Open API Design and First Release

MARCH 2015

DELIVERABLE

Project Acronym: **SDI4Apps**
Grant Agreement number: **621129**
Project Full Title: **Uptake of Open Geographic Information Through Innovative Services Based on Linked Data**

D3.3.1 OPEN API DESIGN AND FIRST RELEASE

Revision no. 01

Authors: **Premysl Vohnout (CCSS)**
Stein Runar Bergheim (AVINET)
Michal Šrédl (CCSS)
Michal Kepka (UWB)
Alessandro Pironi (HYPER)
Matteo Lorenzini (HYPER)

Project co-funded by the European Commission within the ICT Policy Support Programme		
Dissemination Level		
P	Public	X
C	Confidential, only for members of the consortium and the Commission Services	

REVISION HISTORY

Revision	Date	Author	Organisation	Description
01	8/03/2015	Matteo Lorenzini	HYPER	Initial draft
02	12/03/2015	Alessandro Pironi	HYPER	Contribution with initial suggestion about components and service
03	13/03/2015	Stein Runar Bergheim	AVINET	Definition of development's methodology and suggestion about service and components
04	20/03/2015	Premysl Vohnout	CCSS	Contribution with suggestion about service and components
05	31/03/2015	Matteo Lorenzini	HYPER	Final Version and Formatting

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

This document presents the design and the proof of concept implementation of API service will be used in SDI4APPS framework. The document starting with a brief introduction about API and the interaction with the other SDI4APP framework's component then will be examined the methodology used for the definition of the API's architecture. Will be described the different steps used for the development of the service and for definition of the different functionalities that the service will have to cover. Finally, will be described different kind of service previously identified in their basic and advanced functionalities.

1 INTRODUCTION

In the context of this document we mention API with the implied meaning given in the context of the SDI4Apps, where it can refer either:

- a JavaScript library for HTML5 to be embedded in client applications
- a server side Web Service to query, retrieve and manage data to be used in client applications
- a remote cloud service provided to manage the platform where the application is hosted, needed for managing server-side scalable data processing
- a remote cloud service to be used for scalability and high availability management

In this document we define the details of an open API that leverages both server side services, through the SDI4Apps platform services, and client side services, in the Advanced Tools API. The API will grant easy access to data to be used in the end applications.

This document covers both the design of specific non standard API, and the relationship with standard API that have been integrated so far in the first prototype as a first implementation.

1.1 Relationship to the SDI4APPS platform

The SDI4Apps platform is a cloud-based geospatial application hosting environment that virtualizes the hardware and operating system infrastructure from the perspective of organizations using the platform. In addition to providing server power and operating system, the platform is pre-configured with selected server applications, each of which expose standard or non-standard APIs.
Each of these server applications are a realization of the abstract “enablers” described in D3.1. The OpenAPI is a wrapper “on top” of these APIs that provides end-users with an integrated and homogeneous set of methods to interact with the platform.
These methods are in turn exposed as Web Services over HTTP, enabling the platform to be invoked from any client-application capable of issuing and HTTP requests, including but not limited to the SDI4Apps pilot applications.
The OpenAPI benefits from the Cloud technologies in two ways:

- If widely adopted, the Cloud architecture allows for quickly scaling up the platform to handle more parallel requests. This is done by “firing” up additional server nodes either incrementally as the service evolves - or on demand.
- Second, the Cloud architecture, along with the API permits single operations to be branched and partially executed on more than one server. This solves the challenge of server-side simulation and analysis models that run too slowly to deliver synchronous results to requesting HTTP clients.

The SDI4Apps is not a singleton but may have any number of instances running in parallel in different places and for different purposes. Each instance may have one or more server nodes, all of which expose the API through a single endpoint through a load-balancing proxy server.
The virtual platform that is being hosted by SDI4Apps partner MU during the project life-time may, if the business model dictates it, be hosted in other Cloud environments, e.g. Amazon EC2[1]. Furthermore, the SDI4Apps platform may even be hosted by individual enterprises. The objective of this architecture is to ensure flexibility and portability of the platform in order to secure its sustainability beyond the project funding period.

[1] "AWS | Amazon Elastic Compute Cloud (EC2) - Scalable ..." 2006. 20 Mar. 2015 <http://aws.amazon.com/ec2/>

The OpenAPI builds on and expands the set of enablers listed in D3.2.2. The first set of enablers that were addressed by the project were naturally the most important components of the platform:

- Data storage services in the form of PostgreSQL + PostGIS
- Web map/feature services in the form of MapServer

However, in order to provide a meaningful API that provides functionality that goes beyond existing mainstream tools, it is necessary to extend, or specialize, these basic enablers into a more comprehensive set of components. This is also part of the project plan, something that is implicit from the fact that enablers are incrementally released between months 7 and 30 of the project execution period.

1.2 Relationship to advanced tools API

The Open API ends with the Web Services, from there on out, it is the responsibility of the applications invoking it to format valid Web Service requests and correctly parse Web Service responses.
This is not always a trivial task. Some stateless operations such as searching a public dataset may be implemented in a simple manner - however, many stateful or semi-stateful operations will require complex and non-trivial chains of requests to the OpenAPI in order to reach the desired end.
For this purpose, the OpenAPI is supplied with a client-side user interface in the form of a JavaScript library for rapid geospatial application development.
SDI4Apps pilot applications will be built as HTML5 and JavaScript applications and will invoke the API through the s4a.js library that is further described in D4.2 “Advanced Tools API”.

1.3 Relationship to pilot

The pilots are part of the SDI4Apps project in order to validate the usability and usefulness of the platform from the perspective of third-party application service providers and end-users. The OpenAPI is going to provide re-usable functionality for all the pilots but:

1. The OpenAPI will not provide all functions required for special use-cases in each pilot application - these will be part of the individual application code. Only re-usable components will be implemented in the API
2. The OpenAPI will not be restricted to functions required by the pilot applications. The OpenAPI must attempt to anticipate a wider range of use-cases and more “cutting edge” features than those typically required by mainstream geospatial web applications.

Whereas the OpenAPI will not provide custom methods for all “one-of” features in pilot applications, all the components of the platform are at the disposal to application developers if they choose to host their applications from a server with an SDI4Apps OpenAPI instance. That is, pilot applications may exploit software specific features that are not part of the OpenAPI but that is exposed by the individual software components in the platform.

2 METHODOLOGY

The Open API Design methodology defines the steps to undertake in order to design the services that constitute the API itself. The methodology goes through the following steps:

- identification of the services to be provided by the Open API
- definition of the goals and requirements of each service
- assessment of the technology and standards available for the implementation
- specification of how to exploit the service goals and how to meet its requirements, including choice of technology and standards
- definition of examples to be used both as part of the service documentation and as a starting point for the testing phase of the service implementation

The tools that can support the design of the Open API are the tools commonly used in software design, to better capture and communicate the service concept, goals and mean of use. We highlight here:

- UML diagrams to design and document the service in all its aspects: interface, components, data structure, use cases, activity and interaction
- code markup and syntax highlighting of the examples for both the service functions and data structures used in the service
- references to documentation and support, to support the design with the details that cannot be captured in its definition

On this topic it is vital to have proficuous discussion among the partners of the consortium, through meetings, communication tools (chat, conference and so on), shared documents and other means, to achieve consensus on the decisions in the design. This is especially true in the identification of the services and the definition of their goals and requirements, but it also applies to the assessment of the technology to be used.

3 OPEN API SERVICES

The OpenAPI consists of two levels of functionality; basic and advanced services. This grouping reflects the implementation sequence as well as the centrality/importance of the service in terms of satisfying identified and/or envisaged end-user requirements.

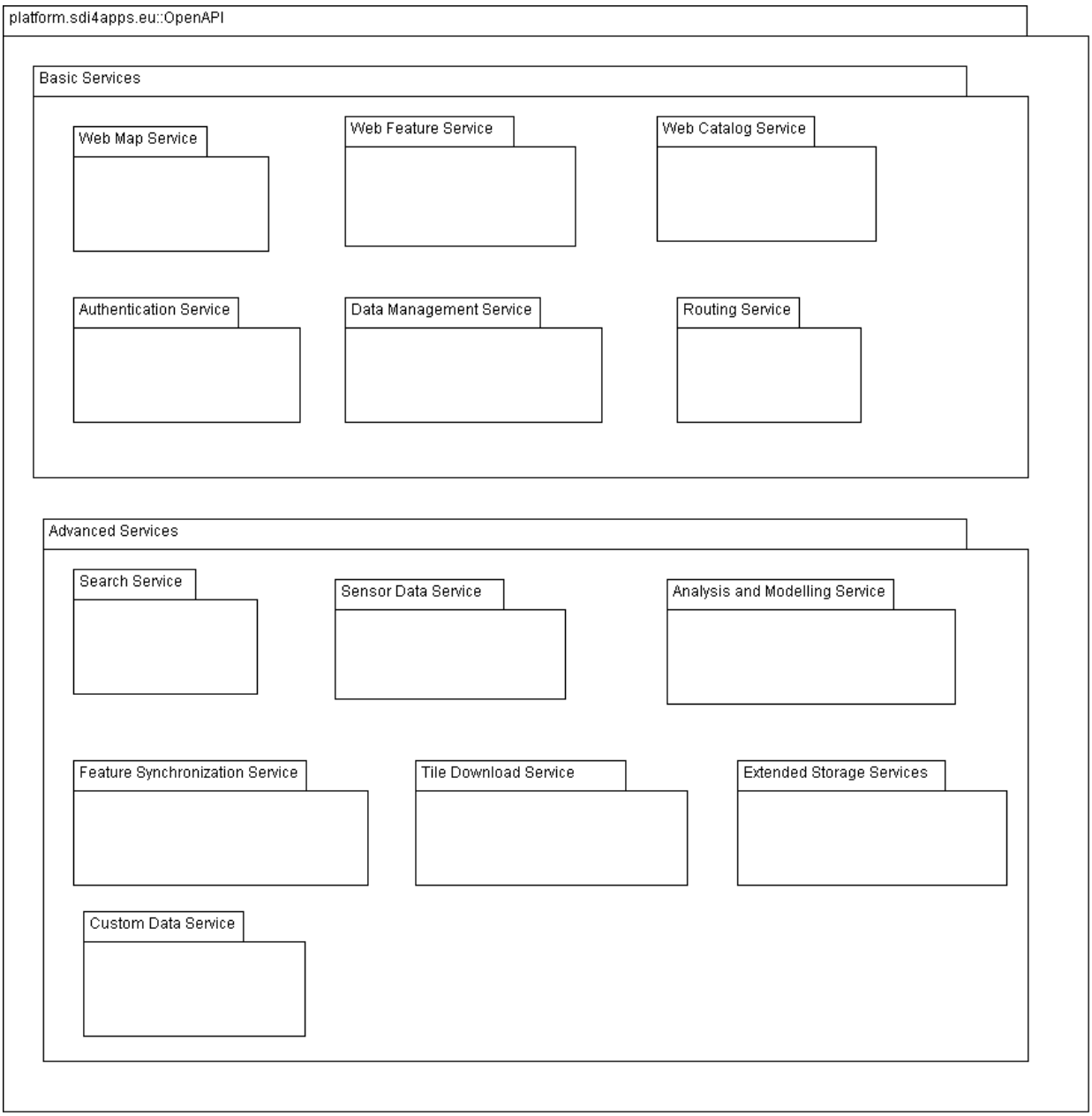

Figure 1: Overview of the basic and advanced services in the SDI4Apps OpenAPI

3.1 Services include in the first relase

At the time of release of the first prototypical version of the OpenAPI in M12, only core data storage and OGC/OWS services are implemented in their final form.

