SemSorGrid4Env

FP7-223913

Deliverable

D5.3v2

Programming patterns and development guidelines for Semantic Sensor Grids

Kevin R. Page, David C. De Roure, Alex J. Frazer, Kirk Martinez, Bart J. Nagel

University of Southampton

30/04/2010

Status: Final

Scheduled Delivery Date: 30/04/2010

Executive Summary

The web of Linked Data holds great potential for the creation of semantic applications that can combine self-describing structured data from many sources including sensor networks. Such applications build upon the success of an earlier generation of 'rapidly developed' applications that utilised RESTful APIs.

This deliverable details experience, best practice, and design patterns for developing high-level web-based APIs in support of semantic web applications and mashups for sensor grids.

Its main contributions are a proposal for combining Linked Data with RESTful application development summarised through a set of design principles; and the application of these design principles to Semantic Sensor Grids through the development of a High-Level API for Observations. These are supported by implementations of the High-Level API for Observations in software, and example semantic mashups that utilise the API.

Outline of changes since the previous version (v1)

Major changes since the previous version of the deliverable have been driven by the following developments:

- Consolidation of design principles for combining Linked Data and REST approaches in the context of sensor network data.
- Realisation of these design principles, and a full implementation of the High-Level API for Observations, in a redesigned "HLAPI" service (delivered as [D5.2v2]) and the experience drawn from this design and development process.

Alongside more iterative revisions to the document, these developments have led to more significant changes in the following sections:

- The addition of chapter 4.
- The restructuring and updating of chapter 5.
- The addition of section 6.3 and updating of prior sections.
- The addition of chapter 7.

Note on Sources and Original Contributions

The SemSorGrid4Env consortium is an inter-disciplinary team, and in order to make deliverables self-contained and comprehensible to all partners, some deliverables thus necessarily include state-of-the-art surveys and associated critical assessment. Where there is no advantage to recreating such materials from first principles, partners follow standard scientific practice and occasionally make use of their own pre-existing intellectual property in such sections. In the interests of transparency, we here identify the main sources of such pre-existing materials in this deliverable:

- Material previously published as: Page, K. R., De Roure, D.C., Martinez, K., Sadler, J. and Kit, O. (2009) *"Linked Sensor Data: RESTfully serving RDF and GML".* In proc. 2nd International Workshop on Semantic Sensor Networks, Washington DC, 2009.

- Material previously published as: Page, K. R., De Roure, D. C. and Martinez, K. (2011) "REST and Linked Data: a match made for domain driven development?". In proc. *2nd International Workshop on RESTful Design*, Hyderabad, India, 2011.

- Material to be published as: Kevin R. Page, Alex J. Frazer, David C. De Roure, and Kirk Martinez (2011) "*Semantic access to sensor observations through Web APIs*" (submitted to the 4th International Conference on GeoSensor Networks; the conference has since been cancelled and the paper will be resubmitted shortly to an alternative publication).

- Chapter 7 includes an updated mashup example extended from that found in D5.2v2.

- Several sections build upon experience and details (particularly diagrams) from prior deliverables including D1.3v2, D4.3v2, D5.1, D5.2v1, and D7.4v1.

Document Information

Contract Number	FP7-223913	**Acronym**	SemSorGrid4Env
Full title	SemSorGrid4Env: Semantic Sensor Grids for Rapid Application Development for Environmental Management		
Project URL	www.semsorgrid4env.eu		
Document URL	http://www.semsorgrid4env.eu/home.jsp?content=/sew/viewTerm&content=instance.jsp&_sew_var_name=instance&_sew_instance=D5.3+v2&_sew_instance_set=SemSorGrid4Env&_origin=%2Fhome.jsp		
EU Project officer	Antonios Barbas		

Deliverable	**Number**	5.3v2	**Name**	Programming patterns and development guidelines for Semantic Sensor Grids – Phase 2		
Task	**Number**	5.3	**Name**	Describe programming patterns and development guidelines		
Work package	**Number**	5				
Date of delivery	**Contractual**		30/04/2011	**Actual**	30/04/2011	
Code name				**Status**	draft ☐	final ☑
Nature	Prototype ☐ Report ☑ Specification ☐ Tool ☐ Other ☐					
Distribution Type	Public ☑ Restricted ☐ Consortium ☐					
Authoring Partner	University of Southampton					
QA Partner	Universidad Politécnica de Madrid					
Contact Person	Kevin R. Page					
	Email	krp@ecs.soton.ac.uk	**Phone**	+44 23 80594059	**Fax**	
Abstract (for dissemination)	This deliverable details experience, best practice, and design principles for developing high-level web-based APIs in support of semantic web applications and mashups for sensor grids.					
Keywords	Web Applications, Mashups, High-level API, REST, Linked Data, Sensor Networks					

Version log/Date	**Change**	**Author**
1.0 / 15/02/2011	**Import from D5.3v1**	**K. R. Page**
1.1 / 17/02/2011	**Restructuring of motivation and existing technology sections**	**A. Frazer, K. R. Page**
1.2 / 01/03/2011	**Added design principles section**	**K. R. Page**
1.3 / 15/03/2011	**Added implementation patterns section, restructured v1 implementations and added HLAPI engine**	**K. R. Page**
1.4 / 05/04/2011	**Added Use Cases and example semantic mashups**	**B. Nagel, K. Page**
1.5 / 14/04/2011	**Re-written HLAPI Design section**	**K. Page**
1.6 / 23/04/2011	**Revised remaining sections**	**K. R. Page**
1.7 / 24/04/2011	**Updated details in domain model section**	**K. R. Page, R. Garcia**
2.0 / 03/05/2011	**Final QA complete**	**K. R. Page, A. Frazer**

Project Information

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number FP7-223913. The Beneficiaries in this project are:

