semi-structured search facilities. This requirement is more related to the distributed semantic indexing infrastructure. By solving this goal we will have a set of search tools that allow us to find information in distributed repositories belonging to different domains and using the Linked Open Data, which allows us to structure and link information from the semantic point of view.

- Provide a semantic front-end framework for environmental data visualization: This goal, more related to the task of providing a visualization framework, allows us to offer to the considered roles (data providers, data analyzers, end-users) a search and information representation environment tailored to the characteristics of this information. To do that, it will be implemented as a search engine based on facets and a spatial visualization client based on maps and overlays.

2.2 Relation to other front-end tools

2.2.1 Distributed Semantic Indexing Infrastructure

The purpose of this tool is to integrate semantic indexing and ETL (extract, transform, load) platform.

Sindice will provide ETL workflows for continuous transformation of the environmental data produced by the partners according to the vocabulary defined in WP3. The transformed data
will be indexed by Sindice and made available through public or private cloud spaces, according to the defined exploitation plan. In addition, search capabilities based on SIREn will be provided so that environmental data can be searched in natural language.

This task will further investigate the creation of a Software-as-a-Service platform for environmental open data together with supporting activities for effective and efficient publication and consumption of data and creation of applications using the data. This task will produce D4.1.

### 2.2.2 Visualization framework

The purpose of this tool is to support demonstrators in the development of user interfaces for consulting and visualising geospatial data exposed as Linked Open Data resources.

To contribute to solve problems related to spatial data visualization, a semantic visualization tool named SEFARAD\(^1\) 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 a utility for creating and processing human-readable SPARQL queries which can be used for representing linked data information. This task will be led by UPM and produce D4.2.

### 2.3 Relation to other WPs

#### 2.3.1 WP2 - Requirements and Architecture

WP2 defines requirements and the architecture of the SmOD platform. The main focus of the project is to build a Linked Open Data infrastructure fed by public data resources, existing sources for biodiversity and environment protection in European protected areas that satisfies the requirements of four kinds of target users identified: (i) public bodies, (ii) researchers (iii), companies and (iv) citizens.

A detailed collection of requirements and user cases for the SmOD infrastructure were generated, aiming at making available a valuable quantity of open data resources through Linked Open Data (LOD) and semantic approach until now unavailable by easing its fruition and integration. The requirements contributed to defining the overall technical architecture of the project, specifying the interfaces between modules in order to provide a modular solution, which can be configured to provide more or less functionalities depending on user needs and the commercial strategy, as well as integrating future modules.

The defined architecture of SmOD is a high-level technical specification including the main components and connection points to other tools and systems. It defines platform neutral components and provides suggestions for concrete implementation. The architecture of SmOD is described in deliverable D2.3.

The work described in this deliverable is related to the work done in WP2 as we take into account the following information:

\(^1\)https://github.com/gsi-upm/Sefarad
- Detailed collection of requirements for the SmartOpenData infrastructure, including LOD interoperability, multilingual and semantic requirements.
- Formal description of users’ requirements and use cases, taking into account the four kinds of target users detected.
- Reference architecture model and high-level technical specification for the SmartOpenData infrastructure.

2.3.2 WP3 - Data modelling and Linked Open Data (LOD) alignment

The work package aims to research and define the SmartOpenData data model at query, data and validation scope. This task takes into account the infrastructure requirements, including interoperability and multilingualism.

The initial version of the data model was provided and it is being adapted to different language schemas.

The work described in this deliverable is based on the work done in WP3 as we take into account the following information:

- Alignment between geographic and environmental information.
- Harmonization of open data resources to the SmartOpenData data model.
- Definition and validation of the data transformation process to turn unstructured data into RDF triples.

2.3.3 WP4 - Demonstration Pilots

This WP evaluates SmartOpenData infrastructure and tools by the development and deployment of advanced demonstrators. The WP focuses on evaluating the effectiveness of the approach for the Open Linked Data and semantic services as well as how the proposed architecture can be adapted to different scenarios, evaluating the limitations and benefits of the approach by comparison with existing technologies.

The pilot applications will be developed by pilot partners with support of technological partners. There are planned 5 pilots:

- Agroforestry management pilot leader Tragsa (Spain) partners DGI
- Environmental research, Biodiversity pilot leader MAC (Ireland)
- Water monitoring pilot leader ARPA (Italy)
- Forest sustainability pilot leader FMI (Czech republic)
- Environmental data reuse SAZP (Slovakia)

Currently demonstrators are being defined and requirements are established. Pilots need applications from the Semantic Front-end facilities to deliver a first iteration of the demonstrator, which will provide early prototypes for its evaluation and assessment, in order to provide feedback to other WPs, such as WP2 – WP4. The first iteration of pilots will
first be tested by partners for the enhancement and completeness of the integrated architecture, and bug fixing. Then evaluation and assessment will be carried out in-depth by end-users to get their user experiences and feedback to iteratively improve the project.
3 Technology review

This section presents a study of the state of the art in technologies to carry out the development, implementation and testing of this tool.

3.1 Related to Big Data

‘Big Data’ is the term applied to large quantities of data (often hundreds of terabytes or petabytes in scale) that are beyond the ability of most existing software and technology to efficiently store, manage and analyze. With the massive boom in popularity of social media for both social interaction and business transactions there is a big move to exploit this data source for business and intelligence advantage.

However the vast infrastructure requirements and data/content analytics made possible by Big Data, pose big security implications for users (see General issues and challenges below).

New tools, many of them open source, are emerging to handle and analyze these massive amounts of data. The current trend is to distribute and process the data across clusters of huge numbers of low spec servers to obtain cost effective scalable architecture. The data is distributed as a file system (e.g. Google File System, Hadoop Distributed File System (HDFS)) or as a distributed Structured Query Language (SQL) - or ‘No SQL’- database (e.g. CouchDB, MongoDB). The analysis is then mapped across the data using ‘Advanced SQL’ or Hadoop/MapReduce.

3.1.1 CouchDB

Apache CouchDB is a document-oriented NoSQL database that uses JSON to store data, JavaScript as its query language using MapReduce, and HTTP for an API. One of its distinguishing features is multi-master replication. CouchDB was first released in 2005 and later became an Apache project in 2008.

CouchDB implements a form of Multi-Version Concurrency Control (MVCC) in order to avoid the need to lock the database file during writes. Conflicts are left to the application to resolve. Resolving a conflict generally involves first merging data into one of the documents, then deleting the stale one.

CouchDB includes a number of other open source projects as part of its default package:

- SpiderMonkey: a code name for the first ever JavaScript engine maintained by the Mozilla Foundation.
- jQuery: a lightweight cross-browser JavaScript library that emphasizes interaction between JavaScript and HTML

---

2Apache CouchDB. [http://couchdb.apache.org](http://couchdb.apache.org)
3MongoDB. [http://www.mongodb.org](http://www.mongodb.org)
- International Components for Unicode (ICU): an open source project of mature C/C++ and Java libraries for Unicode support, software internationalization and software globalization.

- OpenSSL: an open source implementation of the SSL and TLS protocols. The core library (written in the C programming language) implements the basic cryptographic functions and provides various utility functions.

- Erlang: a general-purpose concurrent programming language and runtime system. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing.

After the study of CouchDB and other similar tools such as MongoDB we found more suitable the SQL-like querying syntax provided by MongoDB, instead of MapReduce syntax. This decision is based on the provided experience using these two tools, as Administration and Notification service developers are more used to the SQL-like querying syntax.

MongoDB is also a document oriented store like CouchDB and it also support ad hoc querying, which is probably one of the crucial features why people searching for DRBMS substitution choose MongoDB over the other NoSQL solutions.

### 3.1.2 MongoDB

MongoDB is an open-source NoSQL-Database developed by 10gen\(^4\) in C++. NoSQL is a new trend in database development and refers generally to databases without fixed schema. Such databases usually have a lower transaction safety but are faster in accessing data and scale better than relational databases. MongoDB is one of the newer NoSQL databases developed in 2009. The database belongs to the category of the document-based databases.

MongoDB works with dynamic schemas, every collection can contain different types of objects. Every object – also called document – is represented as a JSON structure: a list of key-value pairs. The value can be of three types: a primitive value, an array of documents or again a list of key-value-pairs (document).

![Figure 1Class diagram of MongoDB data model](http://www.10gen.com)
To query these objects, the client can set filters on the collections expressed as a list of key-value pairs. It is even possible to query nested fields. The queries are also JSON structured; hence a complex query can take much more space than the same query for a relational database in SQL syntax. If the built-in queries are too limited, it is possible to send JavaScript logic to the server for much more complex queries. MongoDB requires always using the correct type: If you insert an integer value into a document, you have to query for it also with an integer value. Using its string representation does not yield the same result.

The dynamic query support and the optimized querying speed is the main reasons to choose this technology over other very similar No-SQL solutions such as CouchDB.

We use MongoDB to save all information that the system uses in its operations. All entries will be used in the generation of notifications as well as various system configuration parameters will be stored in different collections.

Since our system uses two main separate modules, each of them will use a different MongoDB instance, but it can be integrated into a single service by setting different ports for each instance. Schemas used for the different collections will be explained in chapter 4.

3.2 Related to Web technologies

Web technologies are used for the implementation of the Administration and Notification service. In this section we review some programming languages and framework that were used for this task.