- PostgreSQL + PostGIS: basic data storage service
- Virtuoso: SPARQL, GeoSPARQL
- MapServer: WMS, WFS, SLD, WFS 2.0, WCS, WMC
- MICKA: CS-W

4 BASIC OPEN API SERVICES

In the experience of SDI4Apps technical partners the majority of spatial web applications share a common set of very basic components: they implement (1) a map interface capable of displaying (2) one or more custom datasets on top of (3) a basemap. Furthermore, they implement controls to interact with the map including (4) buttons to pan, (5) zoom, (6) query through point-and-click or point-and-drag and (7) turn on/off thematic layers and base maps.
The web services needed to implement these components in client applications are classified as “basic services” and are positioned at the front of the implementation queue.
These services are all based on well-tested, well-proven protocols and standards. The emphasis is therefore not on the technical sophistication of the services but rather the performance, scalability and robustness under heavy load.

4.1 Web Map Service

URL: http://portal.sdi4apps.eu/geoserver/wms

One of the fundamental services of the OpenAPI is the implementation of the WMS[2] interface that is enable HTTP clients to issues stateless requests for map-images and feature identification from a map server

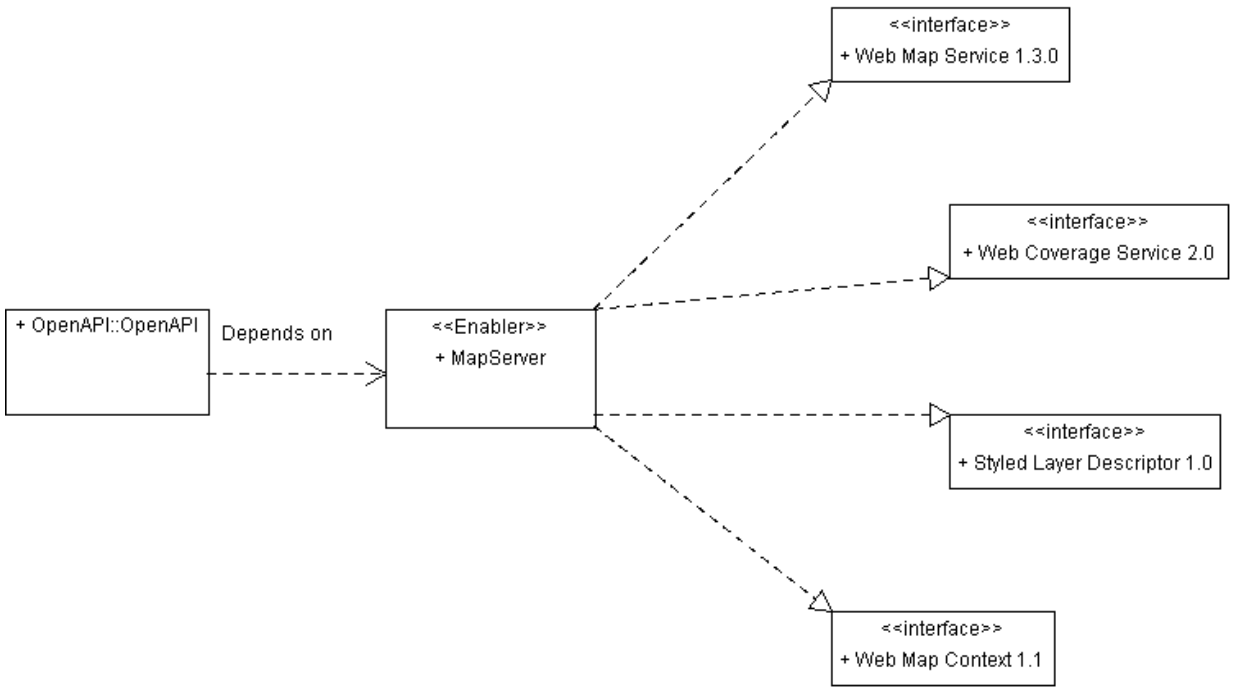

Figure 2: Realization of the WMS interface in the OpenAPI

[2] "Web Map Service | OGC." 2006. 20 Mar. 2015 <http://www.opengeospatial.org/standards/wms>

The WMS 1.3.0 interface is realized through a map server, in the case of SDI4Apps, the platform permits the use of two different map servers; GeoServer and MapServer. In the UML class diagram above, the realization of the interface is achieved through the use of the enabler MapServer. In addition to WMS, MapServer also provides other related services that are of relevance to the OpenAPI. The Web Coverage Service 2.0[3] interface provides methods for interacting with spatial raster and image data, the Styled Layer Descriptor 1.0[4] interface provides methods to manipulate the graphical representation of map elements - and the Web Map Context 1.1[5] interface permits persistent storage and recall of map states and conscious compositions created during user sessions. WMS has matured as a standard and does not hold the interest of novelty, yet its performance is vital to most spatial applications as it is responsible for drawing background maps that are loaded in large volumes. A link to the comprehensive formal specification of the WMS standard is included in the footnote references of the current chapter and is therefore omitted from the document.

4.2 Web feature service

URL: http://portal.sdi4apps.eu/geoserver/wfs

Another fundamental service in the OpenAPI is a realization of the WFS 2.0[6] interface. This service permits download and transactions of spatial vector data between a HTTP server and client.

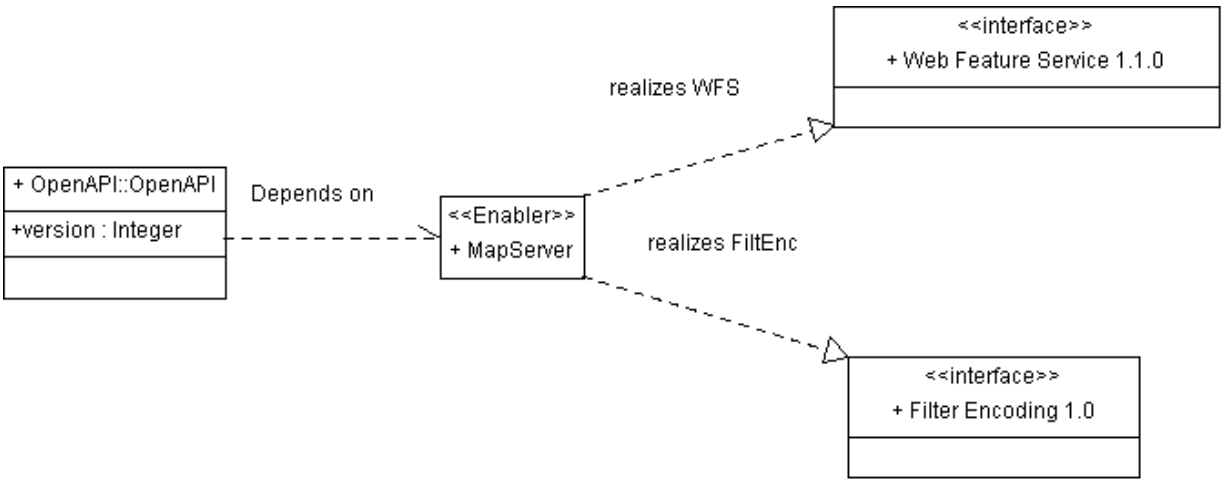

Figure 3: Realization of the WFS, WFS-T interface in the OpenAPI

While WMS provides superior performance in terms of drawing composite graphics such as base maps, WFS comes to play when the client applications need access to vector data for further processing or export to file formats.

Spatial data are represented as GML, requests may be in the form of SOAP/XML or simple HTTP Get requests. Service responses are always encoded as XML. While this allows for ease of parsing due to the widespread support for XML, it also means a significant semantic overhead on top of the actual

[3] "Web Coverage Service | OGC." 2006. 20 Mar. 2015 <http://www.opengeospatial.org/standards/wcs>
[4] "Styled Layer Descriptor | OGC." 2006. 20 Mar. 2015 <http://www.opengeospatial.org/standards/sld>
[5] "Web Map Context | OGC - Open Geospatial Consortium." 2003. 20 Mar. 2015 <http://www.opengis.org/docs/03-036r2.pdf>
[6] "Web Feature Service | OGC." 2006. 20 Mar. 2015 <http://www.opengeospatial.org/standards/wfs>

geographical data. Hence, the SDI4Apps OpenAPI also envisages custom services that represent spatial data in less bulky formats such as GeoJSON or even CSV/TSV files.

While the client/server applications reside on the same platform, the performance degrade from data transfer is neglectable. However, in a distributed software application architecture where individual applications consume resources from a central server it is desirable to keep the data transfer at an absolute minimum. In addition to WFS, GeoServer also implements the Filter Encoding 1.0[7] standard. This permits advanced spatial filters to be applied to queries, in principle allowing for spatial analysis operations being handled directly by the GeoServer, bypassing the need for any custom code. A link to the comprehensive formal specification of the WFS standard is included in the footnote references of the current chapter and is therefore omitted from the document.