Partner	Acronym	Contact
Universidad Politécnica de Madrid (Coordinator)	UPM	Prof. Dr. Asunción Gómez-Pérez Facultad de Informática Departamento de Inteligencia Artificial Campus de Montegancedo, sn Boadilla del Monte 28660 Spain #e asun@fi.upm.es #t +34-91 336-7439, #f +34-91 352-4819
The University of Manchester	UNIMAN	Prof. Norman Paton Department of Computer Science The University of Manchester Oxford Road Manchester, M13 9PL, United Kingdom #enpaton@cs.man.ac.uk #t +44-161-275 6910, #f +44-161-275 62 04
National and Kapodistrian University of Athens	NKUA	Prof. ManolisKoubarakis University Campus, Ilissia Athina GR-15784 Greece #@ koubarak@di.uoa.gr #t+30 210 7275213, #f +30 210 7275214
University of Southampton	SOTON	Dr. Kirk Martinez University Road Southampton SO17 1BJ United Kingdom #@km@ecs.soton.ac.uk #t+44 23 80594491, #f +44 23 80595499
Deimos Space, S.L.	DMS	Mr. Agustín Izquierdo Ronda de Poniente 19, Edif. Fiteni VI, P 2, 2º Tres Cantos, Madrid – 28760 Spain #@agustin.izquierdo@deimos-space.com #t+34-91-8063450, #f +34-91-806-34-51
EMU Limited	EMU	Dr. BruceTomlinson Mill Court, The Sawmills, Durley number 1 Southampton, SO32 2EJ – United Kingdom #@bruce.tomlinson@emulimited.com #t+44 1489 860050, #f +44 1489 860051
TechIdeas Asesores Tecnológicos, S.L.	TI	Dr. Jesús E. Gabaldón C/ Marie Curie 8-14 08042 Barcelona, Spain #@ jesus.gabaldon@techideas.es #t+34.93.291.77.27, #f ++34.93.291.76.00

Table of Contents

Glossary

API	Application Programming Interface
CCO	Channel Coastal Observatory
CSS	Cascading Style Sheets
CSV	Comma-Separated Variables
FDD	Feature-driven Development
GIS	Geographic Information System
GML	Geography Markup Language
HATEOAS	Hypertext As The Engine Of Application State
HLAPI	High-Level API
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IQS	Integration Query Service
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
O&M	Observations and Measurements
OGC	Open Geospatial Consortium
OWL	Web Ontology Language
PHP	Hypertext Preprocessor
REST	Representational State Transfer
RDF	Resource Description Framework
RDFS	RDF Schema
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOS	Sensor Observation Service

SPARQL	SPARQL Protocol and RDF Query Language
SWE	Sensor Web Enablement
UDDI	Universal Description, Discovery and Integration
UI	User Interface
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WFS	Web Feature Service
WMS	Web Map Service
WP	Work Package
WSDL	Web Services Description Language
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
XP	Extreme Programming

1. Introduction

1.1. Scope

This document represents the D5.3v2 of Work Package 5 *High-level Application Programming Interfaces for Semantic Sensor Grids* within the EU project "*Semantic Sensor Grid Rapid Application Development for Environmental Management (SemSorGrid4Env)*".

This final version of the deliverable is the culmination of experience gained over the course of the project developing semantic APIs for sensor observations. From this experience we present principles for designing domain driven APIs to support development of lightweight web applications and mashups, and a design pattern that applies these principles to a High-Level API for Observations.

1.2. Document Structure

This document contains six main sections plus an introduction and summary.

The first, chapter 2, briefly introduces the motivations for High-Level APIs: Web 2.0, Web APIs and the agile development methodologies that drive many of these developments.

Chapter 3 gives more detail on two key technologies we use to realise High-Level APIs, REST and Linked Data, and existing alternative approaches within the GIS community and SemSorGrid4Env project.

Chapter 4 introduces Domain Driven Design, and in this context analyses the similarities and differences between REST and Linked Data. Based on this analysis a short set of design guidelines is given to improve provision of high-level APIs.

Chapter 5 applies these principles in a novel design for a semantic High-Level API for Observations from Sensor Grids that support both Linked Data and OGC derived representations through a RESTful interface. Chapter 6 describes three increasingly sophisticated and complete implementations of the API design, while chapter 7 details how application developers can use the API to create mashups.

2. Motivation for a High-level API

In this chapter we introduce the broader practices and technologies that underscore our motivation for development of a high-level API.

We begin with an overview of light-weight web applications and APIs, the agile development practices they support, and introduce their underlying principles. We consider how they might be applicable to users of a Semantic Sensor Grid.

In considering these users of an API, we distinguish between *domain users* and *domain developers*. In the example of the Flood Use case (WP7) the domain users are those who use the web applications and mashups, such as the emergency planning and decision support web applications described in [D7.1v2]. Domain developers are users of the high-level API: those who build the web applications and mashups using the high-level API, which will then be used by the domain users.

2.1. Web 2.0 and the Interactive Web

The interconnected nature of Web 2.0 means that more and more applications and services rely on bringing together two or more different services or data sources. But for this to work efficiently, there need to be standard mechanisms for interoperability.

Various "heavy-weight" technologies exist to enable service description (WSDL), discovery (UDDI) and communication between services (SOAP). While these technologies support a wide range of features, the architectural underpinnings needed to include them in a system are often complex, and are not especially well-suited to the rapid, iterative update cycle of a typical Web application (see Supporting agile development models, section 2.3).

In contrast to these more verbose technologies, "light-weight" technologies such as RESTful resources and Web APIs allow resources and services to be included in a Web "mashup" with much lower architectural overheads. Because of this, changes can be incorporated into a Web application much more quickly, making these technologies more suited to the typical mashup life-cycle. [Ben2008]

Within the SemSorGrid4Env project, the Web Applications (e.g. for the flood use case [D7.1v2], [D7.4] primarily present a User Interface to *domain users*, so they can access, utilise, and manipulate, sensors and associated data provided by systems employing the SemsorGrid4Env architecture.

Through the high-level API we must also support *domain developers* such that they can easily, quickly, and simply, create Web Applications and mashups. In the following sections we outline the lightweight APIs and agile development methods that support domain developers.

2.2. Web APIs

An Application Programming Interface (API) is a defined set of functions made available by one system, to allow other applications to communicate with it. Typical

examples include operating system APIs to allow desktop applications to interface with hardware, or scripting language APIs to interface with operating system calls — all through a standard, well-defined set of functions. [Pro2010]

In a similar way, online systems offer APIs to allow other systems to interact with their functionality. As these interactions occur most naturally across Web architecture, these APIs are known as "Web APIs". ProgrammableWeb.com provides a catalogue of over two thousand Web APIs, grouped by category, interface style and data format.[1]

As an example, a simple Web API listed on ProgrammableWeb.com is "Yahoo Weather"[2]. As a provider of weather data (such as temperature, humidity, wind speed and direction), Yahoo have exposed this data to developers via an HTTP interface. This particular API requires the client to make an HTTP "GET" request to a specified URL, with two additional parameters: a location identifier, and a flag to say which units the measurements should be returned in. As a response to this request, the service returns XML content describing the various weather details for the area requested, given in the units specified. Now that the client has this data, it can be displayed or manipulated in whatever way the developer chooses.