3.2.1 HTML5

HTML5 is the fifth revision of the HTML standard. Beside changing and enriching old ones, it provides plenty of new elements. This includes new media elements like `<audio>` and `<video>`, new structural elements like `<section>` and `<article>`. HTML5 specifies not only markup but also APIs which can be used with JavaScript. Many new APIs are introduced such as:

- Canvas element, for 2D drawing
- Offline applications for interacting with application event when network connection is not available
- Native drag and drop - no need to write complex JavaScript code to implement drag and drop functionality - it will be supported natively
- Web storage - key-value storage framework similar to Cookies, but with larger storage capacity and improved API
- Databases API - to manipulate client-side databases using SQL

Some related technologies are not included in the W3C HTML5 specification, however they can be used together with HTML5. W3C publishes specification for them separately. Here is some of them:

- Geolocation - provides access to geographical location information associated with a client
- Web workers - give ability to run background scripts independently from UI scripts. This will help to keep web pages responsive and execute long running scripts simultaneously
- WebSockets - to enable bidirectional communication between browser and webserver

The HTML5 specification was released on October 2014 and it is partially supported by many browsers. A very large problem in web development is the incompatibility between the browser implementations. These incompatibilities are in various areas. Most of the functionality of the web is standardized, but not all of the functionality. Also not all standards are implemented or are implemented correctly. This makes writing code that runs well in different browsers difficult.

Libraries have been made to deal with the incompatibilities between the different browser implementations. Well-known examples are jQuery and Prototype. Libraries like these make an abstraction over the browser interface to deal with the incompatibilities.

We used HTML5 web technologies and jQuery/Prototype libraries to deal with browser incompatibilities. Because of HTML5, it is often no longer needed to use plug-ins to build Rich Internet Applications. The compatibility between the browsers is improved, because no plugins have to be used.

### 3.2.2 Bootstrap

Bootstrap is a free collection of tools for creating websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions.

It contains the following features:

- Compatible with the latest versions of all major browsers. It gracefully degrades when used on older browsers such as Internet Explorer 8.
- Since version 2.0 it also supports responsive web design. This means the layout of web pages adjusts dynamically, taking into account the characteristics of the device used (desktop, tablet, mobile phone).
- Starting with version 3.0, Bootstrap adopted a mobile-first design philosophy, emphasizing responsive design by default.
- Bootstrap is open source and available on GitHub. Developers are encouraged to participate in the project and make their own contributions to the platform.
- Strong community support: Recently, community members have translated Bootstrap's documentation into various languages, including Chinese, Spanish and Russian

---

5 How well does your browser support html5? [https://html5test.com](https://html5test.com)

We chose Bootstrap over other frameworks such as Foundation because it supports a large selection of themes and also is compatible with previous versions of Internet Explorer, such as IE8.

3.2.3 Node.js

Node.js is an open source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript. Node.js provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. These technologies are commonly used for real-time web applications.

Node.js uses the Google V8 JavaScript engine to execute code, and a large percentage of the basic modules are written in JavaScript. Node.js contains a built-in library to allow applications to act as a Web server without software such as Apache HTTP Server or IIS.

Node.js is gaining adoption as a server-side platform and is used by Groupon, SAP, LinkedIn, Microsoft, Yahoo!, Walmart, Rakuten and PayPal.

We chose Node.js over other backend technologies based on our Javascript programming experience, as it is the most mature framework for developing applications in Javascript.

---

7Industry Usage, Node.js Website: http://nodejs.org/industry/
3.2.4 Express

Express.js is a Node.js web application framework, designed for building single-page, multi-page, and hybrid web applications. Express 3.x helps organizing web application into MVC architecture on the server side. It is possible to use a variety of choices in the templating language (like EJS, Jade, and Dust.js).

It provides an easy way to connect to a database such as MongoDB with Mongoose connector (for modeling) to provide a backend for your Node.js application. Express.js basically helps managing everything, from routes, to handling requests and views.

Below we can find the basic folder structure generated by this framework.

```
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   │   ├── javascripts
│   │       └── style.css
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── user.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

**Figure 3** File and folder structure in the server

The functionality of each file and folder is described:

- **apps.js** - is the main application and also the file that will be executed when the server starts. It contains the necessary information to initiate and open a port to listen from requests of possible clients.

- **bin** - a folder containing references to external scripts

- **package.json** – a folder containing a *json* where application dependencies are defined as well as other basic information about the application.

- **public** – a folder containing all the files to be accessible from internet, and also common to all web pages. It contains some other folders such as images, *javascripts* and *stylesheets*. In this last folder some stylesheets are defined to define the style and visual aspect of the application.

- **routes** – is a folder that includes the code to be executed when invoked the corresponding views.
- **view** – is a folder that includes the views to generate when some pages are invoked. There are two main ways to define views and express framework supports both, providing that the adequate modules are imported. On the one hand Jade defines the html elements programmatically, on the other hand *Embedded Javascript* or *ejs* contains *html* and enables to embed code in the own file. In both cases logic is processed in the server and the client only receives an *html*.

There are other Node.js frameworks that we revised, such as Koa\(^8\) and Hapi\(^9\), but Express.js is the most popular Node.js web application framework used today, it has the biggest community, with almost 5 years of development behind it. It offers a simple way to get a server up and running and promotes code reuse with its built in router.

### 3.3 Related to Communication models

#### 3.3.1 Pub/Sub model

According to Tarkoma et al. [1], “*The Publish/Subscribe technology encompasses a wide number of solutions that aim at solving a vital problem pertaining to timely information dissemination an event delivery from publishers to subscribers*”. Therefore, in communication networks a Publish/Subscribe (Pub/Sub) service, also called event dissemination service, enables the communication components to dynamically detect, isolate, manage and push tailored interests and the corresponding events, to particular participants of the network.

According to Eugster et al. [2], a Publish/Subscribe system is basically composed of three main components **publishers**, **subscribers** and **brokers**. **Publishers** produce information in the form of asynchronous events, which are later consumed by subscribers. **Subscribers** that prior to receiving these events, express their willingness to consume the information in the form of subscriptions to interests; and finally **brokers** that put in contact publishers and subscribers by storing subscriptions, matching events, and finally notifying the tailored information according to the interests.

Publishers and Subscribers remain anonymous to each other, since they do not have to be aware of any state of each other. It means that publishers and brokers are fully decoupled in time, space and synchronization. Brokers are the core components of the Publish/Subscribe system as they implement all of the event routing, subscription storing, matching and notification tasks needed to link other brokers in order to a build Publish/Subscribe network. A Publish/Subscribe network can be built following a centralized\[^3\], semi-centralized\[^4\]\[^5\] and distributed\[^6\] approach. Despite the architectural differences, the capabilities of a Publish/Subscribe system always depend on a well-known set of characteristics:

- The subscription model
- The subscription management and matching
- The event notification

---

\(^8\)Koa Homepage: http://koajs.com/

\(^9\)Hapi Homepage: http://hapijs.com/
The event routing or event dissemination

The following Figure depicts the characteristics of a Publish/Subscribe-based system.

![Figure 4 Characteristics of a Publish/Subscribe-based system](image)

The subscription model (also called subscription language) of a Publish/Subscribe system specifies the expressiveness of the subscriber’s interests and determines the complexity of the matching of content that fulfill these interests. Therefore, event matching consists of filtering the events (that enclose the content) received from publishers; events that later must be correctly and precisely notified to clients; so, the subscription model conditions the capability of a broker to match events and the realization of subscribers’ interests of information.

For the design and development of the Administration and Notification service we use the Publish/Subscribe model in order to let clients to be subscribed to changes in SmartOpenData databases so that they can receive up-to-date notifications.

We selected a topic based subscription model. In this model, the interests of subscribers are expressed as topics, which can be hierarchically organized. Today, standards such as WS-Topics\(^{10}\), and lightweight protocols such as MQTT use the concept of topics in order to identify specific resources. According to\([2]\) and \([7]\), the advantage of a topic based subscription model is its simplicity; so it leads to lightweight and very efficient implementations that can meet the requirements of mobile Publish/Subscribe networks.

### 3.3.2 MQTT

**Message Queue Telemetry Transport**\(^{11}\) is a Publish/Subscribe, simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency and unreliable networks. The design principles are to minimize network bandwidth and device resource requirements while attempting to ensure reliability and some degree of assurance of delivery.

MQTT follows the publish/subscribe communication model (Figure 5) and needs a client and broker implementations. There are several MQTT brokers available\(^{12}\). They vary in their feature set and some of them implement additional features on top of the standard MQTT functionality.

---

\(^{10}\) WS-Topics. Retrieved from [http://docs.oasis-open.org/wnsn/wnsn-ws_topics-1.3-spec-os.pdf](http://docs.oasis-open.org/wnsn/wnsn-ws_topics-1.3-spec-os.pdf)


\(^{12}\) MQTT-supported server comparison: [https://github.com/mqtt/mqtt.github.io/wiki/server-support](https://github.com/mqtt/mqtt.github.io/wiki/server-support)
3.3.3 AMQP

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardized but extensible "messaging capabilities".

We have studied the AMPQP specification and we found out that, although AMQP provides a richer built-in functionality it is also heavier. We wanted to choose a pub/sub solution with small footprint that provided some quality of service for bandwidth constrained or congested environments. For that reason we selected the MQTT approach.

3.3.4 Mosquitto

Mosquitto is an open source (BSD licensed) message broker that implements the MQTT Telemetry Transport protocol versions 3.1 and 3.1.1. MQTT provides a lightweight method of

\[\text{MQTT-S}^{13}\] is an extension for the MQTT protocol that is designed to operate on low-cost and low power sensors devices. In our case it is not necessary to use MQTT-S, as data to be transmitted is geographical data requiring medium bandwidth and high-powered devices.

Figure 5 The publish/subscribe communication model

\[\text{MQTT-S}.\text{Retrieved from http://mqtt.org/MQTT-S_spec_v1.2.pdf}\]
carrying out messaging using a publish/subscribe model. This makes it suitable for "machine to machine" messaging such as with low power sensors or mobile devices such as phones, embedded computers or microcontrollers like the Arduino.

Websocket support is incorporated into the last version of Mosquitto. Instead of writing an own Web application to translate MQTT messages to a Web-based language, now MQTT websocket libraries (mqtt.io or the HiveMQ Websocketclients) can be configured to use the host name and TCP port number (from the listener directive) of the Mosquitto broker.

![Mosquitto MQTT broker – Websocket support](image)

3.3.5 REST

REST stands for Representational State Transfer. It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used.

REST is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture.

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, et al.). Later, we will see how much more simple REST is.

Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture.

For example there are some URIs that identify individual items or collections:

- `http://example.com/products/4554`

We use REST principles in the Administration and Notification Service to provide the responsive interfaces of adding, editing and to delete query notifications. As long as the user uses the web application, REST methods are invoked to perform operations in the server side of the system.
3.4 Related to access control

Related to the access control functionality we have reviewed some technologies to provide authentication and authorization mechanisms to our solution. We also reviewed how access control is implemented in similar projects: CKAN, Socrata and GeoKnow.

3.4.1 Passport

Passport is authentication middleware for Node.js. It is designed to serve a singular purpose: authenticate requests. When writing modules, encapsulation is a virtue, so Passport delegates all other functionality to the application. This separation of concerns keeps code clean and maintainable, and makes Passport extremely easy to integrate into an application.

In modern web applications, authentication can take a variety of forms. Traditionally, users log in by providing a username and password. With the rise of social networking, single sign-on using an OAuth provider such as Facebook or Twitter has become a popular authentication method. Services that expose an API often require token-based credentials to protect access.

Passport recognizes that each application has unique authentication requirements. Authentication mechanisms, known as strategies, are packaged as individual modules. Applications can choose which strategies to employ, without creating unnecessary dependencies.

In our implementation we specifically use passport in the login screen, to check the user’s credentials against a MongoDB database. Our app will direct the user to a home page if the log in succeeds. On the home page, we’ll be using socket.io to connect to our web server and maintain a list of current users.

Since passport knows where to redirect if a login fails or succeeds, we have to define what to return for each of those paths in our index.js file. We’ll direct the user back to our login page if it fails. If it succeeds, we’ll direct the user to a page which will show all the users currently online.