4.3 Transactional web feature service

URL: http://portal.sdi4apps.eu/geoserver/wfs

While both WMS and WFS are publishing centric services, the WFS protocol permits for spatial CRUD transactions through a mechanism formerly known as WFS-T, now a part of the core specification.
With the advent of citizen science, crowdsourcing[8], location based services[9] and the ubiquity of Internet connected mobile devices, the ability to interact, not merely read/receive is increasingly important. Transactional WFS permits application scenarios where any number of users can collaborate to build or maintain a centrally managed/stored dataset. A link to the comprehensive formal specification of the WFS standard is included in the footnote references to the preceding chapter and is therefore omitted from the document.

4.4 Catalog service

URL: http://portal.sdi4apps.eu/php/metadata/csw/index.php

Being a core INSPIRE service, a data discovery oriented metadata catalog service is also included in the SDI4Apps OpenAPI platform. The set of methods included in the API conforms to the OGC Catalog Services for the Web[10] (CSW) and are realized through the existing software component MIcKA.

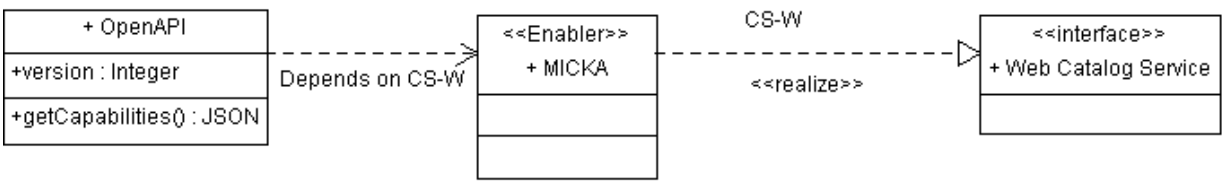

Figure 4: Realization of CS-W interface in the OpenAPI

[7] "Filter Encoding | OGC." 2006. 20 Mar. 2015 <http://www.opengeospatial.org/standards/filter>
[8] "Crowdsourcing - Wikipedia, the free encyclopedia." 2006. 20 Mar. 2015 <http://en.wikipedia.org/wiki/Crowdsourcing>
[9] "Location-based service - Wikipedia, the free encyclopedia." 2005. 20 Mar. 2015 <http://en.wikipedia.org/wiki/Location-based_service>
[10] "Catalogue Service | OGC." 2006. 20 Mar. 2015 <http://www.opengeospatial.org/standards/cat>

MIcKA, manage datasets, metadata
A link to the comprehensive formal specification of the CS-W standard is included in the footnote references to the preceding chapter and is therefore omitted from the document.

4.5 Routing service

The OGC has specified the methods of a routing service as part of the OpenLS[11] standard. This specification is targeted at road network routing. Routing algorithms for road network navigation are widely implemented in location based services such as personal navigation systems.
Network calculations do however have other commercial business cases also, for an example within utilities management. In SDI4Apps, routing therefore applies to any pgRouting compliant topological network, regardless of logical feature type. A network can be a road network, an electricity distribution grids or a utility pipelines - as long as it is a topological network dataset consisting of nodes and links, the API will handle it. For this reason, the OpenAPI route service will implement the OpenLS routing service interface but also additional methods to permit for more generic route calculations that supports a wider set of potential use cases. This is interesting bot as a stand-alone service and as a partial step of an analytics and modelling execution chain.

[11] "Location Service (OpenLS) | OGC." 2008. 21 Mar. 2015 <http://www.opengeospatial.org/standards/ols>

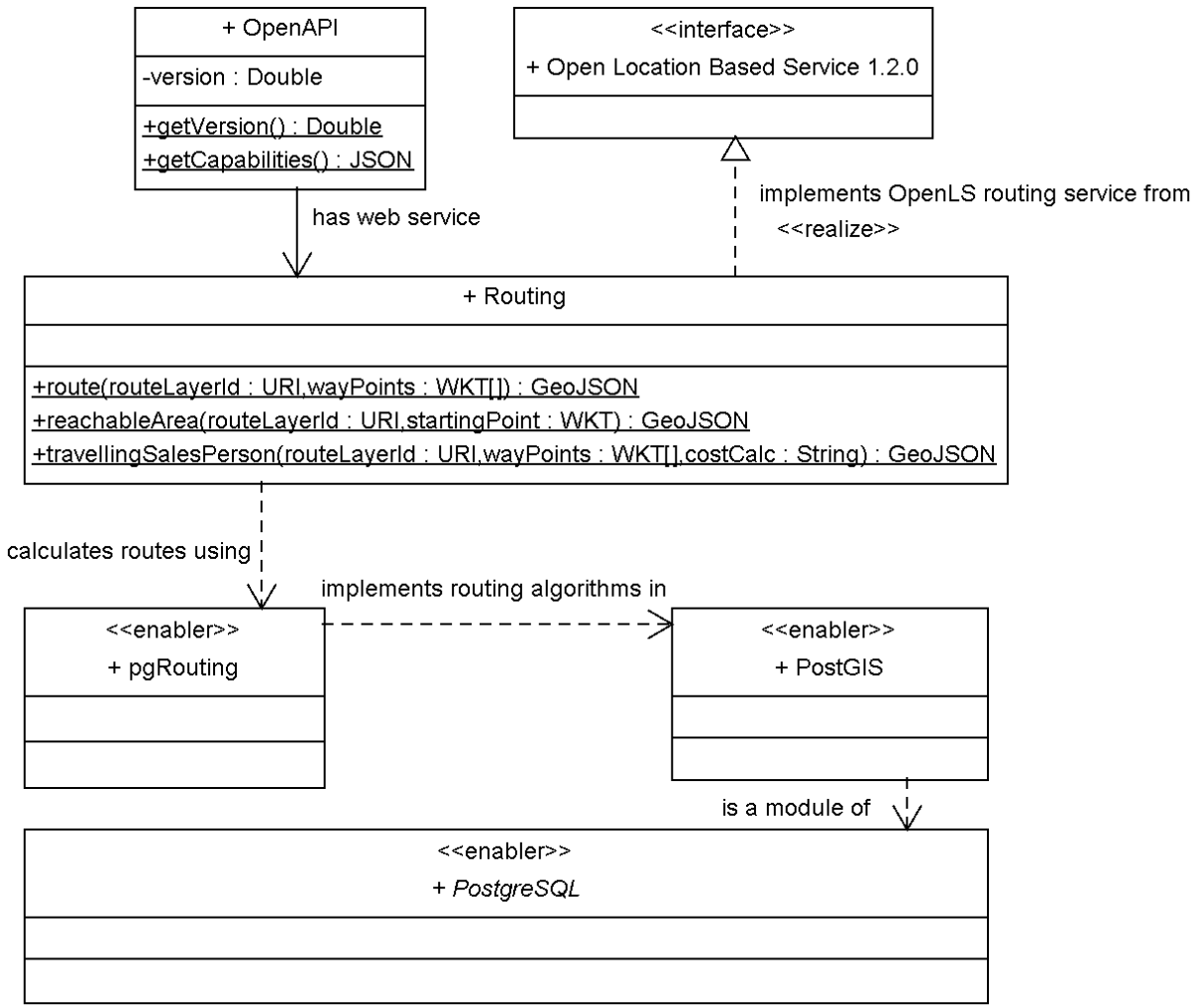

Figure 5: Realization of the partially custom routing service in the OpenAPI

As shown in the figure above, the key component for the realization of this service is the pgRouting[12] module that sits on top of PostGIS - that in turn sits on top of PostgreSQL, all of which are already enablers in the SDI4Apps platform.
Beyond regular turn-by-turn route calculations from point A via points B and C to D, pgRouting supports an extensive set of network analysis algorithms, including:

- All Pairs Shortest Path, Johnson's Algorithm
- All Pairs Shortest Path, Floyd-Warshall Algorithm
- Shortest Path A*
- Bidirectional Dijkstra Shortest Path
- Bidirectional A* Shortest Path
- Shortest Path Dijkstra
- Driving Distance

[12] "pgRouting Project — Open Source Routing Library." 2010. 21 Mar. 2015 <http://pgrouting.org/>

- K-Shortest Path, Multiple Alternative Paths
- K-Dijkstra, One to Many Shortest Path
- Traveling Salesperson
- Turn Restriction Shortest Path (TRSP)
- Shortest Path Shooting Star

This permits us to implement not only point-to-point routing but also generate service areas, calculate optimal routes for reaching a specific set of destinations in the shortest possible time etc.
A link to the comprehensive formal specification of the OpenLS standard is included in the footnote references to this chapter and the realized methods from this interface are therefore omitted from the document.

4.6 Data management service

URL: http://portal.sdi4apps.eu/cgi-bin/layman
LayMan is a tool for publishing geodata. It uploads vector and raster files, imports rasters into the PostGIS database, creates metadata records in MIcKA metadata catalogue and publishes layers in GeoServer. LayMan Client provides web GUI that allows user on a single click to publish his or her geodata. LayMan Server does the work behind and offers a REST API that is used by the LayMan Client and that can be used by any other interested party. LayMan REST API version 2.0 is described below.
API - Work in Progress: This section describes LayMan REST API as it is being implemented in version 2.0 of LayMan. Please, before starting the implementation of the API, check with sredl@ccss.cz for possible updates.
REST Basics: In REST API, there are just four methods to be called: GET, POST, PUT and DELETE. In the REST API design, we design the *resources* to be manipulated with these four methods.

4.6.1 REST resources in LayMan

In LayMan, we care about three types of resources: **files, data** and **layers**. Files are private and belong to a single user. Data and layers typically belong to a group and can be shared. Metadata are handled as properties of data; standalone metadata records not belonging to any existing data are out of scope of LayMan.

4.6.2 Files

URL: `/layman/files/<user>`		
Method	*Functionality*	*Format*
GET	Get list of the files in the user's directory	JSON
POST	Upload new file	File data
PUT	X	

DELETE	X	

user - user name corresponding to the user directory

URL: `/layman/files/<user>/<file>`		
Method	*Functionality*	*Format*
GET	Download the file	File data
POST	Update existing file*	File data
PUT	Update existing file*	File data
DELETE	Delete the file	

user - user name corresponding to the user directory
file - file name

* We allow both to POST and PUT a single file to update. JavaScript clients inside a web browsers can have troubles PUTting the file.