Figure 2-1: An example request to/response from the Yahoo! Weather RSS API

[1] Web API catalogue, ProgrammableWeb.com – http://www.programmableweb.com/apis

[2] Yahoo! Weather, Yahoo Developer Network – http://developer.yahoo.com/weather

Slightly more complex is the weather API from WorldWeatherOnline.com[3]. Again, weather information is requested via an HTTP "GET" request, although this time, there are a few more options. The client can specify the format in which the data is returned this time – either XML, CSV or JSON. This allows the provider to cater for a wider range of developers, all from a single set of data. The developer simply adds the appropriate flag to their HTTP request, and the provider works out how to return the data in the correct format. Similarly, the client can specify location using either latitude/longitude, or various regional postal codes. Again, the client sets the correct flag, and the API provider works out which data needs to be returned.

Even more complex still, is the NOAA Weather Service[4]. Unlike the previous examples, this system implements a SOAP interface, rather than a RESTful HTTP interface.

To interact with a SOAP interface, the client's code will need to create function calls to the service's remote functions, as described in the service's WSDL file. This request is packaged into a SOAP "envelope", along with the appropriate parameters, and sent to the service endpoint. Once received, the service unpacks the message, and runs the appropriate service function, before packing the response in another envelope and returning it to the client. In this way, SOAP-based APIs work more like traditional functional programming, with clients passing parameters to functions, and receiving the corresponding return values.

In the case of the NOAA Weather Service, there are separate functions to return weather data for a given latitude and longitude, as well as for returning a latitude and longitude based on other location data. This allows the client to request a latitude and longitude by passing a postcode, city name or other identifying feature to the appropriate function, then passing the resulting latitude/longitude to the weather data function, to receive weather data for that area. The client can also pass additional parameters to the weather data function, to specify the time period which the data should cover.

As these online systems become more comprehensive in the features they offer, the set of functions required to interact with them becomes more complex. In this way, the Web itself can be thought of as a "platform", much like an operating system [Pro2010]. Where the operating system is a platform on which to build desktop or server-based applications, the Web becomes a platform for interconnected, online systems. Like the operating system APIs, Web APIs offer a standard set of functions to interface with this "platform", exposing its functionality to developers in a standard, well-defined way.

[3] WorldWeatherOnline - http://www.worldweatheronline.com/free-weather-feed.aspx

[4] NOAA Weather Service API - http://www.programmableweb.com/api/noaa-weather-service

2.3. Supporting agile development models

Modern Web applications and mashups are typically built by reusing as much existing functionality and data as possible, developing new code to tie current systems together in new and interesting ways. Because of this heavy reliance on re-use, mashups can be developed much more quickly and easily than traditional software systems.

This kind of quick development cycle lends itself well to Agile software development practices. The most common of which are “Extreme Programming”, “Scrum” and “Feature-driven development”.

Extreme Programming (XP) relies on short development cycles and frequent, stable releases, in order to cater to new user requirements as soon as possible [Bec2000]. Unlike many traditional software models where user requirements are established at the beginning of the project, XP uses its rapid, short development cycles to introduce new requirements throughout the project's life. After gathering the first set of requirements, a combination of paired programming and unit testing will follow, followed by acceptance testing with the user. Once accepted, the next set of requirements is discussed, and a new development cycle begins.

This particular style suits Web application development well, as it inherently accounts for the changing requirements inevitable in the ever-changing Web. These changing requirements will come not only from users — whose expectations will changed based on competing products, new technologies or shifts in social dynamics — but also in changing technologies. As Web apps and mashups rely on supporting frameworks, APIs and data sources, any change in one of these systems means a corresponding change in the new system. By embracing this inevitable change, XP ensures that a stable system will always be released relatively frequently, without becoming mired in platform migrations and feature requests.

Scrum takes its name from the game of Rugby, where a team of players — each with their own distinct roles — all work together in pursuit of a common goal, “passing the ball back and forth” [Sch2004]. Like XP, Scrum aims to complete a “shippable” product increment at the end of a short development period (known as a “sprint”). However, unlike XP, these sprints are of a fixed length. Where XP will plan features for a particular iteration such that the release cycle is still relatively short, it is still primarily concerned with the implementation of the features as its primary goal. In Scrum, the time-based deadlines of more traditional software development are used, albeit estimated based on the features to be implemented.

In addition, the feature requirements for the overall system are determined and prioritised at the beginning of the project, with one or more selected for implementation at the beginning of each sprint. As a result, changes can be made to the list of overall project requirements, but once a sprint has begun, the requirements selected for implementation during this cycle remain fixed.

By retaining XP's rapid development cycle, Scrum can still react relatively well to the inevitable changes of Web development. However, its fixed-time sprints have been criticised, as they are easily disturbed by unexpected programming errors, or by changing user requirements which directly affect the requirements of the current sprint.

The ability to re-evaluate these issues at regular intervals, however, still makes Scrum more suited to the changing environment of the Web than most traditional software development models.

Like XP and Scrum **Feature-driven development** (FDD) relies on short development iterations, and uses a list of desired features to determine a given iteration's tasks [Pal2002]. After an initial planning stage, the required featured are determined, and assigned to classes. Once this stage is complete, chief programmers begin a series of two-week iterations by selecting one or more individual features, generating the appropriate design diagrams and documentation, and finally implementing the features in code. At the end of each iteration the code is tested, inspected and integrated into the main build.

Like the previous two methods, this rapid cycle of development ensures that new features make it into the published build as quickly as possible. However, because FDD still relies on many of the tenets of traditional software design (verbose UML diagrams, regular code inspections), a comparable feature would is likely to take longer to develop from start to finish via FDD, than via XP or Scrum. In addition, because requirements elicitation is only done at the beginning of the project, FDD is unsuited to situations where user requirements change throughout development.

Agile Method	**Release Cycle**	**Reaction to changes in:**	
		User Requirements	**Supporting APIs**
Extreme Programming	Frequent, varies	As required	Release delayed
Scrum	Frequent, fixed	Per iteration	Set deadlines affected
Feature-Driven Development	Frequent, fixed	Per Project	Set deadlines affected

Table 1: A comparison of the strengths and weaknesses of different Agile software development models, with respect to Web application development

Table 1 highlights the relative strengths and weaknesses of the various approaches to Web application design. While all produce rapid, iterative releases of usable code, they each handle changes in user requirements and supporting APIs differently.

3. Enabling technologies and existing domain approaches

In capturing programming patterns for high-level APIs for Semantic Sensor Grids we must also understand the principles that underpin successful Web APIs, and how technologies already employed in domains associated with sensor grids might be applicable.

In the first two sections of this chapter we describe two approaches fundamentally built upon and with the architecture of the Web: Representational State Transfer (REST), and the Linked Data principles as applied to the Semantic Web.