```javascript
router.get('/loginFailure', function(req, res, next) {
  res.render('login');
});

router.get('/loginSuccess', function(req, res, next) {
  res.redirect('home');
});
```

Figure 7 Passport implementation of the login screen

3.4.2 CKAN

CKAN\(^{14}\) is a leading open-source data portal platform and powerful data management system that makes data accessible – by providing tools to streamline publishing, sharing, finding and using data.

\(^{14}\)CKAN Web page: [http://ckan.org](http://ckan.org)
CKAN is built with Python on the backend and Javascript on the frontend, and uses The Pylons web framework and SQLAlchemy as its ORM. Its database engine is PostgreSQL and its search is powered by SOLR. It has a modular architecture that allows extensions to be developed to provide additional features such as harvesting or data upload. CKAN uses its internal model to store metadata about the different records, and presents it on a web interface that allows users to browse and search this metadata. It also offers a powerful API that allows third-party applications and services to be built around it.

Among the features provided by CKAN we study the following ones:

- Fine-grained access control
- Integrated data storage and full data API
- Administration of all edits and versions of dataset metadata

From the administration’s point of view we base on the CKAN concept of providing complete history of changes to a dataset, and compare different revisions. Activity stream messages show the reason for successive changes in the history of a dataset.

3.4.3 Socrata

Socrata is a Seattle-based cloud software company, focused exclusively on democratizing access to government data. They help public sector organizations improve transparency, citizen service and fact-based decision-making by efficiently delivering data to citizens, employees and developers in a user-friendly experience on web, mobile and machine-to-machine interfaces.

Socrata enables:

15 Socrata Web page: [www.socrata.com/](http://www.socrata.com/)
- Measuring the success of an User’s Initiative in Real-time - Find out what’s working and what’s not based on user interactions, in addition to real-time consumption and distribution of your data and APIs.
  - Data Publishers track which data is popular with their constituents, how it is being consumed, and where it is shared on the web.
  - Real-time reporting allows data publishers to monitor poignant (“hot”) datasets, trending keyword searches, and influential traffic sources.
  - Track usage of your APIs and follow which apps are relying on your data.

3.4.4 GeoKnow

GeoKnow\textsuperscript{16} is an EU FP7 research project, motivated by previous work in the LinkedGeoData project (LGD)\textsuperscript{17}, which makes OpenStreetMap (OSM) data available as an RDF knowledge base.

The main feature analysed from GeoKnow is the efficient geospatial RDF querying. Existing RDF stores lack performance and geospatial analysis capabilities compared to geospatially-enabled relational DBMS. GeoKnow focuses on introducing query optimisation techniques for accelerating geospatial querying by at least an order of magnitude.

Another feature that is important from the point of view of the Administration and Notification service is versioning functionality. As resources in RDF data have global identifiers as URIs, this make it possible to manage versions in a much more sophisticated way, as well as querying for changes between one version and the next. So metadata for provenance of geospatial information should be maintained mostly from external sources. However, in case existing geometries get modified, this requires a much more advanced mechanism for the reconciliation of conflicting updates.

3.5 Conclusions

In this section we review the state of the art in technologies, communication models, and related projects to carry out the definition, development and testing of the Administration and Notification service.

From the analysis related to Big Data we determine that it is needed to take into account the processing and storage demand of geospatial data and geospatial analysis. We use \textit{MongoDB} to store users, queries and query results, expressed in RDF and containing spatial information in the form of Well-known Text (WKT) geometries.

From the analysis related to Web Technologies, in which we reviewed some programming languages and framework such as \textit{HTML5} and \textit{Node.js}, we extract that it is beneficial to use HTML5 web technologies and \textit{jQuery/Prototype} libraries to deal with browser incompatibilities. We also use \textit{Express} as it provides an easy way to connect to a database


such as MongoDB with Mongoose connector (for modeling) to provide a backend for your Node.js application.

Related to communication models analysis, we determine that the use of Publish/Subscribe models (and the MQTT protocol) is very adequate to let clients to be subscribed to changes in SmartOpenData databases so that they can receive up-to-date notifications, using also REST principles to communicate the Administration front-end to the Node.js back-end and databases.

Finally, related to access control, we use the authentication middleware for Node.js provided by Passport to implement the login page and profile management, whereas we use the CKAN concept of providing complete history of changes to a dataset, and compare different revisions.

As resources in RDF data have global identifiers as URIs, this allow us to reuse some functionalities from the GeoKnow project and manage versions (in our case query results) in a much more sophisticated way, as well as querying for changes between one version and the next.
4 Administration and Notification Service

4.1 Requirements

4.1.1 Requirements imposed by the project nature

Related to the reuse of existing “INSPIRE based” relevant spatial data sets, services and appropriate metadata:

- Provide a system that understands the relationships between objects and repositories that are involved (OGC and RDF structures). The project will build an infrastructure of objects and relationships with the added value of further links. Allows for flexible integration of datasets from different sources, without needing the data to be moved.

- Foster the reuse of information from reference/authoritative sources.

- Focus on allowing semantic interoperability among geographic and environment Open Data catalogues mentioned above.
  - SuperCat\(^{18}\) now offers seamless access to INSPIRE services.
  - HSCAT\(^{19}\) is harvesting metadata from many European INSPIRE portals, and GEOSS portals.
  - List of OGC services discovered on Google\(^{20}\) & Geocatalogue of OGC resources discovered on Google\(^{21}\)
  - Linked Open Data and the semantic technologies approach will support fluent and flexible interaction among repositories.

- Support and reuse of definitions of spatial objects in major databases (DBPedia and GeoNames)

The notification service must support SPARQL/geoSPARQL queries for objects that are defined in SmartOpenData but linked to the main databases. An example of this link can be seen in the following example, which queries all the cities close to Madrid with a population less than 40000.


\(^{19}\)https://play.google.com/store/apps/details?id=cz.hsrs.hscat&hl=en

\(^{20}\)http://92.52.20.94:8082/ogcwxs/html/

\(^{21}\)http://92.52.20.94:8082/geonetwork/apps/tabsearch/
We can see that in this example, the ID, name, location, and geometry of the city are queried. This distributed query employs different information sources to generate the response. For example, the value of the population of the city is derived from Geonames (gn:population) but, the geometry of this city is obtained from a repository with prefix ex (ex:geometryGML), that could provide a partner of SmartOpenData.

Although the Administration and Notification Service is not properly a visualization service we provide some tools to visualize notification results.

There are two possibilities:

a) To visualise a list of objects in some form;

b) To provide cartographic visualisation, in relation to the original data. It will be necessary to define some mechanisms, like Filter Encoding etc. that could visualise results on the base of queries²².

This leads us to several questions:

- How will this new mechanism influence efficiency and speed of data processing and querying?

- How are metadata captured and processed?

Metadata plays the crucial role as interface to the spatial content it describes. In the context of SmartOpenData, metadata serves as the exchange component allowing to bridge INSPIRE and other spatial worlds.

Managing the data lifecycle is a challenging task. Mechanisms for handling updates and deletions in the data should be devised.

- To enable engagement in entirely new forms of scientific research and to explore correlations between research results.

Linked Open Data stimulates semantic correlation among repositories and data sources beyond organisational boundaries. SmartOpenData will need to cover this correlation in the environmental protection scope, unlocking new ways of research by leveraging the semantic use of scientific data: the huge amount of linked information made available will allow more

²²For instance, similar to the OpenDataSoft Map Tab functionality and Geoclustering API which use scalable and efficient server-side clustering, for easy exploration of large datasets and dealing with the stacking of several points in the same place, as described at http://www.opendatasoft.com/2013/12/06/introducing-new-map-tab-geoclustering-api-2/?goback=%2Egde_3731775_member_5814763452148776961#%21
powerful computation studies, data assessment and connecting public bodies’ researchers, companies and citizens in an improved effort for addressing research topics.

4.1.2 Requirements imposed by pilots

From Deliverable 2.1 “Requirements of the SmartOpenData Infrastructure” we can extract that SmartOpenData will need to be able to test the semantic technologies developed in 5 pilots that will be fully interoperable at data, metadata and semantic levels.

Therefore, one of the functions that must be provided from the Administration and Notification service is to explore research indicators such as public geoportal available covering the 5 interoperable pilots developed into a single access point and including a Governance Model, framework and procedures allowing at least 5 other environment protection initiatives to join SmartOpenData.

In this section we highlight the requirements for pilots and end users. We focus on the Agroforestry Management pilot, which has a well-defined use cases and a design solution so far.

4.1.2.1 Agroforestry Management

The agroforestry management pilot will mainly be focused on meeting the needs and requirements of the public sector regarding forest management and land use planning normative requirements. In both countries, public bodies from these areas are involved.

The pilot has a use case defined in detail in Section 5.1 of this deliverable; however in this section the essentials of this scenario are summarized to extract the requirements to solve the proposed service.

An owner or forester wants to purchase some seeds. For this he needs to know the location of the closest seed nurseries or public seedbeds. Currently, the reservation procedures for seedbed visitation and seed stock consultation are conducted in person, by filling out reports, and registering them through the Spanish public administration. This clearly results in complex and slow processes. The location of public seed nurseries and their availability for seed collection may be obtained on the Internet and updated in real time. Thus, any change in the status or information on any of the public seedbeds would be immediately accessible and available to anyone who needed it.

Another requirement of this pilot is to bring the dataset to the various levels of public administration in a unified way, so that they can be accessible to local, regional and national level, facilitating the exchange of data between national and regional institutions.

In addition to this requirement, it is important to link these data with services that will generate added value information. For example, foresters and landowners usually select species to plant in their fields based on experience or intuition. No information or analytical data are available to define what the most recommended species is. In this regard various value added services are proposed:

- A geographic information system showing a recommendation on the distribution of species according to some variables. These variables can be geology, weather (temperature, precipitation), vegetation, orientation, slope, etc.

- A real-time system that provides an accessible way, availability of seeds in seedbeds.
- A system that allows easy access to studies that analyzes and recommends planting a species depending on the region of origin and quality of seed.

The solution of these requirements would allow better use of forest genetic resources and increase forest production. From the technical point of view the above requirements are translated into the following challenges:

- Discovery: An efficient discovery of data can be achieved by implementing large catalogs of metadata.

- Federation: Provide a publication and catalog maintenance more decentralized and close to the players and owners of the data.

- Interoperability: Compatibility between data catalogs and services, although they have been developed by different vendors. Harmonization of geospatial metadata (based on ISO19115/19119 standards) with the principles of the Semantic Web. In the context of SMOD metadata can be exchanged compatible with INSPIRE to/from other data models.

- Display of linked open data: Publication of information resulting from the process of structuring and binding in accordance with the requirements of the stakeholders. Metadata serve as an entry point for the provision of information necessary for the transformation of spatial data to RDF (Resource Description Framework) structures.

4.1.3 Requirements imposed by technology

4.1.3.1 Large Volumes of Real Time Data

Another area of research for the project is the handling of large volumes of real time data. Activities of partners within the consortium include real time monitoring of environmental conditions that generate data that needs to be added to the dataset using SPARQL Update. This puts a strain on the infrastructure and so methods to reduce that stress need to be researched, perhaps using, through ERCIM, the W3C POWDER Working Group recommended technology as a data compression tool, as currently being tested in the SemaGrow project. Tracking the provenance of any data is important, but as yet there is no (standardised) linkage within the Semantic Web technology stack between Provenance and SPARQL Update.