URL: `/layman/files/<user>/<file>/details`		
Method	*Functionality*	*Format*
GET	Get the file details	JSON
POST	X	
PUT	X	
DELETE	X	

user - user name corresponding to the user directory
file - file name

Examples

`GET /layman/files/<user>` response:

```
[
	{
		date: "2013-09-26 17:22",
		mimetype: "application/x-qgis",
		name: "pest.shp",
		size: 12364
	},
	{
		date: "2013-04-30 08:32",
```

```
            mimetype: "application/x-qgis",
            name: "hlavni.shp",
            size: 6344
        }
]
```

`GET /layman/files/<user>/<file>/details` response:

```
{
        date: "2013-09-26 17:22",
        extent: [24.329387, 48.199749, 33.383884, 52.334394],
        features_count: 438,
        mimetype: "application/x-qgis",
        name: "pest.shp",
        prj: "EPSG:4326",
        size: 12364,
        type: "point"
}
```

4.6.3 Data

URL: **/layman/data/**		
Method	*Functionality*	*Format*
GET	Get the list of all the tables, views and data files from all the schemas and directiories user can access	JSON
POST	X	
PUT	X	
DELETE	X	

URL: **/layman/data/<group>**		
Method	*Functionality*	*Format*
GET	Get the list of all the tables, views and data files from the schema and the directory belonging to the given group	JSON
POST	Create new database table from vector file. For rasters, copy the file into the group directory.	
PUT	X	
DELETE	X	

group - user group name corresponding to a schema (for vectors) or a directory (for rasters)

URL: `/layman/data/<group>/{table|view|file}/<data>`		
Method	*Functionality*	*Format*
GET	Download data	
POST	X	
PUT	Update data	
DELETE	Delete data	

group - user group name corresponding to a schema (for vectors) or a directory (for rasters)
data - name of the table, view or file

URL: `/layman/metadata/<group>/{table|view|file}/<data>`		
Method	*Functionality*	*Format*
GET	Get metadata record corresponding to the referred data	JSON
POST	X	
PUT	Update metadata record corresponding to the referred data	JSON
DELETE	X	

group - user group name corresponding to a schema (for vectors) or a directory (for rasters)
data - name of the table, view or file

Examples

`GET /layman/data/` response:


```
[
	{
		datatype: "vector",
		name: "azov_sea_salinity1976_00",
		owner: "hsrs",
		roleTitle: "testing",
		schema: "testing",
		type: "table"
	},
	{
		datatype: "vector",
		name: "pest_00",
		owner: "hsrs",
		roleTitle: "testing",
		schema: "testing",
		type: "table"
	}
]
```

4.6.4 Layers

URL: `/layman/layers/`		
Method	*Functionality*	*Format*
GET	Get list of the layers from all the workspaces (groups) the user can access	JSON
POST	X	
PUT	X	
DELETE	X	

URL: `/layman/layers/<group>`		
Method	*Functionality*	*Format*
GET	Get list of the layers from the given workspace (group)	JSON
POST	Create new layer in the given workspace (group).	
PUT	X	
DELETE	X	

group - user group name corresponding to GeoServer workspace

URL: `/layman/layers/<group>/<layer>`		
Method	*Functionality*	*Format*
GET	Get the layer details	JSON
POST	X	
PUT	Update layer	
DELETE	Delete layer	

group - user group name corresponding to GeoServer workspace
layer - layer name

URL: `/layman/datalayers/<group>`		
Method	*Functionality*	*Format*
GET	X	

POST	Syntactic sugar for
	`POST /layman/data/<group>`
	`POST /layman/layers/<group>`	

PUT	X	
DELETE	X	

group - user group name corresponding to GeoServer workspace

Examples

`GET /layman/layers/` response:


```
[
	{
		datagroup: "testing",
		dataname: "azov_sea_salinity1976_00",
		layergroup: "testing",
		layername: "azov_sea_salinity1976_00",
		layertitle: "Azov Sea Salinity",
		owner: "hsrs",
		roleTitle: "testing",
		type: "vector"
	},
	{
		datagroup: "testing",
		dataname: "pest_00",
		layergroup: "testing",
		layername: "pest_00",
		layertitle: "Pesticidy",
		owner: "hsrs",
		roleTitle: "testing",
		type: "vector"
	}
]
```


4.7 Authentication service

URL: http://portal.sdi4apps.eu/cas-server/

The CAS server is Java servlet built on the Spring Framework whose primary responsibility is to authenticate users and grant access to CAS-enabled services, commonly called CAS clients, by issuing and validating tickets.[13]

4.7.1 Web API

HTTP Authentication - Get Login Ticket

[13] http://jasig.github.io/cas/4.0.x/planning/Architecture.html

URL: **/cas-server/login**		
Method	*Functionality*	*Format*
GET	Get a login ticket	HTML
POST	X	
PUT	X	
DELETE	X	

Example of the output

```
<html>
…
<div class="row btn-row">

<input type="hidden"
name="lt" value="LT-8-CPsDXqbfTs0ASnebytgrB3KxNs6pDm" />

<input type="hidden" name="execution" value="e1s1" />
<input type="hidden" name="_eventId" value="submit" />
…
</html>
```

HTTP Authentication - Get CAS Ticket

URL: **/cas-server/login**		
Method	*Functionality*	*Format*
GET	X	
POST	Send user credentials (username, password) and the login ticket	Cookie JSESSIONID
PUT	X	
DELETE	X	

Notes

JSESSIONID for the CAS authentication status.

The CAS ticket (within cookie) authenticates a CAS client against the CAS server.

Example of the output

```
Cookie CASTGC=TGT-2-yBSJsep0oAwprW0gygSg9b7uYISgBvlzW0fSPkuZRaIzYrKYiI-
portal.sdi4apps.eu for portal.sdi4apps.eu/cas-server/
```

4.7.2 REST API

Applications need to programmatically access CAS. Generally, proxying works for this. However, there are cases where an application needs to access a resource as itself, in which case proxying doesn't make any sense.

REST Authentication - Get a Ticket Granting Ticket Resource

URL: **/cas-server/v1/tickets**		
Method	*Functionality*	*Format*
GET	X	
POST	Send user credentials (username, password)	HTTP response
PUT	X	
DELETE	X	

Example of the output

Location: http://portal.sdi4apps.eu/cas-server/v1/tickets/{TGT id}

REST Authentication - Service Ticket

URL: **/cas-server/v1/tickets/**{TGT id}		
Method	*Functionality*	*Format*
GET	X	
POST	Service URL	HTTP response
PUT	X	
DELETE	Logout service	

Example of the output

ST-1-FFDFHDSJKHSDFJKSDHFJKRUEYREWUIFSD2132

4.7.3 CAS & OpenID

OpenID allows you to use an existing account to sign in to multiple websites, without needing to create new passwords. With OpenID, your password is only given to your identity provider, and that provider then confirms your identity to the websites you visit. Other than your provider, no website ever sees your password, so you don't need to worry about an unscrupulous or insecure website compromising your identity.[14]

[14] http://openid.net/

CAS can be configured as an OpenID provider. Since CAS version 4.0 OpenID support has to be enabled during installation process.

5 ADVANCED API SERVICES

The second class of OpenAPI services are “advanced services”. These are mostly services where no formal or de-facto standards exists for the service request/response formats. Typically, these services are tailored specifically for the SDI4Apps project.
Here, the emphasis lies in exploiting the possibilities of the technologies in addition to the fundamental requirements of performance, scalability and robustness.

5.1 Search service

Search service implements a REST API to retrieve specific records of data from datasets.
It performs a full-text search on the attribute values of the datasets, and returns the full REST representation of the objects fulfilling the criteria expressed in the search query.
The query scope can be further restricted specifying geometrical criteria:

- bounding box or geometry
- distance from a given point or geometry

Furthermore, the search can successively be refined by a faceted query mechanism, allowing the user to drill into search results specifying further criteria based on the values of some attribute.
The faceted search is a very powerful mechanism that let the user perform a custom navigation (the mechanism is also described as *faceted browsing*) upon clustered lists of items, or (as in our scenario) upon a list of results from a search query.
The navigation (*drill down* of the set of results) is based on one or more classifications of the search results: together with the results itself, the service returns to the requester a list of classifications (facets), each one based on a different attribute of the result objects.
For each facet is returned the set of values than can be selected to refine the search.
The typical handshake for a faceted browsing is described as follow:

1. The requester performs a full text search, specifying the search criteria (terms to be found by full text search) and a set of attributes of the result object that can be browsed by subsequent faceted browsing
2. The service response is filled up with all the objects matching the desired criteria, and with a list of values for every facet specified in the request. For every value, the service returns the number of results matching with that specific value. The values can be clustered, depending on the type of the attribute (see the price attribute on the example below)
3. The requester performs a new search, based on the same text criteria as before, and furthermore specifying one or more value of the facets to drill down. The new query can again ask for some facets to be returned. The facets can be the same as before (usually), or a new set.
4. The service returns a new result set that is a subset of the previous one. If facets were required again, the new list of attribute values for every facet is added to the response, and so on.

Let us go through a full example of a typical faceted search to better explain the handshake: a search query performed on the web site of an electronics store.

1. First of all, the requester looks for all the articles containing the word “brilliant” in some of the fields; furthermore, he ask for the manufacturer, price, category and color attributes to be faceted.
2. The service answers with a list of 30 articles, all matching the criteria “brilliant”, and:
    a. For the manufacturer facet, the list of values returned is: 10 Samsung, 9 Apple, 7 Canon, 3 Asus.

    b. By the price point of view, the facet result is: 8 results below €300, 10 results between €301 and €500, 12 over €500.
    c. The category facet result is: 6 TV, 18 PC, 3 mobile devices, 7 photo cameras.
    d. For the color facet, we have 14 white products and 8 black products (here we assume that for the remaining 8 the color is not specified)
3. The requester decides to refine the request, because he is looking for a Personal Computer. So he choose the value PC for the category facet, and submits a new search request, asking again for the same facet classification as before
4. The service returns a list of 18 PC matching the "brilliant" criteria, and:
    a. for the manufacturer facet, the list is: 9 Samsung, 6 Apple, 3 Asus
    b. for the price: 7 between 301 and €50, 11 over €500 (there are no more values below €300, so the value is not returned)
    c. the only category remained : PC
    d. by the color point of view: 8 black and 5 white

The benefits of a faceted search are obvious:

- clustering the results, we give the user a richer feedback: not only the results, but also a classification
- the user can navigate (drill down) through the results in many ways, following paths that are not predefined.
- we prevent dead-end browsing: user will not refine the query asking for red products (and obtaining zero results) because the facet result of the first query reports only black and white as possible values

5.2 Notes

From the service point of view, the full handshake is completely stateless. The second request (listed at the point 3.) is only logically related to the first one by the requester point of view, but for what concerns the service it could have been issued independently, with the same results.
All the requests described above can be performed using a single API. We expose a single **GET** method, using parameters to specify:

1. the dataset
2. words to (full text) search on
3. facets to be returned in results
4. facets values to refine the query

It must be noted that parameters specified at 3. and 4. have not to be related: we can refine the query using an attribute, and ask for facets based on different ones.