We then give an overview of the Sensor Web technologies developed by the widely accepted Open Geospatial Consortium, and position high-level APIs within the SemSorGrid4Env architecture.

3.1. REST and Resource Orientated Architectures

Representational State Transfer (REST) is a set of design principles which have been popularly and successfully adopted in many (RESTful) web services, and is typically framed as an alternative to ‘heavyweight' web services, including as the WS-* family. The key principle of REST is the use of resources for specific things that we wish to reference, and the referencing of these resources using URIs. Representations of these resources — encoded in a particular format — are then accessed through the URI, usually using HTTP.

3.1.1. Design Principles

REST, as an architectural style, is an effort to bring together the set of design principles enshrined through implementation in Web Architecture. Primarily, REST aims to capture the features of the Web which allow it to scale so successfully, that is:

- Everything is a resource which is addressable
- Resources have multiple representations
- Relationships between resources are expressed through hyperlinks
- All resources share a common interface with a limited set of operations
- Client server communication is stateless

REST is not, however, defined in terms of, nor limited to, the web (though HTTP meets the REST criteria) and while there have been attempts to clarify the application of REST to web services through definition of a Resource Oriented Architecture the term is still often loosely, sometimes incorrectly, applied.

In order to maintain these principles, certain architectural constraints were applied [Fie2000]. These constraints include the following:

- The **client-server** constraint is used to separate the concerns of the parties involved. Typically, this involves separating the user interface and business logic processes from the data storage concern. This offers two major benefits:

the user interface can be ported to different platforms while using the same data, and the data sources can scale more easily by keeping the server architecture relatively simple. In a Web environment, this also allows both client and server elements to develop separately, suiting the multiple organisational domains of the Internet more appropriately.

- A **stateless** system insists that any request from the client to the server contains all the necessary information to make the request, without relying on any context stored on the server. In a REST-based system, the state is made implicit by the status of any current HTTP requests. If there are no outstanding HTTP requests, the system is "at rest" in a single "application state". By initiating an HTTP request, the system is implicitly changing state – it enters a "transition state" between application states. Once these requests resolve, the system "at rest" again, in whatever new application state is associated with the most recent request.
  By making a system stateless, several emergent benefits arise. Visibility is improved, as a single request can describe that request's entire nature. Because no state is explicitly stored on the server, calls to a REST service are idempotent. This allows failed requests to be safely made again and again, without adversely affecting the server's internal state, improving user-perceived reliability and preserving the server's data integrity. Scalability is also improved, as the server has no need to manage resources between individual requests.

- One disadvantage is that network performance may drop, as almost identical request data will be sent with every request. This can be mitigated somewhat by making the system **cacheable**, i.e. by explicitly stating whether a returned resource can be reused for equivalent future requests. In this way, some interactions will we completely avoided, improving scalability and reducing latency. However, reliability decreases if the cached data is allowed to become different from the comparable data stored on the server at the time of request.

- The **uniform interface** of REST is the feature which distinguishes it most clearly from other Web services. By making the interface generalised, the system architecture can be simplified, and its interactions are made more visible. The main disadvantage is that efficiency is reduced, as information is transferred in a standardised, generic form, rather than on specific to the application. In this way, REST is optimised for the generic data transfer of the Web, but is less than optimal for any other type of interaction.

3.1.2. Architectural Elements

As REST is an abstraction of the Web, it ignores details such as component implementation and protocol syntax. Instead, it focuses on the roles of its components, how they interact with other components, and the identification of significant data elements [Fie2000].

REST components communicate by transmitting a representation of a resource in a format specified by the requesting system. The representation can be the same as the

original resource, or could be derived from it — whichever the case, the client receives only the representation, and its construction remains hidden.

The *resource* is REST's key abstraction of information, and can include any concept that could be the target of a hypertext reference. A resource is a temporally varying, conceptual mapping to a set of entities. This set could be empty — identifying an as yet unrealised concept — or could contain resource representations or identifiers. While two different resources could reference the same entity at one point in time, there is no requirement for them to always do so. For example, a code tarball release with version number "1.6" could be represented by a resource "tarball release 1.6" and another called "latest". However, when version number "1.7" is released, the "latest" resource would most likely now point to this entity instead.

This method has several advantages. It allows the representation of a resource to be bound at the last minute, enabling content negotiation based on the request. It also allows the client to reference a concept instead of a specific representation, so that no links need to be changed whenever the underlying resource itself changes.

One of the main tenets of REST is the primacy of resources that are uniquely identified by opaque URIs – in order to avoid coupling between clients and servers, no assumptions must be made about the structure of the URI [Ala2010]. REST limits the operations exposed by a web service to a small, well-defined, standard, set [Ric2007]. For HTTP, these are:

- GET – to return a list of URIs representing a collection's members, or to retrieve a representation of a member resource itself
- POST – to add a new member URI to an existing collection, or to turn an existing member resource into a collection by inserting a new member URI into it
- PUT – to update, replace, or create a new collection or collection member, depending on whether it exists already or not
- DELETE – delete a collection or member of a collection completely

This contrasts with a potentially expansive set of operators (for RPC style web services) or message contracts (for SOA style web services). It also means HTTP is retained as an application layer protocol as per its originally design, rather than being re-purposed as a transport layer, e.g. for SOAP; this brings both benefits (e.g. compatibility and scalability with standard web infrastructure) and further constraints (e.g. idempotence becomes desirable across operations to cope with network unreliability).

This constrained set of operations leads to a design process focused on correctly identifying the resources that should be exposed for a service and their representations; while the interface to the resources is simple, the number of resources – every piece of information that could be served - is likely to be many, with a URI for each. Since an application client cannot possibly know of every URI in existence it is important that resources hyperlink to other resources so a client application can navigate around them.

A Resource Oriented Architecture also requires statelessness – that each HTTP operation is totally separate from any other. As such, any state the service has must also

be exposed as a resource; an application client enters that state by accessing the URI for that resource; to enter another state a will use another URI.

Any application state a service requires to provide a representation of a resource must be completely contained within the request to the server (where the application is the client software processing and modifying the resource representations returned by the service). Transitions in application state are made by moving - "navigating" in a web sense – to alternate resources provided as URI links in the representation of a resource provided returned by the server.