4.1.3.2 Using INSPIRE

The SmOD project seeks the use of INSPIRE as the basis for the data structures of each pilot. However, the INSPIRE guidelines are defined as XML and GML schemas, i.e., using non-linked technologies. There are some works describing the use of linked data in the context of INSPIRE, particularly framed in the GeoKnow [1] project. The European Commission Jointed Research Centre initiated the activities under the ARE3NA study on RDF and PIDs for INSPIRE focused on the development of the guidelines on methodologies for the creation of RDF. Outcomes of study will be taken into the consideration in development of the final SmartOpenData model (D 3.4).

http://www.w3.org/2007/powder/
www.semagrow.eu
4.1.3.3 Semantic indexing

Semantic indexing requires that the data supplied by the data providers are transformed to RDF so they can be properly linked and used by pilots.

Figure 9 shows the process to be followed to transform some data found in a relational database to RDF. The first step is to transform structured data in tables to structured data using RDF. Besides this transformation process a binding process is required so that relationships between entities are properly set. These two processes produce an RDF graph that contains the semantic relationships between the concepts defined in the database.

For the development of this functionality some tools were studied.

For transformation from RDB to RDF Sparqlify\textsuperscript{26} was analyzed, which rewrites SPARQL queries into SQL and defines views in RDF of relational databases for subsequent query in SPARQL. We also analyzed TripleGeo\textsuperscript{27}, an open source tool for extracting geospatial features in the form of RDF triples.

To establish relations in the field of linked data GeoLift\textsuperscript{28} was analyzed, a framework of spatial mapping implemented in Java for the enrichment of RDF data with spatial information, and LIMES\textsuperscript{29}, used for the large-scale discovery of links from the Web of Data.

Finally, among the tools related to data storage, once data transformed into RDF, we highlight Virtuoso\textsuperscript{30}, a database manager that provides tools for managing data in RDF, besides supporting N3/N-Triples, data serialization and SPARQL requests; Parliament\textsuperscript{31}, a

---

\textsuperscript{26}Sparqlify: http://aksw.org/Projects/Sparqlify
\textsuperscript{27}TripleGeo: https://github.com/GeoKnow/TripleGeo
\textsuperscript{28}GeoLift: http://aksw.org/Projects/GeoLift
\textsuperscript{29}LIMES: http://aksw.org/Projects/LIMES
\textsuperscript{30}Virtuoso: http://virtuoso.openlinksw.com/
\textsuperscript{31}Parliament: http://parliament.semwebcentral.org/
triplestore compatible with RDF, RDFS, OWL, SPARQL and GeoSPARQL standards, and finally, uSeekM\(^{32}\), which is an extension for Sesame to extend semantic databases with indexing and query capabilities.

### 4.1.3.4 SPARQL limitations

As the amount of Linked Open Data on the web increases, so does the amount of data with an inherent spatial context. Without spatial reasoning, however, the value of this spatial context is limited. Over the past decade there have been several vocabularies and query languages that attempt to exploit this knowledge and enable spatial reasoning. These attempts provide varying levels of support for fundamental geospatial concepts. GeoSPARQL, attempts to unify data access for the geospatial Semantic Web.

The GeoSPARQL language defines both a small ontology for representing features and geometries and a number of SPARQL query predicates and functions. GeoSPARQL is intended to inter-operate with both quantitative and qualitative spatial reasoning systems\(^{10}\). SPARQL limitations will be overcome by using GeoSPARQL when required providing that we offer support for these technology in our LinkedData end-points.

### 4.1.3.5 Usage of Uniform Resource Identifiers

The use of LOD characteristics depends on establishing Uniform Resource Identifiers (URIs) for linking resources, which correspond to real objects, definitions and documents. For the semantic indexing function we build URIs for spatial objects to become as much persistent as possible. The pattern defined for spatial objects is of the form:

\[
\text{http://\{domain\}/\{collection\}[/\{so\}][/\{class\}][/\{inspireLocalId\}][/\{inspireVersionId\}]
\]

where [/so] means that the URI identifies an *spatial object*. Although optional, its use is recommended, as it differentiates immediately whether the object is a spatial object [/so], a place in the real world [/id], or a document describing an element of the real world [/doc]. This entity is of a type defined by the class [/\{class\}]. {inspireLocalId} is an optional parameter and identifies a location according to the Inspire model.

For each application environment, that is, each pilot within the SmOD project, this data structure will have to be extended to include domain specific vocabularies and will also require the definitions of new concepts and relationships. To do this, we recommend the document of best practices published by the W3C\(^{33}\).

---

32 uSeekM: https://dev.opensahara.com/projects/useekm
4.2 Functional design and general architecture

4.2.1 Introduction

In this chapter, we show a detailed architecture level diagram about the integrated components of the final solution. We can see the relationships between components in addition to the main functions performed within the system prototype. The following sections will explain in more detail each module using different elements such as diagrams, screenshots and detailed usage.

4.2.2 Functional design

For an understanding of the developed system, the functional design of the notification system deserves special attention. It has been decoupled into two processes: the process by which notifications are generated, and the way these notifications reach users.

The notification system includes the mechanism whereby the users get a notification if a query result has changed in any manner. This query has to be provided and registered in the system.

The notification system will manage all users’ queries, which can have different database endpoints and they may last a long time to perform complex queries. The results have to be sent by an asynchronous communication, so this is why a novel system for transmitting information between different subsystems is used.

The technique is known as the publish/subscribe mechanism34, as explained in the section on technologies used. To make the system complete, the Administration and Notification Service has been separated into two distinct functionalities; generating new queries and continuous monitoring of the various databases for subsequent notification.

Therefore the functional design of the notification system can be represented as shown in figure 10.

---

34 Using the Publish-Subscribe Model for Applications
http://docs.oracle.com/cd/B10501_01/appdev.920/a96590/adg15pub.htm
The ability offered by this mechanism is the ability to separate query executions to various databases and result visualization, so the user does not have to wait until the query is made to continue their task. The system will warn the user when data is received or the query is complete.

### 4.2.3 Architecture

Here we present the actual design used in our system. We highlight two main modules: Notification provider & DB Operator.

The Notification provider will be manageable by the user and will be supported by a node.js server. This server is publicly accessible and will provide a web interface.

The DB operator queries the database. It will be running continuously as a Node.js server without web access to take care of delivering new data to the notification provider using the publishing mechanism explained above. The decision to use Node.js lies primarily with the easiness that the Javascript programming offers for web requests to various database rest endpoints.

To set a publish/subscribe mechanism, the existence of a "broker", which controls the delivery of the published messages and manages new data subscriptions, is required. All communication between the DB operator and notification provider will necessarily pass through the broker.

Figure 11 illustrates the designed architecture:
All publish/subscribe capable systems communicate with each other by exchanging messages categorized into different topics. So we can distinguish what action requires the received message, either towards one end or the other end of the system. The exchange is done through the broker, as can be seen in figure 12.

An example of a communication flow, in which the user sets a new query to the database, considers these steps:

1. User makes a query in the notification provider and activates it.
2. Notification provider sends the corresponding information to the relevant DB operator through the publish/subscribe mechanism.
3. DB operator saves the query data and performs the query.
4. First results are sent back to the notification provider when the query finishes.
5. DB operator will check continuously if data changes in the database.
6. Notification provider receives the data by the publish/subscribe mechanism and saves the results so that the user can review them.
7. If the user is connected, a real time notification will be sent through a web socket and will be visible in the screen.
The two main players therefore are:

4.2.3.1 Notification provider:

A diagram showing the architecture of this module is represented in figure 13.

This module provides an interface in which the user can easily add new queries to the system and be able to asynchronously provide a response to the user.

All of these layers will be covered by a user control layer where the system security relays into web session protocols. We'll use a session management module based on Node.js called "Passport" which gives us the ability to log either through a local database as well as taking into account other profiles the user may have in other social networks such as Google or LinkedIn.

Users are divided into groups according to what level of privileges they have in the system. Thus we can distinguish two different classes: Service users and system administrators. Being an administrator grants the ability to manage existing users with the possibility to modify a record, delete a user or change the type of privileges of a particular user.

---

35 Passport – Authenticate
http://passportjs.org/guide/authenticate/
4.2.3.2 DB Operator:

The DB Operator is located at the other end of the system. This autonomous system provides the capabilities to interact with the endpoints that the user establishes into the added queries. The communication path, between DB operator and notification provider is done through a publish/subscribe mechanism, so it only needs an MQTT broker in between.

The BD operator performs queries directly to the indicated endpoint. SPARQL servers usually offer a web service to provide their content. The DB operator is designed to perform queries via web services to connect to SPARQL servers with support for REST queries such as Fuseki.

BD Operator services are implemented in a Node.js server using a non-relational database MongoDB. Since Node.js is used in the notification provider and DB operator, both sites are compatible and use the same libraries.

The MongoDB database used in DB operator stores the information obtained directly from the results returned by the endpoints. It will store the old responses in order to compare with new results and to determine whether there has been any change in the endpoints.

A general outline of the components can be seen in figure 14:
4.3 Notification Provider

In this section, each of the notification provider components will be detailed. The internal behaviors of each component as well as the relationships between the various subsystems which are formed are described. The architecture presented in the previous section will be analyzed to visualize the performed implementation.

4.3.1 Express framework

As mentioned, the system will be based on a Node.js server responsible for managing all routines related to the notification provider. For this task, the Node.js server will be provided by a web framework, facilitating the programming and event management.

Express is responsible for providing management procedures as well as deploying a web server that captures external customer requests. Different functions of the framework are implemented following the model-view-controller software architectural pattern. Additionally, this framework is designed to incorporate different middlewares\(^\text{36}\) that add various capacities quickly and flexibly. In our system we will use several of these middlewares to help in programming.

Some of these components are:

1. Morgan: Provides the application logs.
2. Passport: Allows user authentication.
3. Mongoose: Is responsible for providing a connection to the database.

\(^{36}\) A short guide to Connect Middleware
http://stephensugden.com/middleware_guide/
We will describe them in the corresponding sections.

### 4.3.2 User management

The user management service includes the procedures required to provide the system with authentication and administration functions, where users can register and access the service. Like any service, the system is limited to registered users, which cannot access the functionalities if they don’t have a valid username and password.

To obtain this functionality, once the user enters the platform, he will be required to provide a username and a password. If the user doesn’t have one, he/she can sign-up into the system by selecting the corresponding option.

User registration is performed using Passport, a middleware for Node.js. We provide an authentication system based on user name and password or more elaborated ones based on access providers like Facebook or Twitter. In this first implemented version a local registration and access with user and password has been configured. These values are stored into the database “users” which for this purpose has been created in the instance of MongoDB.

The following excerpt shows the scheme that has been followed for the implementation of the user’s registry in the database:

```javascript
var UserDetail = new Schema({
    username: String,
    password: String,
    email: String,
    group: [{type: Schema.Types.ObjectId, ref: 'Group'}],
    notifications: [{type: Schema.Types.ObjectId, ref: 'QueryNotifications'}],
    admin: Boolean
}, {
    collection: 'usercollection'
});
```

The username and password fields are required for a successful user localization in the collection. Furthermore, it has been incorporated a field of notifications identifiers for a cross connection with the user associated notifications, useful for viewing and retrieving individual user notifications.

Since various types of users with different level of access are expected to coexist, a new collection has been added to the database to catalog the registers we have of the different users. This allows us to know the access level for each user on a scale of 0-5, with 0 being the level with the most limited access and 5 being the level with permanent access to all functionality.

The scheme followed for this collection is:

```javascript
var Group = new Schema({
```
group_name: String,
access_level: Number,
users: [{type: Schema.Types.ObjectId, ref: 'UserDetails'}]
},
{
collection: 'groups'
});

For each user group an administrator can configure the access level to the different functionalities. A field for storing references to users allows us to recover the level of access of each user for later use.

We defined “Administrator” as a special access level intended for users able to make changes to the database of registered users. Administrator can revoke permissions or increase privileges to a particular user. He is also able to convert existing users into administrators.

### 4.3.3 Pub-Sub data handler

The interconnection between the notification provider and the DB operator has to be asynchronous. The use of non-traditional systems is necessary to avoid waiting for the response of a request when a new query is added.

The procedure of data transmission is performed with publish/subscribe messages, in which, as it has already explained earlier in this document, is based on the existence of a broker located in the middle of communication that acts as a relay for the different end paths to where the information is destined.

To manage these communications, a middleware for the Node.js server will be added to each of the subsystems: MQTT.js is the library that will allow us to receive and send data through the publish/subscribe mechanism. Once the Node.js server boots, each system subscribes to “topics” or channels that are used in the data transmissions. We have detailed the open topics that the notification provider uses to interact with DB operator:

- **“newqueriestopic”**: Used to transmit new queries that must be added to the system. Once the user enters the parameters for the new request, this information will be sent and replicated in "BD operator" through this subscription channel.

- **“updatequeriestopic”**: Used for updating existing queries. When the user updates a request, either to change the query to perform or to correct any included data, this information will be transmitted through this channel.

- **“deletequeriestopic”**: As the name suggests, this channel is used when the user deletes a query from his profile. If an "id" query is received on this channel, the system will eliminate the item in database and cease the comparison routine of the corresponding query.

- **“queryupdates”**: In this topic, all new database results that the "DB operator" collects (also different from its previous stored result) are transmitted. That is, whenever there is a change in the response of a query, DB operator will send that information to allow notification provider to know that there has been a change.
• “queryacks”: Finally on this topic, all acknowledgement that "DB operator" sends, will be transmitted. That is, when “DB operator” confirms that new data have been received and it have been stored properly, and ack is sent. This way we get feedback on the notification provider.

4.3.4 Query Administrator

One of the key features of the notification provider is the query administrator. User can manage the different queries that they had added in his profile. There are two distinct sections for this function:

4.3.4.1 New query notification

This option will be selected if the user wants to add a new query notification to the system. The user must provide the SPARQL query to perform along with the database endpoint.

Once the user adds it, the system will notify her or him if any changes are detected in the results the query returns, and then, new data could be visualized.

The registration process of new queries is performed through a web page created for that purpose on the notification provider web server. The user has the option of detailing the name by which the query will be identified, as well as a description that describes the query. This enables an easier identification of a specific query.

To facilitate the task of filling the new query form, the user can use query templates. They are managed by an administrator or other users with permissions, and stored in a mongoDB collection called “queryTemplates”. A query template fills the query fields with a predefined query and then the user can customize it to their needs.

Using a POST request to the server, the application registers the new query in the database and also the notification provider sends the corresponding information to the DB operator, which will be responsible for carrying out the final requests to the specified endpoint.

The information transmission is done by publish/subscribe topics. The previously specified topic called "newquery" is used in this transmission to incorporate new queries into the DB operator.

Once the new query information reaches DB operator and it stores it correctly, the DB operator sends an ack message to the notification provider also through the publish/subscribe mechanism to indicate query results were successfully received. This message will be sent via the"queryack"topic.

The information sent by the notification provider to the DB operator is equivalent to the scheme that has been used in the queries database, storing all data filled by the user that describes the query. The scheme followed for this collection in MongoDB is described in the next excerpt:

```javascript
var QueryNotification = new Schema({
    queryName: String,
    queryDescription: String,
    queryEndpoint: String,
    query: String,
    users: [{type: Schema.Types.ObjectId, ref: 'UserDetails'}],
})
```
active: Boolean,
lastupdated: { type: Date, default: Date.now },
lastresult: String,
changes: Boolean,
ack: Boolean
},
{
collection: 'querynotifications'
});

- queryName: The name of the query
- queryDescription: A description of the query, useful to identify a specific query.
- queryEndpoint: A URL of the SPARQL endpoint. In the form of for example: http://dbpedia.org/sparql/.
- query: The query to be executed. One line, no breaks.
- users: An array showing the users associated with the query.
- active: Boolean, if true, the query is continuously compared waiting for changes.
- lastupdated: Time when last changes were found.
- lastresult: The last result obtained from the endpoint, encoding as a JSON string.
- changes: Boolean, if true, there are changes in the query that the user has not checked yet.
- ack: Boolean, if true, the “DB operator“ acknowledges the last update of the query.

For sending this information through the "newquery" topic, a conversion to a JSON string of the complete schema object is made in order to facilitate the reusability in the DB operator system.

An example showing how the form data is handled can be seen in the next excerpt:

```javascript
router.post('/', function(req, res){
  var newquery = new req.db.QueryNotifications({
    queryName: req.body.name,
    queryDescription: req.body.desc,
    queryEndpoint: req.body.endpoint,
    query: (req.body.querytext).replace(/\r\n|\n|\r)/gm,"").replace(/\s+/g," "),
    users: [req.user._id],
    active: true,
    lastupdated: new Date,
    lastresult: "empty",
    changes: false,
    ack: false
  });
  ...
  DB savings
  ...
  var jsonstring = JSON.stringify(newquery);
  req.mqtt.publish('newqueriestopic', jsonstring);
  res.redirect('/newquery');
});
```
Once all processes have finished in a successful manner, a redirection to the new query administration page is made and the user has added correctly a new query notification to the system.

In figure 15, the message path is shown between the different actors involved in the process of adding a new query to the database and its new result update:

![Figure 15 Data flow adding a new notification query](image)

The “Async” result can be generated at any time, as the DB operator will continuously request the query endpoint (DB) and updating the results if new changes are detected.

### 4.3.4.2 Query notification administration

If the user has several query notifications in his profile, the webpage will recover all added queries and display them in a table for visualization.

Through a GET request, the system retrieves a list of notifications associated with the corresponding user and categorizes them depending on whether they are enabled or disabled showing a green or red label located in the table.

Furthermore, since the queries may have changes that have not been seen by the user yet, all those queries that have new changes since the last time they were visualized will be marked out in a different color for a quick visual inspection.

Once the user clicks on a query, a new web page will be displayed with information relevant to that specific query. The user can verify the query data and visualize the last result obtained from the query endpoint in the last update. The results will be categorized and
shown in a table with the possibility to view the raw response. Additionally, since most likely the results contain geo data, including spatial geometries, the polygons can be represented on an OpenLayers map. A button for that purpose will be generated if the result supports the specification implemented.

Besides the view capabilities, the user can choose to update the data, i.e. modify one or more parameters and resubmit the request back to the server. The method used is a POST to the "id" of the notification, which is included in the URL and the rest of the data is passed as parameters.

Thus, the query notification database is updated with the new values and the data is resent to the “DB operator” by a publish/subscribe mechanism into the “updatequery” topic:

The process of updating the query notification registry can be shown in the next excerpt:

```javascript
req.db.QueryNotifications.findByIdAndUpdate(newquery._id,
   {$set: {
      queryname: newquery.queryname,
      queryDescription: newquery.queryDescription,
      queryEndpoint: newquery.queryEndpoint,
      query: newquery.query
   }}, function (err, query) {
   req.mqtt.publish('updatequeriestopic', JSON.stringify(newquery));
});
```

Similarly, we can also choose to delete the query system if we don’t want more notifications from that query. Clicking on the "delete" button, will proceed to the removal of the information saved in the database and a new message will be generated and published to the DB operator.

```javascript
req.db.QueryNotifications.findByIdAndRemove(notId, function(err){
  if(err){
    req.session.error = 'Error deleting';
  }
  else{
    req.mqtt.publish('deletequeriestopic', notId);
    req.session.success = 'Notification deleted successfully!';
  }
  res.redirect('/status');
});
```

### 4.3.5 Real time events

To allow users have real-time information of new changed results, real time functionalities have been added into the web server. A communication system based on web sockets has been implemented as a method of real-time communication between the web server and the client side.
What is described below will only occur if the user has an active session in the browser and it has successfully logged into the system. This condition is the one that activates the listening socket on the client side and uses the unique user ID and registration info to obtain a unique identification in the server.

Real time implementation varies significantly between versions of different browsers, because this web sockets functionality is not present in older browsers. Therefore, the used middleware supports polling as a fallback option when web sockets can’t be used.

Socket.io is responsible for checking that the used web browser supports the web-socket specification and according the case, activates this functionality in the user’s web browser. Web socket connections are used in the latest browser versions, as this is an automated process carried out by the socket.io middleware.

Once the user has logged in, a connection between the server and the user’s browser is open, allowing the transmission of different messages such as new update notifications. On every connection, the web browser asks the server if there are new notifications to serve. This, together with javascript and css code running on the user’s web page allows viewing these new notifications.

The color change of an information badge in the header of the page is done by the following snippet:

```html
<script>
    var socket = io();
    socket.send('{{user._id}}');
    socket.on('message', function(msg){
        console.log("new message: " + msg);
        $('#realtimetab').css('color','red');
    });
</script>
```

On the server side, the notification provider has to manage the reception of this new type of message using the framework provided by socket.io. Each time the web browser sends a message with its identifier, the server examines whether there are changes in any query that have not revised yet. If the condition is met, the server notifies the user.

```javascript
socket.on('message', function(msg){
    console.log('new message: '+msg);
    dbmodel.QueryNotifications.count({
        users: {$in:[msg]},
        changes: true
    },function(err, count){
        if(!err){
            if(count != 0)
                socket.send('changes');
        }else{console.log("error in query"+err);}
    });
});
```

In addition, each time a new result update from the server reaches the notification provider through a publish/subscribe message, means that a query has new results changes, so the
notification provider sends a new real time notification to the user as long as it is connected at that moment with an open session with the server.

### 4.4 DB operator

In this chapter, the other main entity of the system design is explained. It is called “DB operator” as its main role is to communicate with the SPARQL databases. It will detect new changes in any query results that are constantly being monitoring. It is communicated with the notification provider over the publish/subscribe mechanisms described above and uses a similar set of software components to work.

The core of the DB operator is also a Node.js server, but this time the number of frameworks that have been added to it is less, since there is no need to implement a web server. As a middleware, MQTT.js is used for communications and Mongoose for interconnection with the local database, in which active queries and their responses are saved.

#### 4.4.1 Query management

Although query management is not as thorough as in the notification provider, in this module it is important that new queries that arrived at DB operator have to be properly stored and available to the comparison routine that compares the previous results of the query to search for changes.

The topics subscribed in the publish/subscribe mechanism to receive the data sent by the notification provider are: “newqueriestopic”, “updatequeriestopic”, and “deletequeriestopic”.

For each different topic, the message received comes with a different type of request, which should be properly treated.

##### 4.4.1.1 New query

The most important topic is related to the reception of a new query. The message is received encoded as a JSON string with all of the data sent by the notification provider. Once it has been decoded, the query is stored in the database and an ACK is returned to indicate notification provider the correct reception of the message.

Subsequently, the query must be executed to retrieve the first response of the results. This first response will be stored into the database asynchronously and routed back to the notification provider so the user who made the request can visualize the results.