5.2.1 Query example

In the table below we describe the API and the role of each parameter, and then we provide an example of a request with the related JSON response.

URL: `/search?ds=<dataset>&q=<words>&geo=<wkt>&dist=<distance>&facet=<facet1>,<facet2>,...&fq=<attribute:value>,<attribute:value>`		
Parameters	*Format*	*Role*

ds	url-encoded purl	dataset identifier
q	words separated by blanks	words for full text search
facet	list of comma separated attribute identifier	facets return in the results
fq	list of couples (attribute identifier - value) separated by colon	attribute values to refine the query
geo	well-known text format	geometry constraint for the results. To fulfill the criteria, the object distance from the geometry must be less then *dist* parameter
distance	numeric values (meters)	the maximum distance from geometry described by *geo*. If the parameter is not specified, the distance must be 0 (object must intersect or be inside the geometry)

request: GET /search?ds=XXX&q=text&facet=attr1&fq=attr3:red

response:


```
{
	"result_count": 3,
	"results": [
		{
			"id": …,
			"geom": …,
			"attr1": "some text",
			"attr2": 12.25,
			"attr3": "red",
			...
		},
		{
			"id": …,
			"geom": …,
			"attr1": "some text",
			"attr2": 12.25,
			"attr3": "red",
			....
		},
		{
			"id": …,
			"geom": …,
			"attr1": "some other text",
			"attr2": 0,
			"attr3": "red",
			....
		}
	],
	"facets": {
		"attr1" : {
			"some text" : 2,
```

```
                    "some other text": 1,
                }
        }
}
```

5.3 Sensor data service

URL:
Proton
http://portal.sdi4apps.eu/ProtonOnWebServerAdmin/
http://portal.sdi4apps.eu/AuthoringTool/
Orion
http://portal.sdi4apps.eu:1026
Access to sensor data will be based on SensLog component. The SensLog is a tool for receiving, storing, analysing and publishing sensor data in various ways. The SensLog consumes sensor data in form of HTTP GET requests and stores them to PostGIS database. The SensLog can publish data in standardized form using OGC Sensor Observation Service 1.0.0, or using proprietary API version 1.0. SensLog contain only GUI with several functions to show data, because is supposed to be used in combination with other clients.
For specific issues can be used Orion Context Broker to provide current status of the sensor network. Orion Context Broker can provide system of notifications on changes in sensor networks and with combination with other Generic Enabler Complex Event Processing can provide system of alerts for defined situations.
Complex Event Processing Generic Enabler was designed and developed to react to situations rather to single events according to standard reactive applications. A situation is a condition that is based on a series of events that have occurred within a dynamic time window called a context. Situations include composite events (e.g., sequence), counting operators on events (e.g., aggregation) and absence operators. Proactive Technology Online (Proton) engine is a runtime tool that receives information on the occurrence of events from event producers (other components from platform or gateway from sensor network), detects situations, and reports the detected situations to external consumers (another component in the platform).

5.3.1 Sensor observation service

SOS provides access to observations from sensors and sensor systems in a standard way. The some way is suitable for any type of sensor systems. It could be remote sensing, in-situ, fixed and mobile sensors. SOS leverages the O&M specification for modelling observations and the TML and SensorML specifications for modelling sensors and sensor systems. SOS is primarily designed to provide access to observations. The SensLog is mainly focused on publication of observations in standard form for consumers of observations.

SOS contains mandatory operations:

- GetCapabilities – provides the means to access SOS service metadata.
- GetObservation – provides access to sensor observations and measurement data, a spatio-temporal query filtered by phenomena can be used
- DescribeSensor – retrieves detailed information about the sensors and processes generating those measurements.

All messages are XML documents with defined structure and elements by OGC standard document. Schemas of the documents are part of OGC Schemas Repository[15].

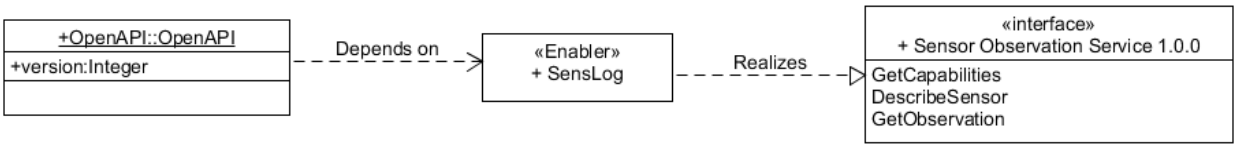

Figure 6: Realization of the SOS in the OpenAPI

Version 2.0 of OGC SOS for OpenAPI is under implementation work at this time.

5.3.2 SensorLOG API version 1.0

SensLog API version 1.0 provides methods to get metadata about sensor units, user groups and to get observations, alerts and positions of sensors units.

Data services

URL: `/DataService?Operation=<operationName>`		
Method	***Functionality***	***Format***
GET	**Provides detailed information about sensor units**	**JSON**

GetUnits

URL: `/DataService?Operation=GetUnits`		
Method	***Functionality***	***Format***
GET	**Provides detailed information about each units connected to login user. Response contains connected sensors, first and last time stamp of entered observation, last positions of unis.**	**JSON**

GetPositions

URL: `/DataService?Operation=GetPositions&user=<username>&limit=<limit>`		
Method	***Functionality***	***Format***
GET	**Request provides users specified number of last positions of all units in current group.**	**JSON**

[15] http://schemas.opengis.net/

Parameters	*Format*	*Role*
user	text	Identifier of user group
limit	Numeric value	Number of positions to receive

GetLastPosition

URL: `/DataService?Operation=GetLastPosition&user=<username>`		
Method	*Functionality*	*Format*
GET	Request provides user last positions of all units in specified user group.	JSON

Parameters	*Format*	*Role*
user	text	Identifier of user group

GetLastPositionWithStatus

URL: `/DataService?Operation=GetLastPositionWithStatus&user=<username>`		
Method	*Functionality*	*Format*
GET	Request provides user information about alert events and other attributes in addition to previous GetLastPosition request.	JSON

Parameters	*Format*	*Role*
user	text	Identifier of user group

GetTracks

URL: `/DataService?Operation=GetTracks&user=<username>&limit=<limit>`		
Method	*Functionality*	*Format*
GET	Request returns entered number of trajectory geometries of all moving units in entered group.	JSON

Parameters	*Format*	*Role*
user	text	Identifier of user group
limit	Numeric value	Number of positions to receive

GetRecentTracks

URL: /DataService?Operation=GetRecentTracks&user=<username>		
Method	*Functionality*	*Format*
GET	Request returns trajectory geometries of all moving units in entered group.	JSON

Parameters	*Format*	*Role*
user	text	Identifier of user group

Group services

GroupService provides detailed information about user groups. User groups can be arranged in hierarchy.

URL: /GroupService?Operation=<operationName>		
Method	*Functionality*	*Format*
GET	Provides detailed information about user groups.	JSON

GetGroups

URL: /GroupService?Operation=GetGroups&user=<username>		
Method	*Functionality*	*Format*
GET	Request returns information about entered group.	JSON

Parameters	*Format*	*Role*
user	text	Identifier of user group

GetSuperGroups

URL: /GroupService?Operation=GetSuperGroups&user=<username>		
Method	*Functionality*	*Format*
GET	Request returns information about superior group to entered group name.	JSON

Parameters	*Format*	*Role*
user	text	Identifier of user group

GetSubGroups

URL: **/GroupService?Operation=GetSuperGroups&group_id=<groupId>**		
Method	*Functionality*	*Format*
GET	Request returns information about subordinate groups to entered group.	JSON

Parameters	*Format*	*Role*
group_id	Numerical value	Identifier of group

Sensor service

SensorService provides information about sensors and enable to get measured or processed data.

URL: **/SensorService?Operation=<operationName>**		
Method	*Functionality*	*Format*
GET	Provides detailed information about sensors and provides methods to get sensor data.	JSON

GetSensors

URL: **/SensorService?Operation=GetSensors&unit_id=<unitId>**		
Method	*Functionality*	*Format*
GET	Request returns list of sensors connected to entered unit.	JSON

Parameters	*Format*	*Role*
unit_id	Numerical value	Identifier of unit

GetObservations

URL: **/SensorService?Operation=GetObservations& &unit_id=<unitId>&sensor_id=<sensorId>&from=<fromTime>&to=<toTime>&trunc=<trunc>**		
Method	*Functionality*	*Format*
GET	Request provides access to measured or processed observations for entered unit-sensor pair and entered time range. If user doesn't enter time range, servlet	JSON

	returns all available observations for entered unit-sensor pair. Another optional parameter is trunc that executes average of values for entered epoch (hour, day, week,...).	

Parameters	*Format*	*Role*
unit_id	Numerical value	Identifier of unit (mandatory)
sensor_id	Numerical value	Identifier of sensor (mandatory)
from	Timestamp (ISO 8601)	Time stamp of beginning time range (optional)
to	Timestamp (ISO 8601)	Time stamp of end time range (optional)
trunc	Text	Average epoch (optional)

Alert service

AlertService provides information about alerts events that arrived in sensor network. Methods allow user to get description of potential alerts connected to specific unit and list of arrived alert events including solving state.