3.2. The Semantic Web and Linked Data

The term "Semantic Web" describes methods and technologies to allow machines to understand the meaning of data on the Web. The addition of machine-readable metadata to existing content would allow automated agents to access the Web more intelligently, performing relevant tasks and locating related information without explicit user input. While not formally defined, the term is generally used to describe the technologies used to implement it [Ger2006], including:

- Resource Description Framework (RDF) [RDF1999] – a language for expressing data models, referring to objects and their relationships. This is typically expressed as a subject-predicate-object "triple", e.g. "wave height", "is-a-type-of", "metocean measurement"
- RDF Schema [RDFS2004] – extends RDF, allowing the properties and classes of RDF-based resources to be described, with semantics of generalised hierarchies.
- Web Ontology Language (OWL) [Lac2005] – adds additional vocabulary for describing these properties and classes, including the relationship between classes, cardinality, equality, characteristics of properties and enumerated classes.
- SPARQL [Cox2007] – a query language for Semantic Web data sources

The concept of Linked Data centres on using the Web to create typed links between different data sources. Technically, the term refers to data published on the Web in a machine-readable way, with explicitly defined meaning, that is linked to other external data sets and has the potential to be linked to itself from other data sets. Where the Web is based around HTML documents linked by untyped hyperlinks, Linked Data is based on Resource Description Framework (RDF) documents. These documents described the typed links which link the document to other arbitrary data sources. Initially, Linked Data was only concerned with data itself, with URIs being used primarily as unique identifiers. However, when combined with the hypertext Web, these URIs become equally important in retrieving the data across the network.

Berners-Lee outlined a set of "rules" for publishing data on the Web, such that it meets the goals of Linked Data:

- Use URIs as names for things
- Use HTTP URIs, so people can look up these names

- When someone looks up a URI, provide useful information using the standards
- Include links to other URIs, so they can discover more things

These "rules" have become known as the "Linked Data Principles" [Biz2009], and provide simple guidelines for publishing connected data on the Web, in accordance with its architecture and standards.

Linked Data relies on two fundamental Web technologies: URIs and HTTP. Entities identified by URIs can be located by simply dereferencing the URI over HTTP. Thus, HTTP provides a simple mechanism for retrieving the resources themselves, or descriptions of resources which cannot be sent. RDF provides a generic, graph-based data model with which to structure and link the data which describes things. RDF encodes data in the form subject-predicate-object. These "triples" take a URI as the subject and object, with a predicate used to describe how one relates to the other. In this way, RDF links can be created, with the subject and object each referencing the namespace of a different data set. Dereferencing these namespaces will result in the graph described by that namespace, i.e. the graph describing the entity represented by the URI, or the relationship represented by the predicate.

By combining the features of HTTP URIs, HTTP as a retrieval mechanism and RDF as a data model, Linked Data builds directly on the Web's generalised architecture. As such, the Linked Data web can be seen as an additional Web layer, with many similar properties:

- Generic, and can contain any type of data
- Anyone can publish to the Web of Data
- There is no constraint on data publishers to choose a specific vocabulary
- Entities are connected by links, creating a graph of linked data sources, and enabling the discovery of new data sources

From an application development perspective, this means:

- Data is separated from formatting and presentation
- Data is self-describing, as the describing vocabulary can be dereferenced via its URI
- HTTP as a transport mechanism and RDF as a data model are much simpler than WS-*-based APIs, with heterogenous data models and interfaces
- The Web of Data is open, so data sources can be discovered at run-time via RDF links, rather than being hard coded from the start

While the use of URIs is common throughout the Semantic Web - not least as the basic element of RDF - the requirement to use HTTP URIs sets Linked Data deployment apart. It is a departure from the use of URIs purely as unique identifiers within the graph; in Linked Data they are also a means of retrieving parts of the graph relevant to that resource - the URIs can be dereferenced.

This dual use of HTTP URIs does not, however, remove the need to distinguish between the two uses: a web client must be able to tell the difference between a URI representing the person Tim Berners-Lee (a non-information resource) and a URI

providing information about Tim Berners-Lee (an information resource); even if, in the linked data web, we dereference the former to retrieve the latter.

A web server communicates this distinction to the client through a combination of the HTTP "303 See Other redirection code (referred to as the httpRange-14 solution) and content negotiation, i.e. returning different representations according to the HTTP Accept header set by the client [Sau2008].

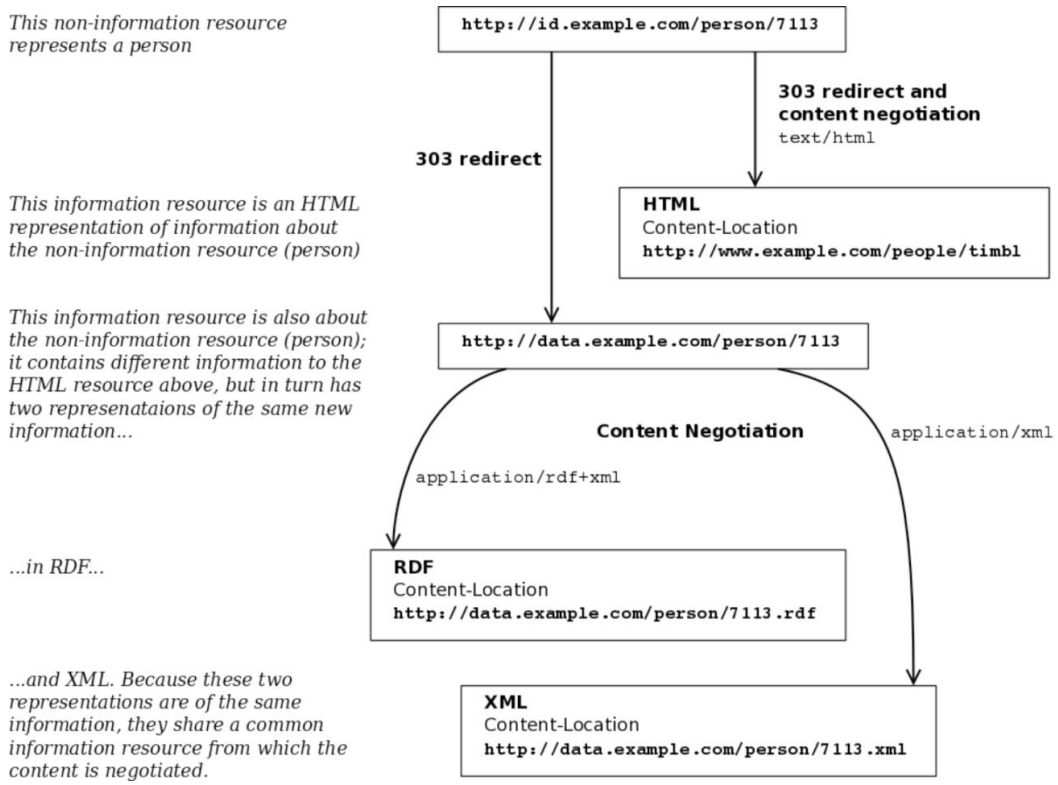

Figure 3-1: An example of content negotiation, based on the MIME type specified by the client (from [Pag2009])

There are two general cases for this solution:

1. If an information resource describing the non-information resource has multiple representations (e.g. in RDF and HTML) of the same information then the web server should first redirect the client (via a 303 response) to the intermediate information resource URI (indicating the move from a non-information resource to an information resource), and then use content negotiation to return the appropriate representation. The Content-Location header should be used to confirm the URI of this representation.
2. If the information resources describing the non-information resource contain different information depending on which representation is requested (e.g. the

RDF representation contains different information to the HTML, not just a different representation), then the web server should redirect the client (via a 303 response) directly to the information resource appropriate to the requested content type, without an intermediate common information resource. The Content-Location header should be used to confirm the URI of the returned representation.