```javascript
mqttclient.on('message', function(topic, message, packet) {
    if(topic == 'newqueriestopic'){
        var newquery = JSON.parse(message);
        newquery.save(function(err,newquerydata){
            mqttclient.publish('queryacks', objJson._id);
        });
        var results = getResults(newQuery);
        mqttclient.publish('queryupdates', results);
    }
}
```
Once the query is stored into the database, the responsibility of obtaining new results is transferred to the Query Executor module. All query requests are made through REST methods getting the new results from the endpoints contained on the query configuration.

### 4.4.1.2 Update query

After reception of a message through this topic, the system updates the query and stores it into the database. An ACK is sent back to the origin and subsequently, the new query will be executed. If the result is different from the previous saved result, a new message will be sent to the notification provider.

### 4.4.1.3 Delete query

The message received through this topic contains the query unique identifier to be deleted. Once it has been received, the content of the database is deleted and the corresponding query stops running. Subsequently, an ACK is sent back to confirm the origin a successful delete.

### 4.4.2 Query Executor

This is the main module of the DB operator. It is responsible for performing requests to different queries endpoints, and contains a routine which is repeated indefinitely for continuous monitoring of all configured queries.

The requests to the database endpoints are done through REST methods. Configured database endpoints must be compatible with this specification. Examples of these servers satisfying this condition may be "Virtuoso" or "Fuseki", consuming sets of RDF data and offering them for web access.

Once the comparison routine is started, all queries stored in the database will be executed and each query requested to the corresponding endpoint. The results are compared with those previously stored, and if they don’t match, the new data is sent to the notification provider through the publish/subscribe mechanism.

In the next excerpt, after a successful response from the query endpoint, the results are compared and updated.

```javascript
res.on('end', function() {
  if(res.statusCode == 200){
    if(body != query.lastresult){
      console.log('NOT THE SAME');
      query.lastresult = body;
      query.lastupdated = new Date;
      query.changes = true;
      var jsonstring = JSON.stringify(query);
      mqttclient.publish('queryupdates', jsonstring);
      QueryNotifications.findByIdAndUpdate(query._id,
        { $set: {
          lastresult: query.lastresult,
          lastupdated: query.lastupdated,
          changes : query.changes }}),
        function (err, querydata) {
        console.log("Update added to BD on serverside");});
      }else{console.log(Same string!');}}
    else{console.log(Same string!');}}
```
This routine should be run continuously, so the above methods are placed into a timer which is invoked by:

```javascript
routineTimer = setTimeout(arguments.callee, repeatTime);
```

In this way, stored queries are continuously monitored into the DB operator.

### 4.5 Usage and user interface

These components are required by our Administration and Notification Service for a correct execution:

- MQ Telemetry Transport protocol Broker: such as Mosquitto.
- MongoDB.
- SPARQL Enpoints: Running as web services.
- Node.Js: 2 instances will be running simultaneously.

When these services are running, the notification provider must be available at the predefined port. Once a user gets into the website, it must login with a correct credentials or register as new user. In figure 16 the welcome page to the system is shown:

![Figure 16 Login page](image)

If the login is successful, and the access to the system is granted, the main page shows up. The login data are displayed and the different options are shown.
The navigation bar displays available tabs:

- Home: displays the home page of the web site.
- New query: adding new query notifications is done in this page.
- Notifications: a list of current notifications is shown in order to manage them.
- A badge-icon: it will become red if the user has un-seen notifications
- Administration tab: Only visible for admin users. Managing users and query templates.
- Profile: manages the user profile
- Log out: leaves the current user session.

The interface to add a new notification system requires filling the new data to store the query in the notification provider. Once you click the “send and activate” button, the system stores it and the user starts to receive notifications.
If the user does not want to perform a query from scratch, the system proposes different default queries templates which they are saved and manageable by an administrator. These queries are preconfigured and can be searched through the search field located at the top of the window. Subsequently predefined values can be changed to suit the user preferences and to customize the query.

When the user has some notifications added to the system, they will appear in the "Notifications" tab, where the user can see all active and disabled queries in his profile:
As it is shown in figure 20 the user has three notifications, 2 of them are active. None is marked with a special color, so all data is reviewed and there has been no change in the databases since his last access.

In the event of a new notification of changes in the database, the corresponding query would have a different color, as it can be seen in the figure 21:

If the user wants to know the last results of one query, clicking in the desired query opens the display page, where the user can access the data of the query and review the last result obtained from the query endpoint. The user is also allowed to modify the actual query, and submit it to the notification provider again.

Results can be viewed in a table and the geometries are represented in an OpenLayers frame, as shown in figure 22.
Focusing on the administrative part of the system, each user can edit his or her profile and edit the password for accessing the site. The Profile page is represented in figure 23.

An administrator user can view the current list of the registered users in the system and their associated notifications. Admin users can modify the role of other user or change it to a different grant level.

Figure 24 shows the users administration page.
Figure 23 Edit profile page

Figure 24 Users administration

Figure 25 shows the single user administrating page, where an admin user can modify the role associated with a single user.

Figure 25 Single user administration
5 Pilot exploitation

This section describes how we can validate the developed services in the SmartOpenData pilots that are currently being defined. These two pilots expressed interest in the current version of the application so far: Agroforestry Management and Environment Data reuse.

5.1 Agroforestry Management (Spain & Portugal)

5.1.1 Overview

The Administration and Notification Service will be validated in the Spanish-Portuguese pilot, which has the following objectives: (i) optimize agroforestry land use; (ii) diversify the rural economy and improve the quality of life of rural communities; (iii) improve asset management activities and soil; (iv) preserve biodiversity and environment; (v) involve the public and private sectors on issues related to forest management; (vi) improve the efficiency of administrative processes; (vii) improve watershed management by integrating information of land use; (viii) build consensus on legal and regulatory requirements in land use.

Data providers and services often encounter two problems or questions to answer:

- How to deliver content and functionality within the legislative obligations (INSPIRE) initiatives (Open Government Partnership) and user expectations?
- How to link contents and services from their portals through standard interfaces?

These problems arise from the lack of mechanisms to consult public information, and also by the low quality of information often associated with it.

This pilot describes an example of recommended seeds for growing both agricultural and forest in an area:

An owner or forester wants to purchase some seeds. For this, he must know the location of the closest seed nurseries or public seedbeds. In the case of seed nurseries, finding this information is more difficult because the reservation for visits is not accessible online. The current procedure is to contact an officer of the public administration and complete a formal request for reservation. The officer must complete documentation associated with the details of each visitor. In addition, a supervisor must visit the public seedlings to ensure that the seeds are collected in the right amount, and must also fill some reports containing the dates and places of gathering, company responsible for the collection, etc. This clearly results in complex and slow processes. The location of public nurseries and their availability for seed collection could be accessible on the Internet and the information entered by the supervisor about the details of seed collections could be updated in real time. Thus, any change in the status or information on any of the public seed would be immediately accessible and available to anyone who needed it.
As an additional problem, this administrative process is performed locally, not nationally. Over time, the Spanish administration receives this information from the Autonomous Communities, but this means a duplication of effort due to the absence of a unified body and the absence of mechanisms for information exchange between national and regional entities.

Another problem is the lack of analytical information in the selection of the species to grow. As explained above, foresters and landowners usually select species to plant in their fields based on experience or intuition. No information or analytical data are available to define what the most recommended species is. A geographic information system showing a recommendation on the distribution of species according to some variables would be very suitable. These variables can be geology, weather (temperature, precipitation), vegetation, orientation, slope, etc.

An interesting feature to implement is to offer, in an accessible way, availability of seeds in the seedbed in real time. Thus, owners and foresters would choose the most suitable seedbed to their interests, saving time and money.

There are many studies regarding the management of forest genetic resources, which recommend planting a species depending on the region of origin and quality of seeds. However, these studies are cumbersome and not very easy to find. They are also difficult to interpret for users without proper training and knowledge.

All these problems prevent a proper reforestation process by failing to consider relevant factors: species, region of origin, seed quality, production planning, optimum time of sowing, etc.

The result of this lack of planning is the misuse of forest genetic resources and the reduction of forest production. The solution goes hand in hand with making available the data and services needed for stakeholders to make the process easier. Figure 26 is intended to outline the need for geographic information retrieval for the study of the factors that enable proper selection of the species to grow.
5.1.2 ANS usage

This feature aims to facilitate the exchange of data and user, process and data management. The administration functionality is available for certain users, so that they can enable access to data and processing services to data providers and consumers.

Figure 27 shows the interface developed for user administration. It is possible to see a list of registered users, belonging to different groups (managers, owners of land, etc.). Each of these users may have set a notification so that they receive messages related to changes in the repository under certain conditions; for example if a public seedbed regains stock of *Pinus sylvestris*, sought by user *propPinarMaceda*.

![Figure 27 User administration panel and visualization of associated notifications](image)

The notification provider notifies users when new data exist related to performed requests, according to a set of rules. The notification system uses asynchronous requests by following the Pub/Sub model. The following figure shows the notification process that was defined and is being developed.

![Figure 28 Process for notification of changes in requested data](image)

The notification provider notifies users when new data exist related to performed requests, according to a set of rules. The notification system uses asynchronous requests by following the Pub/Sub model. The following figure shows the notification process that was defined and is being developed.

The user makes an information request for geospatial information to the subscription server, also known as notification provider. This server stores the request and associates it to the user (1). Also it sends a subscription message to the Broker, which is the element responsible...
for maintaining the correspondence between existing publishers and subscribers. The publisher, or DB operator, stores the request and the result of the operation (2) (which can deliver to the user if he is interested in the result). The DB operator continues to execute the query with a certain frequency and communicates via the Broker if the results returned by the database have changed from the last request. Thus the notification provider recognizes changes and notifies the client the requested information, which can consist of such changes or a notification that they have occurred.

5.1.3 Evaluation

The architecture defined in this deliverable will be validated in a real scenario framed in the use case of recommendation for seed selection, defined in Section 5.1. The pilot will be conducted as collaboration between Spain and Portugal by leading companies and organizations in the sector (TRAGSA and Direção-Geral do Território). The scenario is located in the municipalities of Allariz and Macera, in the region of Galicia, northwest of Spain. This area is mainly used for agriculture, being tourism and urban development secondary activities in this area.

The participants in this pilot are owners of forests and parcels used for agriculture, public administration, seed distribution and sales companies and plants and seeds nurseries. These stakeholders are involved in a chain that begins with selecting the best species for a given parcel and finalizes with the harvest and acquisition of seeds after determining the location of the harvest.

The main concern of land owners and foresters is to know what kind of seed is most appropriate for each of the parcels. Then, they need to find out where they can get the seeds of this particular species. Obviously they can visit local seed nurseries, as they exist in Maceda and they are also participating in the pilot. However, they can also go to public seedbeds. These seedbeds, controlled by the administration cannot be visualized online, so they should contact an administration officer to obtain information about which seedbeds are available and their location. Once done, they should contact seedbeds for information of the available seeds, their quantity and quality.
To make public information available to all interested parties and to visualize the location of public nurseries, the stock and quality of seeds available to them we define a model that presents the necessary services and data sources to be used. We defined a diagram (Figure 30) describing this information.

As shown in the figure, for the seed selection phase we consider soil information in the area of Maceda and Allariz, from databases of plant taxonomy, botanic, etc., and databases containing genetic testing such as *Silvadat*. The scenario also considers the use of phytoclimatic models and growth simulators.

Applications and services that support phytoclimatic models, known as *Climaforest*, provide insight into the degree of suitability of a species in a given territory. Growth simulators such as *GESMO*, make it easy for foresters and forest managers using complex growth models, hiding mathematics and data processing complexity from user interfaces and providing therefore a simple tool to domain experts.

For the transfer to industry, we consider timber as the main interested industry in the regions of Maceda and Allariz, as they predominantly contain forests. The transfer to industry will be through the Centre for Innovation and Technological Services for Timber of Galicia, belonging to the *Consellería de Economía e Industria* of the *Xunta de Galicia*.

The expected results after the launch of this pilot are, in general, to make processes more automatic and transparent so that they can publish, in an accessible way, the information relevant to stakeholders at each stage. Particularly it shall consider:

- Selection of the most suitable species for each parcel.
- Selection of a public seedbed based on species that administers and availability.
- Selection of a nursery based on seed species, quality and proximity.

It shall be considered as metrics to evaluate the benefits of this pilot the following:

- Degree of cooperation of the various participants in the pilot to develop and adopt the solution.
- Records of requests for the selection of seeds and seedbeds.
- Number of executions of those services.
- Comparison of use of these services against the use of traditions consultation services for seedbeds and seeds. Comparison with the ratio of queries to the traditional services of the previous years.
- Use of quality indicators defined in the ISO/IEC 25022 standard, which provide satisfaction, suitability, usability, reliability, information security and maintainability.

This information will be analyzed to improve the model deployed in the pilot, from the point of view of quality indicators and the features offered. The information collected will be processed to identify potential technical problems and usability deficiencies.

### 5.2 Environmental Data reuse

#### 5.2.1 Overview

The Slovakian Environmental data reuse pilot aims to investigate possibilities to improve searching for environmental geospatial data by the creation of spatial linked data and their further reuse. Whilst achieving the abovementioned outcomes, the main considerations will be laid on support of interoperability standards, consistency with on-going related initiatives (INSPIRE, eGovernment, open data) and response from the stakeholders.

#### 5.2.2 ANS usage

We used the dataset provided by the Slovakian pilot\(^{37}\), coming from the Slovak Environmental Agency (SEA\(^{38}\)) which contains information about harmonised protected sites in Slovakia. The dataset is based on the INSPIRE Data Specification.

This dataset was used and processed by the following activities:

- Import dataset into a *Fuseki* SPARQL endpoint.
- Generate some SPARQL queries to provide as examples
- Register queries in the Administration and Notification service
- Perform some changes in the *Fuseki* dataset to activate new notifications.
- Display retrieved notifications in a table.
- Represent notification results into an Open Street Map by using the *Openlayers* JS library.

An example of a SPARQL query could be as follows:

```sparql
PREFIX drf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX j.0: <http://inspire.jrc.ec.europa.eu/schemas/gn/3.0/>
PREFIX j.1: <http://inspire.jrc.ec.europa.eu/schemas/ps/3.0/>
PREFIX j.3: <http://www.opengis.net/ont/geosparql#>
```

\(^{37}\) [http://data.sazp.sk/parliament/](http://data.sazp.sk/parliament/)

\(^{38}\) [http://www.sazp.sk/](http://www.sazp.sk/)
SELECT *
WHERE {
  SERVICE <http://localhost:3030/gisairdf/query> {
    ?fGeom j.3:asWKT ?fWKT .
    ?res j.1:LegalFoundationDate ?lfd .
    ?inspire j.2:namespace ?localId .
    ?siteDesignation j.1:designationScheme ?designationScheme .
  }
}
LIMIT 5

At this point, a piece of the information returned from the server is as follows.

"results": {
  "result": [{
    "binding": [
      {"-name": "res",
       "uri": "http://geop.sazp.sk/id/ProtectedSite/ProtectedSitesSK/SKNATS835"}
    ],
    {"-name": "fGeom",
     "uri": "http://geop.sazp.sk/id/ProtectedSite/ProtectedSitesSK/geometry/SKNATS835"}
    ],
    {"-name": "fWKT",
     "literal": {
      "-datatype": "http://www.opengis.net/ont/sf#wktLiteral",
      "#text": "MULTIPOLYGON(((18.750451506157948
      (...
      5.2.3 Evaluation

We expect two types of benefits from the employment of the Administration and Notification service into the Environmental Data reuse pilot. This first list corresponds to general achievements of using an application to obtain process and represent geospatial data:

- New knowledge and experience gained (Skilled experts, information about the relevant use cases, tools, methodologies, guidelines, courses, trainings)
- New data, services and metadata discovered across heterogenic technologic platforms like Spatial data infrastructures, main, deep web (Statistics from tests
comparing results of standard search outcomes with those using principles deployed by pilot solution)

- Increased awareness about the use of linked open geo data and semantic technologies on national level (Outcomes of interaction with the stakeholders via User groups)

- Strengthening of existing and establishment of new technology and domain focused communities and networks (Amount of identified related communities and networks on the beginning and end of the project)

- Provide the information to the users about the changes occurred in the underlying database (e.g. with the SmartOpenData final model harmonisation, where connection to the Environmental burdens/Contaminated sites is foreseen to take place)

Related to specific benefits of using the administration and notification functionalities. We will achieve the validation of the proposed use cases by using the service to register stakeholders, their most used queries to databases from different domains and also the classification and presentation of responses, which varies depending of the retrieved data.

If the data retrieved is non-spatial the representation of the response will be in a table indicating all of the response parameters and their values. If the retrieved data is geospatial we represent the information in a map, rendering the WKT-based geometry as a layer in OpenLayers, and using as a base map OpenStreetMap cartography.
6 Conclusions

This deliverable defines the work performed under the SmOD project related to the Administration and Notification service.

Administration and notification facilitates data exchange and user management. A Pub/Sub model is used so that end-users obtain real-time change in data in which they are interested.

First of all, this service is introduced in the context of the semantic front-end facilities. Other tools to be developed in the scope of the Semantic Front-End facilities are the Distributed Semantic Indexing Infrastructure and the Visualization Framework. Relations between the Administration and Notification Service and these two tools are explained. Also the relation between these tools and the systems and solutions provided by the other work packages is described, in particular WP3 (Data modelling and Linked Open Data (LOD) alignment) and WP5 (Demonstration Pilots).

This deliverable defines the functional architecture, APIs and modules of the Administration and Notification Service. It includes a communication layer composed of several functional components such as: communication brokers, proxies and gateways. It also describes the different involved components and how they relate.

The service is being used in different pilots:

- The architecture defined in this work will be validated in a real scenario of optimal selection of seeds for timber production in the regions of Maceda and Allariz, in Galicia, Spain. It aims to demonstrate the impact of the sharing and use of data from different sources and its application to a scenario in which several actors (companies, public authorities and forest and parcels owners) are involved.
- Application testing and validation were started under the Agroforestry Management and Environmental data reuse pilots. These experiments have allowed us to verify the correct operation of the system and all of its components and realize some improvements needed for a better user experience.

As preliminary results of the development of tools and structured data we can say that the reliability and accuracy of data visualization is always conditioned by data quality and the efficient operation of data repositories, so, a correct modelling, structuring and linking of data is essential to ensure the success of the pilots.

As future work, we intend to complete our data model for the described notification scenario and proceed with a semi-automatic transformation to RDF of the performed requests and the results of notifications received by registered users. Automation in this field is necessary because any change to the data must be translated into RDF and re-linked, so that applications that consume the data from SPARQL access points can be aware of these changes.

Alternative approaches such as using the ambitious proposal for GeoJSON-LD, extension of JSON-LD standard, will also be explored, to add this alternative to the catalogue of options.

---

http://www.w3.org/TR/json-ld/
provided to future actors willing to extend their spatial information into the semantic dimension.
References