URL: **`/AlertService?Operation=<operationName>`**		
Method	*Functionality*	*Format*
GET	Provides information about alerts events that arrived in sensor network.	JSON

GetAlerts

URL: **`/AlertService?Operation=GetAlerts&unit_id=<unitId>`**		
Method	*Functionality*	*Format*
GET	Request provides list of potential alerts for specified unit.	JSON

Parameters	*Format*	*Role*
unit_id	Numerical value	Identifier of unit

GetAlertEventsByTime

URL: **`/AlertService?Operation=`**

GetAlertEventsByTime&unit_id=<unitId>&from=<fromTime>&to=<toTime>		
Method	*Functionality*	*Format*
GET	Request provides list of arrived alert events for specified unit and specified time range.	JSON

Parameters	*Format*	*Role*
unit_id	Numerical value	Identifier of unit
from	Timestamp (ISO 8601)	Time stamp of beginning time range (optional)
to	Timestamp (ISO 8601)	Time stamp of end time range (optional)

5.3.3 Orion context broker operations

The Orion Context Broker (Orion) is an NGSI9/10 server implementation to manage context information and context information availability. In addition is possible to subscribe to context information and when some condition occurs a notification is sent. Orion runs as a backend service daemon. It doesn't have any Graphical User Interface (GUI) and it is accessed through its REST API.

Orion NGSI9 protocol interface

Orion NGSI9 operations can be classified into two groups. First, standard operations are directly derived from the Open Mobile Alliance[16](OMA) NGSI Specification. Second, convenience operations has been defined by the FI-WARE project. Orion NGSI-9 API supports XML and JSON as data serialization format.

Standard operations are following:

URL: /v1/registry/registerContext		
Method	*Functionality*	*Format*
POST	Request registers new entity in the context.	XML, JSON

URL: /v1/registry/discoverContextAvailability		
Method	*Functionality*	*Format*
POST	Request provides information about registered entity.	XML, JSON

URL: /v1/registry/subscribeContextAvailability

[16] http://openmobilealliance.org/

Method	*Functionality*	*Format*
POST	Request registers a subscription to send asynchronous notification to another server	XML, JSON

URL: **/v1/registry/updateContextAvailabilitySubscription**		
Method	*Functionality*	*Format*
POST	Request updates registered subscription	XML, JSON

URL: **/v1/registry/unsubscribeContextAvailability**		
Method	*Functionality*	*Format*
POST	Request cancels registered subscription	XML, JSON

Convenience operations are following:

URL: **/v1/registry/contextEntities/{EntityId*}**		
Method	*Functionality*	*Format*
GET	Request retrieves information on providers of any information about the context entity.	XML, JSON
POST	Request registers a provider of information about the entity.	XML, JSON

URL: **/v1/registry/contextEntities/{EntityId*}/attributes/{attributeName}**		
Method	*Functionality*	*Format*
GET	Request retrieves information on providers of the attribute value.	XML, JSON
POST	Request registers a provider of information about the attribute.	XML, JSON

URL: **/v1/registry/contextEntityTypes/{typeName}**		
Method	*Functionality*	*Format*
GET	Request retrieves information on providers of any information about context entities of the type.	XML, JSON
POST	Request registers a provider of about context entity of the type.	XML, JSON

URL: **/v1/registry/contextEntityTypes/{typeName}/attributes/{attributeName}**		
Method	*Functionality*	*Format*
GET	Request retrieves information on providers of values of this attribute of context entities of the type.	XML, JSON
POST	Request registers a provider of information about this attribute of context entities of the type	XML, JSON

URL: **/v1/registry/contextAvailabilitySubscriptions**		
Method	*Functionality*	*Format*
POST	Request creates a new availability subscription.	XML, JSON

URL: **/v1/registry/contextAvailabilitySubscriptions/{subscriptionId}**		
Method	*Functionality*	*Format*
PUT	Request updates subscription.	XML, JSON
DELETE	Request deletes subscription.	XML, JSON

More information about NGSI9 interface can be found on FI-WARE specification page[17].

Orion NGSI10 protocol interface

Orion supported NGSI10 operations can be divided into two groups. Standard operations are derived from OMA NGSI specification. Convenience operations have been defined by the FI-WARE project.
Standard operations are following:

URL: **/v1/registry/registerContext**		
Method	*Functionality*	*Format*
POST	Request registers new entity in the context.	XML, JSON

URL: **/v1/registry/updateContext**		
Method	*Functionality*	*Format*

[17] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_NGSI-9_Open_RESTful_API_Specification

POST	Request registers new entity or update existing entity in context.	XML, JSON

URL: **/v1/registry/queryContext**		
Method	*Functionality*	*Format*
POST	Request accesses entity information in context.	XML, JSON

URL: **/v1/registry/subscribeContext**		
Method	*Functionality*	*Format*
POST	Request registers a subscription to send asynchronous notification to another server.	XML, JSON

URL: **/v1/registry/updateContextSubscription**		
Method	*Functionality*	*Format*
POST	Request updates existing subscription.	XML, JSON

URL: **/v1/registry/unsubscribeContext**		
Method	*Functionality*	*Format*
POST	Request cancels existing subscription.	XML, JSON

Convenience operations are following:

URL: **/v1/contextEntities/{EntityID*}**		
Method	*Functionality*	*Format*
GET	Request retrieves information on providers of values of this attribute of context entities of the type.	XML, JSON
PUT	Request replaces a number of attribute values	XML, JSON
POST	Request registers a provider of information about this attribute of context entities of the type	XML, JSON
DELETE	Request deletes all entity information	XML, JSON

URL: **/v1/contextEntities/{EntityID*}/attributes/{attributeName}**

Method	*Functionality*	*Format*
GET	Request retrieves attribute value(s) and associated metadata.	XML, JSON
PUT	Request updates context attribute value.	XML, JSON
POST	Request appends context attribute value.	XML, JSON
DELETE	Request deletes all attribute values.	XML, JSON

URL: **`/v1/contextEntities/{EntityID*}/attributes/{attributeName}/{valueID}`**		
Method	*Functionality*	*Format*
GET	Request retrieves specific attribute value.	XML, JSON
PUT	Request replaces attribute value.	XML, JSON
DELETE	Request deletes attribute value.	XML, JSON

URL: **`/v1/contextEntityTypes/{typeName}`**		
Method	*Functionality*	*Format*
GET	Request retrieves all available information about all context entities having that entity type.	XML, JSON

URL: **`/v1/contextEntityTypes/{typeName}/attributes/{attributeName}`**		
Method	*Functionality*	*Format*
GET	Request retrieves all attribute values of the context entities of the specific entity type.	XML, JSON

URL: **`/v1/contextSubscriptions`**		
Method	*Functionality*	*Format*
POST	Request creates a new subscription.	XML, JSON

URL: **`/v1/contextSubscriptions/{subscriptionID}`**		
Method	*Functionality*	*Format*
PUT	Request updates subscription.	XML, JSON

DELETE	Request deletes subscription.	XML, JSON

URL: **/v1/contextEntities**		
Method	*Functionality*	*Format*
GET	Request gets all entities.	XML, JSON
POST	Request appends context attribute values, but EntityID is included in the body.	XML, JSON

URL: **/v1/contextTypes**		
Method	*Functionality*	*Format*
GET	Request gets all types.	XML, JSON

URL: **/v1/contextTypes/{entityType}**		
Method	*Functionality*	*Format*
GET	Request gets detail of a given type.	XML, JSON

More details about NGSI10 interface can be found on FI-WARE specification page[18].

5.3.4 Complex event processing RESTful API

Complex Event Processing (CEP) has three main interfaces: one for receiving raw events from event producers using a RESTful service, second for sending output events to event consumers using an output REST client adapter, and a third for administering the CEP engine state, and its definition repository. Proton consists of at least two application. Proton administration application (in further description as {CEP_Admin}) is responsible for the management of the definitions, event processing networks, repository (add, update, delete a definition) and the multiple Proton instances (update definition of an instance, retrieve an instance state and start\stop and instance). Proton instance (in further description as {CEP_instance}) is responsible for receiving events, detecting situations and sending derived events to consumer. More description with examples of messages can be found on FIWARE wiki[19].

Receiving events

URL: **/{CEP_instance}/rest/events**

[18] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_NGSI-10_Open_RESTful_API_Specification

[19] http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Complex_Event_Processing_Open_RESTful_API_Specification

Method	*Functionality*	*Format*
POST	Request receives events from events producer	tag-delimited, XML, JSON

Sending events

URL: **/application-name/consumer**		
Method	*Functionality*	*Format*
POST	Sends a derived event to a consumer in a push mode	tag-delimited, XML, JSON

Administrative methods

URL: **/{CEP_Admin}/resources/definitions**		
Method	*Functionality*	*Format*
GET	Retrieve all the existing definitions in the repository	
POST	Add a new definition	JSON

URL: **/{CEP_Admin}/resources/definitions/{definition_name}**		
Method	*Functionality*	*Format*
GET	Retrieve the complete definition in JSON format	
PUT	Replace content of an existing definition with new content	JSON
DELETE	Remove the definition from the repository	

URL: **/{CEP_Admin}/resources/instances/{CEP_instance}**		
Method	*Functionality*	*Format*
GET	Retrieve the status of an instance, the definition URI it is configured with and its state (stopped or started)	
PUT	Configuring the instance with a definition file or start\stop the instance	JSON

5.4 Feature synchronization service

With the growth of mobile Internet usage experienced today, the SDI4Apps platform must take into consideration the mixed connectivity scenario, i.e. where a user cannot guarantee persistent Internet connectivity during the time when he or she is using an application. It is not possible that all aspects of a Web Service API will work in offline mode, but a number of simple measures enables core functionality to be delivered. One of these measures is the feature synchronization service.

The platform already implements many services that return data in plain-text formats such as GeoJSON or KML, both of which are self-contained, portable and suitable for caching onto mobile devices using standard methods such as the HTML5 manifest file or similar.

Where the challenge arises is when we introduce the requirement that it shall be possible to edit and add to an existing vector layer while offline. The greater part of this challenge must be solved within the client, yet the client-side solution must necessarily rely on a server-side counterpart that permits data to be "checked back into" the original layer and potential concurrent editing conflicts to be resolved.

These methods are implemented by the feature synchronization service in the SDI4Apps OpenAPI. This service implements methods to check out layers for editing, check in edited layers, identify and resolve editing conflicts if any.