Finally, it is noted that the use of resolvable (HTTP) URIs does not imply the encoding of semantics within a URI, and that the syntax used by a web server when returning a resource should not be interpreted as having such meaning. Apparent abstractions of the URI API (e.g. `http://example.com/<element>/<type>/<time>`) cannot, and should not, be provided - certainly when describing the interaction with a client application. Use and manipulation of such an abstraction might provide a useful shortcut for developers looking to manually locate and trial resources; the use of such 'friendly' URIs that may encourage this misuse are not without merit when providing manageable endpoints for developers and end-users; but a linked data client should primarily access new resources via the links asserted within the (RDF) graph.

3.3. OGC Standards and Sensor Web Enablement

Standardised data encodings and service definitions from the Open Geospatial Consortium (OGC) are widely adopted across industry. Earlier standards introduced services to directly support Geospatial Information Systems (GIS), while more recent efforts have resulted in services defined as part of Sensor Web Enablement (SWE).

The core Open Geospatial Consortium (OGC) encoding is GML, which is an XML schema derived language in which several GML Application Schema are defined. In order to expose an application's data using GML, an XML schema must be created specific to the application domain. This schema describes the relevant data objects, which applications that implement the schema must expose. For example, an application for flood defence monitoring may define wave heights, coastal defence types and heights, tide heights and population densities in its schema. These data objects will in turn reference the primitive data objects defined by GML. These primitive types include geometries, coordinates, units of measurement and directions, as well as concepts such as "features" and "observations".

GML is a particularly interesting XML representation since it has several RDF-like features: an object-property-value model similar to the RDF model, and extensive (if perhaps under-utilised) support for remote properties using `xlink:href`. This probably shouldn't come as a surprise: early versions of GML included an RDFS profile.

Earlier OGC standards used in GIS applications include "Web Map Service" (WMS) and "Web Feature Service" (WFS). WMS offers an interface to get information about a map layer, and to return that map layer for use in mapping software such as OpenLayers [WMS2010]. While straightforward, this method restricts the ways in which the data can be used – a map layer is, in essence, an image, and as such it is impractical to extract information from a layer to further manipulate it. WFS goes beyond this by

returning map data for a single feature in the OGC GML format [WFS2010]. However, this data is retrieved through a more complicated, non-RESTful RPC type service.

OGC SWE is a framework of open standards, designed to exploit Web-connected sensor systems via a Service Oriented Architecture [Bot2007]. It incorporates the Observations and Measurements (O&M) GML [Cox2007] and SensorML [SML2010] schema languages, to enable richly defined models for both sensor characteristics and observations. In addition, the framework includes additional services for discovering sensors (Sensor Registries), accessing sensor information (Sensor Observation Service), and receiving asynchronous sensor notifications (Web Alert Service).

Although Sensor Web Enablement is designed to provide for "Web-connected sensors", the approach taken in the design of the included services is to run *over* Web protocols, but not to adopt a Web *Architecture* through Resource Oriented services. While this is a valid and useful technique to extend GIS services into a more web-like platform, this specialisation of interfaces according to task (Sensor Observation, Alert, etc.) does not provide the kind of RESTful High-Level API required to support lightweight mashup development.

The data models and schemas (used to transfer information between server and client through the interface calls) are of more interest since they are based on a thorough and comprehensive *domain analysis*. Within SWE this is manifest in two perspectives over the data:

> A **provider-centric** approach orientated around and primarily describing the *processes* undertaken by sensors, structured networks of sensors, and constituent elements of sensors. Data is a product of the described sensor network. From the OGC standards this approach is adopted by the SensorML GML application schema and the SWE Sensor Planning Service.
>
> A **consumer-centric** approach orientated around and primarily describing the *observations and measurements* – i.e. the data, the results – captured by sensors rather than the sensors themselves (although the provenance of observations is modelled through an associated process). The OGC Observations and Measurements (O&M) model and GML application schema apply this approach, as used by the SWE Sensor Observation Service (SOS).

While the former might be applicable to provisioning, deploying, and managing sensor network themselves, our domain users (and the domain developers supporting them) are engaged in activities – such as emergency response planning and management – which instead are more aligned with manipulation of the data once it has been collected by the sensor network.

3.4. The SemSorGrid4Env Architecture

The SemSorGrid4Env architecture ([D1.3v2], figure 3-2) takes a Service Oriented approach to integrating data sources, middleware, and applications. While this contrasts with the Resource Oriented approach taken by REST APIs and generally followed in development of the High-Level API, the two approaches are also complementary.

Applications using the Architecture service APIs are more likely to be tightly integrated and dependent on the services discovered (and previously registered); an example of such a "full" application can be found in the SemSorGrid4Env Flood Planning and Response Application [D7.1v2].

Lightweight mashups are more likely to be developed quickly, potentially on an ad-hoc basis, and to take advantage of unintended re-use of sensor data. Semantic mashups benefit from the common self-descriptive models and linking provided by a REST API.

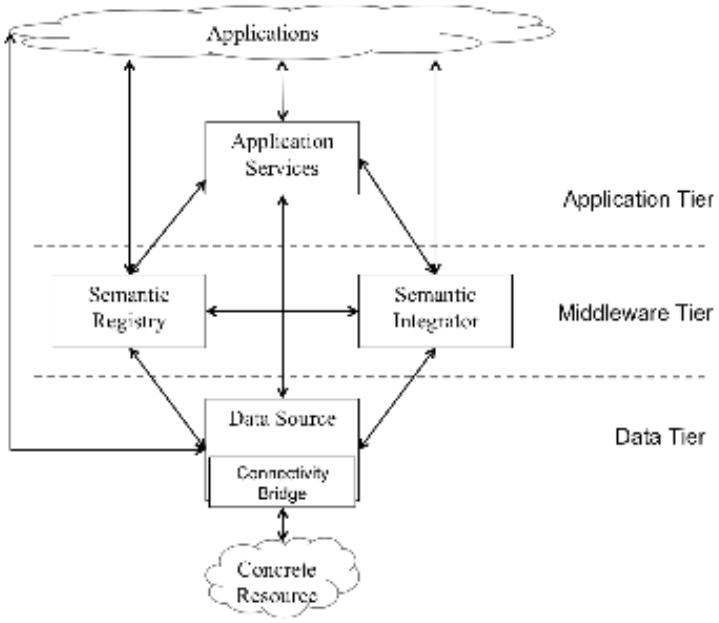

Figure 3-2: The SemSorGrid4Env architecture: the services, their relationships, and the classes that they belong to (from [D1.3v2])

4. Design Principles for High-Level APIs

In the previous chapter an overview of REST and Linked Data was presented, within the context of their suitability for supporting the developments models introduced in Chapter 2 through provision of a High-Level API which embraces Web Architecture.

At a first glance there might appear to be an obvious alignment and overlap between the approaches prescribed by REST and Linked Data; but despite their development on and around the architecture and technologies of the Web, they were developed in relative isolation. On more detailed inspection of the two, divergences in scope and applicability present themselves, and for some aspects, incompatibility. In this chapter we investigate these similarities and differences and suggest the coupling is worthy of a third look: in combination as a flexible environment in which the developer can focus on domain driven applications.

4.1. Domain-driven Design

As introduced in the previous chapter, the resource is the first-class citizen of both RESTful Web APIs and Linked Data exposed data sources; identifying resources and the representations that allow retrieval of them is key to writing the API.

This approach has echoes from existing software design practices and methodology when considering the object model derived *Domain-driven Design* philosophy [Eva2003].

Domain-driven design espouses that:

- The primary focus should be on the domain
- Complex domain designs should be based on a model

These principles are well aligned with the identification of resources (for the former) and the encapsulation of the domain by an ontology through Linked Data (the latter).

This process of "knowledge crunching" with domain experts and domain developers ensures an API exposes a model that is both pragmatic programmatically and representative of the domain:

"Good programmers will naturally start to abstract and develop a model that can do more work. But when that happens only in a technical setting, without collaboration from domain experts, the concepts are naive. That shallowness of knowledge produces software that does a basic job but lacks a deep connection to the domain expert's way of thinking." [Eva2003]

This is essential if the API is to be successfully used by domain developers, and in turn domain users: the power of a successful API is in encapsulating the complexity of a domain in a manner that allows its use to scale through simple usage. This simplicity must be deeply tied to the domain to allow natural and intuitive use by the domain

developers and users; an abstraction unfamiliar or unsuitable to them will have an effect opposite to that desired.

4.2. An Analysis of REST and Linked Data

The Linked Data movement has achieved considerable success constructing a semantic Web of Data [Biz2009]. While much initial semantic web research focussed on building a stack to enable reasoning and logic, the more recent Linked Data programme has attempted to reconnect the semantic web to its roots in the most successful distributed system ever constructed (or at the very least the latter half of its moniker!).

Moving on from an earlier assumption that URIs would do nothing more than uniquely identify Things, the key thrust of Linked Data has been the re-adoption of HTTP URIs for retrieval of resource representations. The approach can be summarised by the four Linked Data 'rules' [Ber2006]: use URIs as names for things; use HTTP URIs so that people can look up those names; when someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL); and include links to other URIs, so that they can discover more things.

A shallow keyword match over these principles would suggest a strong correlation with those underpinning REST [Fie2000], and yet rarely are the two mentioned together as complementary styles. Are they at cross-purposes, completely orthogonal, or can experience from both approaches inform a more coherent framework for building distributed web services and applications?

4.3. Similarities between REST and Linked Data

In this section we evaluate the commonalities between REST and Linked Data that support a new approach to High-Level API development encompassing both methodologies.

4.3.1. The primacy of resources

The key abstraction of information in REST is a resource [Fie2000]; similarly the URI is both the identifier for, and means by which relationships are expressed between, things in the Resource Description Framework (RDF) [RDF1999], which is the foundation of the Semantic Web stack. In both cases, the notion of an identifiable resource is fundamental to implementation; design and development of a system cannot progress without the assignment and association of resources.

Since Resource Oriented Architectures [Ric2007] and Linked Data are the most commonly encountered realisations of REST and the Semantic Web respectively, and since both are built upon *HTTP* and *HTTP* URIs, it is easy to recognise this as a common shared building block. It is therefore also relevant to note that neither REST as

an architectural style nor RDF as originally conceived are monogamously wedded to HTTP.

4.3.2. Linking is not optional

The fourth Linked Data principle is to "*Include links to other URIs*" in the representation provided when a URI is dereferenced "*so that they can discover more things*" [Ber2006]. It is this inclusion of links to other HTTP URIs which, when dereferenced, provide further links to more HTTP URIs that sets Linked Data apart from earlier Semantic Web activity in its explicit encouragement of a dereferencable Web (and the trails of links through it).

"*Hypermedia as the engine of application state*" (HATEOAS) is a defining characteristic of the REST architectural style [Fie2000]. State transitions in an application occur when moving from one resource to another (by retrieving or modifying) using the links provided in a representation.

A representation that supports linking is therefore a requirement for both approaches; neither would function as intended without the hyperstructures described above. While there is no specific mandated linked representation for REST implementations, Linked Data advocates "*using the standards*" which, in the case of RDF and SPARQL, both guarantee support for links to other resources.

4.3.3. Segregating semantics

Semantics about relationships between resources can be expressed by both approaches: in the Semantic Web they are described by ontologies written in RDFS and OWL, while RESTful implementations can encode semantics in link relations.

A common misapplication of both approaches is to assume semantics (or abuse implied semantics) encoded in a URI, when both REST and Linked Data explicitly expect clients to regard URIs as opaque strings when used for identification. In this way both follow the principle of separating identification from the semantics of interaction, description, and structure.

4.3.4. Adaptability

Both REST and the Semantic Web include facets in their design which allow the relationships between resources to be modified, should revision be required, without necessitating interface changes to the client.

Since state in a RESTful application is defined by navigation of the hyperstructure, if a server changes the links that are transferred to a client (via a representation) it also changes the possible state transitions the application can make. It does this without

changing the mechanism by which the client performs the transition (the combination of HTTP and the representations for the specified media type).