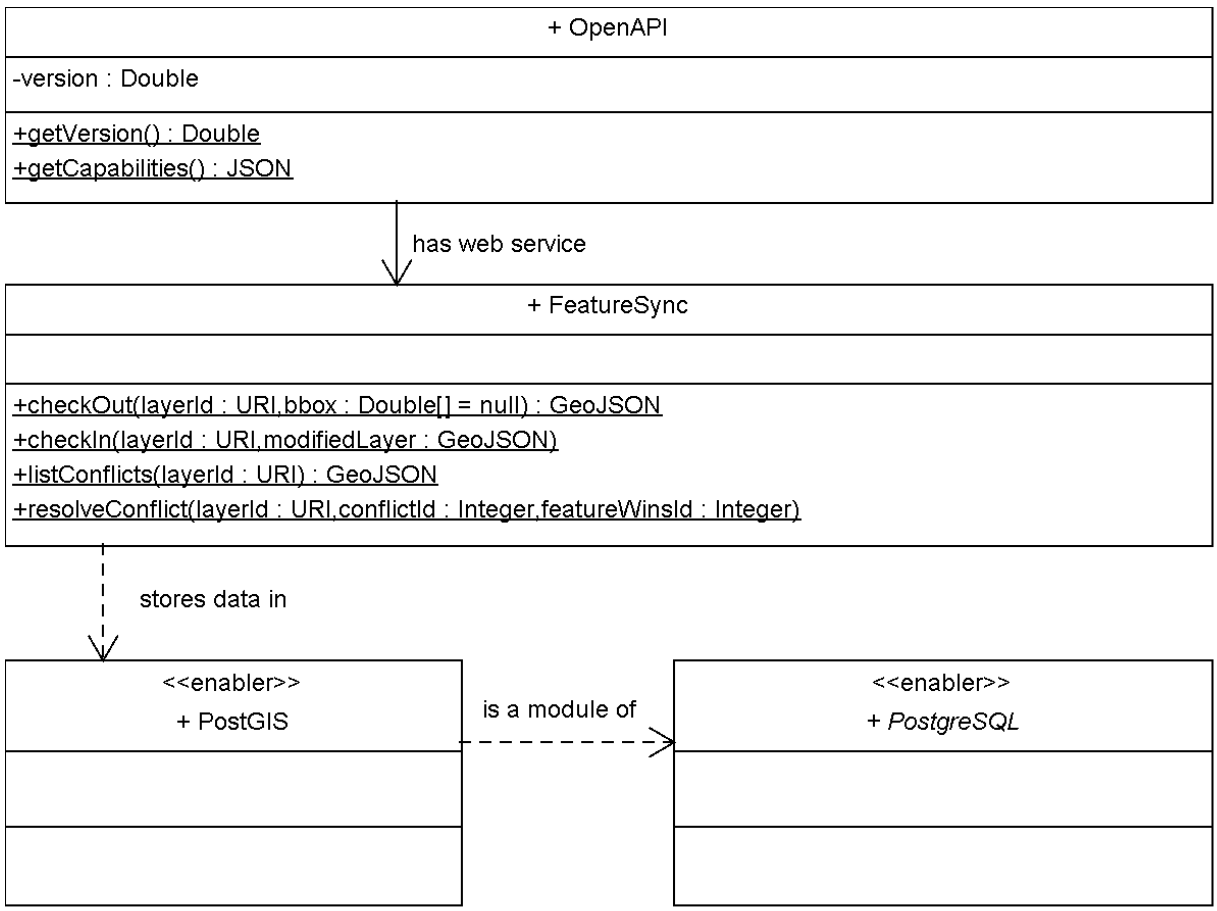

Figure 7: Realization of the feature synchronization service in the OpenAPI, supports offline editing capabilities in mobile applications

5.5 Basemap tile cache download service

Most contemporary spatial web applications implement slippy map user interfaces where base maps, and sometimes thematic overlays, are read as tiled images from a remote server. Instead of reloading the page requests for neighboring tiles are issued as the user pans and zooms the map.

This works reasonably well in online mode if the tiles are spread on a number of different servers through a content delivery network - or if the use is moderate. When offline, this naturally poses a challenge since it is no longer possible to issue new requests for new zoom levels or extents.

This can be handled in several ways. One is to rely on the built-in cache of the device web browser and/or the HTML5 manifest[20] file. However, this leads to the local device cache being flooded with a large number of image files files that pile up memory usage.

An alternative approach that allows for greater ease of management is to generate MBTiles SQLite databases where all the tiles for a specific area can be cached into a single portable file. This file can be put locally on the device where the application is to be used and accessed via a proxy script. This is more closely described under the corresponding module in D4.2 “Advanced Tools API design”.

From the server side perspective, this requires a service that enables users to select an area of interest and the desired minimum and maximum zoom levels - and generates an MBTiles file for download onto the device based on this information. This method is realized within the Web Service MBTilesDownloader in the SDI4Apps OpenAPI.

[20] "HTML5 Application Cache - W3Schools." 2012. 24 Mar. 2015 <http://www.w3schools.com/html/html5_app_cache.asp>

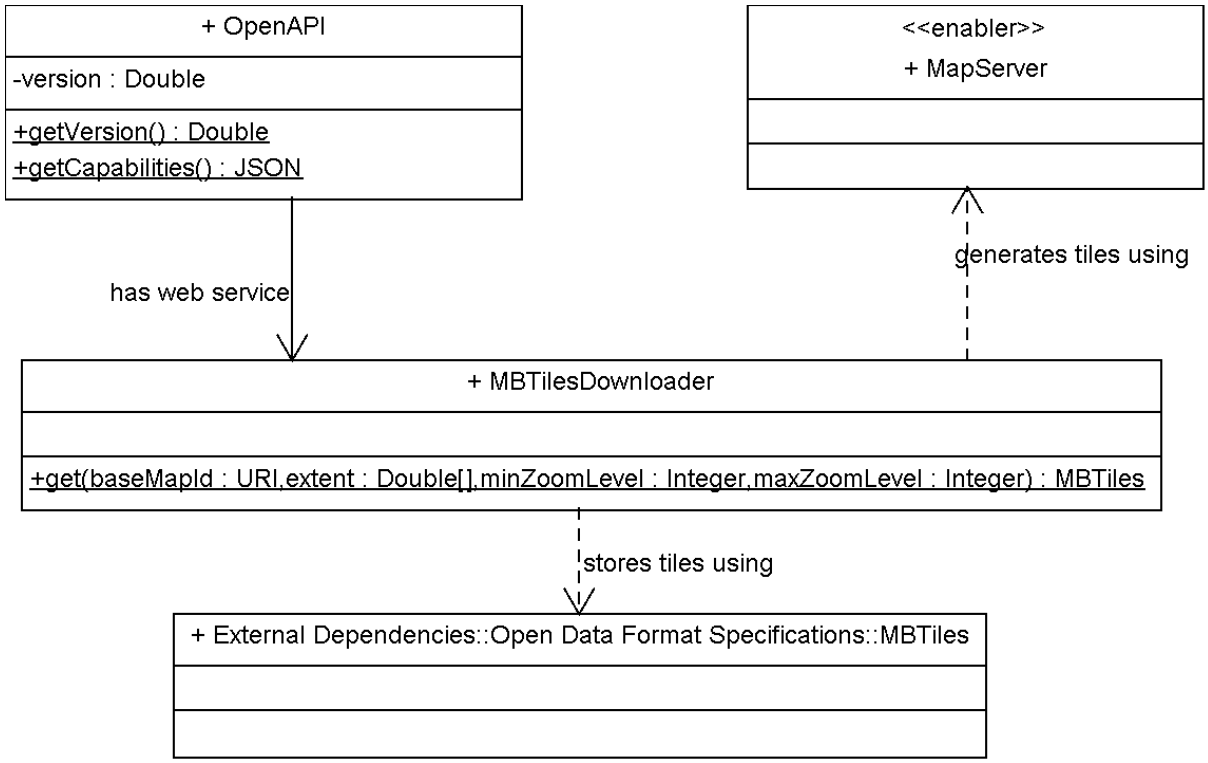

Figure 8: Realization of the feature synchronization service in the OpenAPI, supports offline editing capabilities in mobile applications

5.6 Basemap tile cache Download Service

Most contemporary spatial web applications implement slippy map user interfaces where base maps, and sometimes thematic overlays, are read as tiled images from a remote server. Instead of reloading the page requests for neighboring tiles are issued as the user pans and zooms the map.

This works reasonably well in online mode if the tiles are spread on a number of different servers through a content delivery network - or if the use is moderate. When offline, this naturally poses a challenge since it is no longer possible to issue new requests for new zoom levels or extents.

This can be handled in several ways. One is to rely on the built-in cache of the device web browser and/or the HTML5 manifest[21] file. However, this leads to the local device cache being flooded with a large number of image files files that pile up memory usage.

An alternative approach that allows for greater ease of management is to generate MBTiles SQLite databases where all the tiles for a specific area can be cached into a single portable file. This file can be put locally on the device where the application is to be used and accessed via a proxy script. This is more closely described under the corresponding module in D4.2 “Advanced Tools API design”.

[21] "HTML5 Application Cache - W3Schools." 2012. 24 Mar. 2015 <http://www.w3schools.com/html/html5_app_cache.asp>

From the server side perspective, this requires a service that enables users to select an area of interest and the desired minimum and maximum zoom levels - and generates an MBTiles file for download onto the device based on this information. This method is realized within the Web Service MBTilesDownloader in the SDI4AppsOpenAPI.

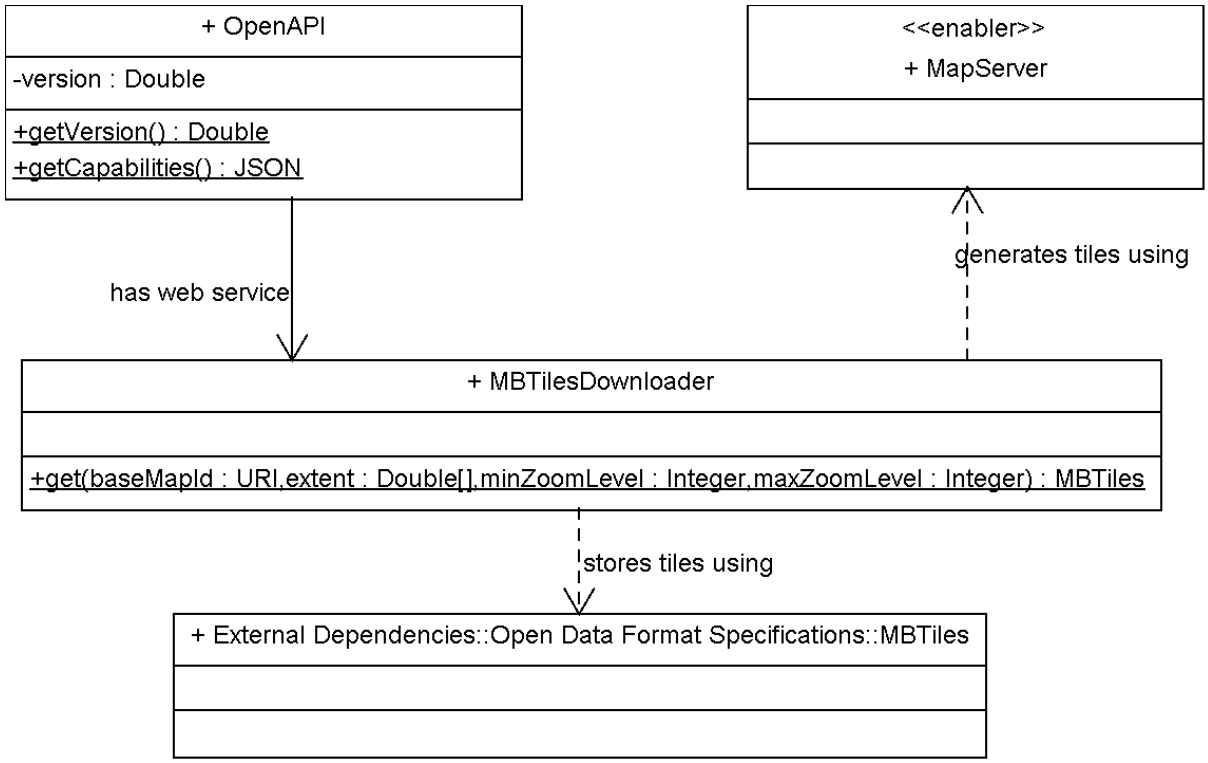

Figure 9: Realization of the custom tile cache download service in the OpenAPI, supports offline map browsing in mobile applications

The tiles will be generated by means of a WMS compliant map server, i.e. the SDI4Apps enablers MapServer or GeoServer and then stored into an SQLite database along with required metadata in accordance with the MBTiles[22] specification.

[22] "mapbox/mbtiles-spec · GitHub." 2011. 24 Mar. 2015 <https://github.com/mapbox/mbtiles-spec>

5.7 Extended storage service

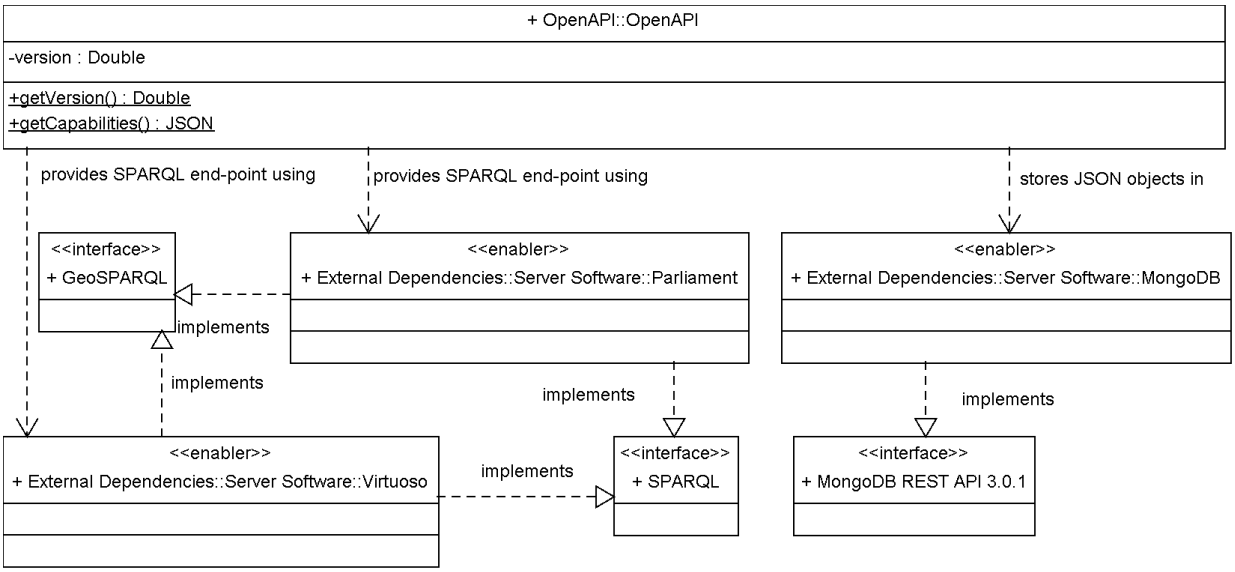

Figure 10: Realization of extended storage service in the OpenAPI, supports advanced queries and data analysis.

5.8 Analytics and modelling service

Data analytics is the process of examining raw data with the purpose of identifying conclusions that are evident from the data. When working with unknown data models, data analytics always require manual interpretation of the data. However, when working with unknown data but known data models, the process of data analytics can be automated into models that can be reused on different data adhering to the same structure.

Data analytics models are non-generic, i.e. they require knowledge of the data in terms of which attributes and data types can be found. The models to be implemented in the OpenAPI therefore must rely on one or more well-known data models.

The focus of SDI4Apps is on creating a versatile and reusable cloud based platform for geospatial applications and the work of narrowing down which underlying data models to work towards remains to be done.

However, a number of ground rules have been determined:

- The selected data sets must conform to common and widely used data models, e.g.
    - topological network
    - INSPIRE data models

While the specific models may not be described at this time, the execution architecture has been defined and is shown in the UML diagram below.

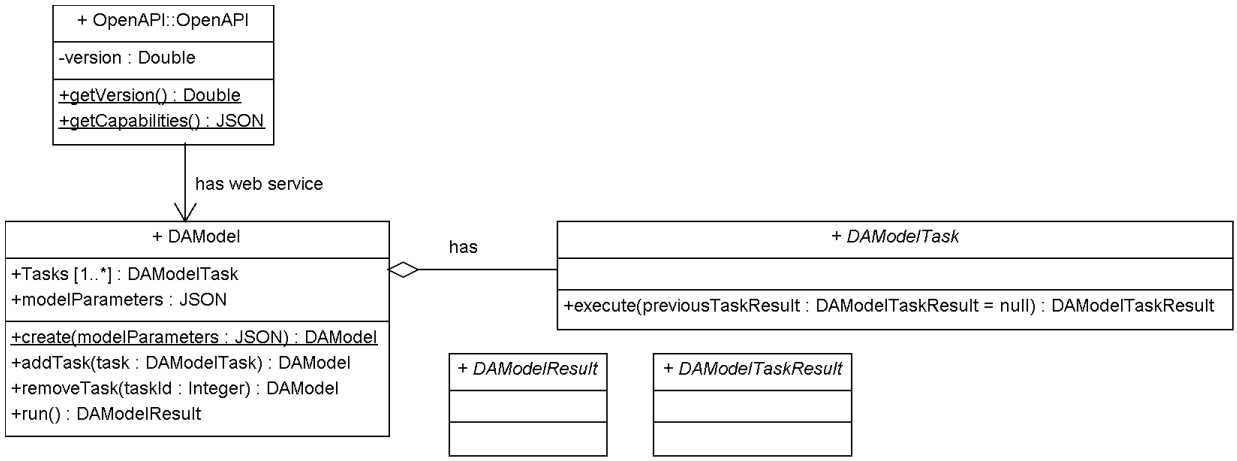

Figure 11: Realization of the analytics and modelling services in the OpenAPI

A model requires a number of user supplied parameters to run, these parameters can be of any data type and range from simple integers, strings and boolean values to complex objects such as GeoJSON feature collections.

A model consists of a number of tasks. The output of each task feeds into the next one. The first task must be runnable without other input than those supplied through the model parameters. A model task is one of the analysis web services that are available and can be invoked independently in the OpenAPI, including search, overlay, and routing.

The tasks are chained together in sequence by adding them to the models task queue. Once the last task has executed, the results of the model are returned to the client applications as an object.

Once the underlying datasets that the analytics and modelling service will work against have been determined, specific model classes that inherits from the abstract classes shown in the UML diagram above will be implemented. The same process will be repeated for each specific model.

5.9 Custom Data Services

The development period of the SDI4Apps OpenAPI and Advanced Tools API stretches over the two remaining years of the project. In order to meet the likely scenario that one or more calculations required to implement client services cannot be realized using open standards, protocols and well-known formats, the OpenAPI design specification includes an placeholder by the name “Custom Data Services”.

The specific content of these services will be described in detail as the work of implementing the first and second versions of the Advanced Tools API, D4.3.1 and D4.3.2. The design specifications will be updated into this document that will be annexed to the deliverable for the second release of the OpenAPI, D3.3.2.

While the detailed service specifications are not yet in place with regards to Custom Data Services, we do envisage that at least the following technologies and standards may be evaluated for potential inclusion:

- The GeoServices REST API 1.0 candidate standard by the Open Geospatial Consortium

6 CONCLUSION

Thanks to the methodology and analysis on the component and service carried out, it was possible defined a scalable API's architecture able to guarantee a complete interoperability with the other framework's components. Furthermore, the service and components that have been identified guarantee a complete customization of the service in order to develop "ad hoc" solution based on SDI4APPS domain.

REFERENCES

Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., Singh, A. SplitStream: High-bandwidth content distribution in a cooperative environment. In: Kaashoek, M.F., Stoica, I. eds. (2003) Peer-to-Peer Systems II. Springer, Heidelberg

Möller B., Löf S.,Mixing Service Oriented and High Level Architectures in Support of the GIG, Proceedings of the 2005 Spring Simulation Interoperability Workshop, April 2005

OGF, Open Cloud Computing Interface - Use cases and requirements for a Cloud API. 2010