As befits a distributed web system (where it is perhaps unlikely – and probably undesirable – for there to be 'one true ontology'), there is no constraint on the application of a single ontology to each resource on the Semantic Web. Assertions can be made using different ontologies, in different places, and at different times; ontologies (themselves expressed in RDF) can be extended and subsumed by other ontologies.
In both cases this adaptability can be seen as a benefit of self-description – a client has prior knowledge of the framework within which relationships are expressed, but there is no requirement of prior knowledge of the relationships themselves.

4.3.5. Applicability of Domain Driven Design

The Domain Driven Design methodology introduced at the beginning of this chapter [Eva2003] espouses a focus on domain modelling throughout an iterative development process. This has particular resonance with the principles and practices outlined above in respect to both REST and Linked Data: the identification of resources and the links between them should naturally map to the domain (and business process) at hand [Rai2010], and the ability to iteratively modify the hyperstructure lends itself well to agile development.

This methodology is key in developing a service that can be successfully used by domain application developers, and in turn domain users: the power of a successful data service is in encapsulating the complexity of a domain in a manner that allows its use to scale through simple usage. This simplicity must be deeply tied to the domain to allow natural and intuitive use by domain developers and users; an abstraction unfamiliar or unsuitable to them will have an effect opposite to that desired.

4.4. Potential differences in the application of REST and Linked Data

In this section we outline those areas where one might perceive differences between REST and Linked Data – although, as we summarise in the next section, we counter that these are rather vestiges of different demands and current practice rather than fundamental incompatibilities.

4.4.1. API vs. Model

In the previous section we explored the similarities between REST and Linked Data, principally centred on the notion of resources and the relationships between them. There is, however, a key difference in the *motivation* for resource identification:

- in RESTful systems, resources and their relationships are identified and exposed to enable a client to retrieve data and transition to other resources; in effect, they define an API to enable application operation and state transition. Linking is the mechanism to navigate the API; link relations encode semantics to enable this.

- in RDF and ontologies, resources are identified to encapsulate an underlying data model. While Linked Data extends this idea so that sections of the model can be retrieved by dereferencing resources, linking in the returned representation is used to bind sections of the model rather than transition state.

By extension the adaptability and self-documentation described in section 4.3 **applies to API interactions for REST, and the data model for the Semantic Web**.

This distinction between *model* and *API*, principally in the identification of resources, is a key finding that informs our development of High-Level APIs.

4.4.2. SPARQL

The third Linked Data rule cites not only RDF, but a sister standard which from a RESTful point of view is a troublesome relative: SPARQL.

SPARQL is the standard query interface for RDF; it is widely deployed as an interface to Linked Data services, and widely used by Linked Data applications. However most SPARQL endpoints are implemented – and used – in the RPC style. RESTful interfaces to SPARQL have been proposed [Wil2009] which expose resources that, when a representation is requested, trigger SPARQL queries. Consistent with the previous section, identification of these query resources is a matter of identifying the "*information units*" which comprise the service API.

Perhaps a more concerning implication is the relative popularity of SPARQL for application development, and particularly for combining Linked Data through SPARQL endpoints. In this scenario, whilst the data model benefits from the distributed nature of resources and linking, the application interaction does not: it eschews the benefits of RESTful operation.

4.4.3. Content negotiation

RESTful services use content negotiation to select a shared envelope that both the client and server can encode and decode the representation through (and the interface to the service is then dynamically carried via the representation as links). Typically a REST service will assume the resource being transferred in these representations can be considered a document; in the terminology of the following section, an 'information resource'.

Linked Data services, in implementing the "HTTP range issue 14" solution [Sau2008], add semantics to the content negotiation to distinguish between URIs that are non-

information resources (identifiers for conceptual or real-world objects) and URIs that are information resources (documents) that describe the non-information resources. This is because assertions in the RDF graph are usually relationships that apply to the non-information resource, but Linked Data overloads URI usage so that it is also a mechanism for retrieving triples describing that resource (in a document, i.e. an information resource)[5].

One widely deployed technical solution is to issue a 303 redirect from the non-information resource URI to a content-negotiated information resource (which will have representations containing descriptive information about the non-information resource; at least one representation will be an RDF serialization; see section 3.2). The HTTP redirect signals the transition from non-information resource to the client. The practical consequence of the redirect is, in our experience, a (variable but) measurable additional delay for each complete transfer of information between server and client [Rou2010]; there is added complexity when compared to a REST API in which everything is simply an information resource.

4.4.4. RESTful through and through?

While there is clearly alignment in approach, and overlap in parts of implementation, are deployed best practice Linked Data services RESTful? On two further counts, we believe they could be considered to fall short.

Firstly, because resources are identified primarily for the purposes of correctly modelling the data (section 4.4.1), less thought is applied to the Linked Data URIs that can be dereferenced and how an application might use them – and the links between them – for RESTful state transition. If an API has not been designed for HATEOAS, then perhaps it is understandable that Linked Data developers appear to prefer SPARQL; or that adoption of SPARQL reduces motivation to design an API with HATEOAS in mind.

Secondly, the majority of Linked Data sites are read-only: they publish data but few have the ability to modify it (i.e. PUT, POST or DELETE). This may, in part, be due to the political Open Data movement which is frequently hard to distinguish from the technical push for Linked Data. Proposals for a SPARQL Update are well progressed, but carry the expected RPC issues; and while a Uniform HTTP Protocol for Managing RDF Graphs has been proposed, it remains a mechanism to encode SPARQL commands that are applied to a whole graph store, rather than manipulation of specific resources exposed through a RESTful API.

[5] This is a change in behaviour from earlier use of HTTP URIs in RDF, when they were *not* expected to be dereferenced.

4.5. Combining REST and Linked Data for Domain Driven Design

In the previous sections we outlined where RESTful and Linked Data approaches share a common method and where they diverge. *We do not, however, believe the differences are irreconcilable*: while worthy of note, issues surrounding SPARQL, 303 redirects, content negotiation and writeable resources could all be mitigated or indeed solved though modifications to implementation and convention.

On the point of API vs. Model, we regard this as a complementarity rather than a "difference", particularly when considered in the context of domain driven design.

It is important for a domain expert (or developer) to be able to use clear domain models that separate concerns to enable the manipulation of the domain data: this is a task RDF has proven adept at. It is equally important for a domain developer to be able to quickly and simply access, modify, and publish domain data through a lightweight API for scalable and distributed services: which REST enables.

If common models can be used for both the API design (the RESTful interactions with resources) and the modelling of resource relationships (the RDF and ontologies) then the focus of complexity in any application can be where it really matters: the domain driven design.

From a RESTful service design perspective, providing Linked Data representations offers an opportunity to use a common domain model for expressing, and identifying, the resources exposed by the *API* as well as the data model and for linking resources (within a particular service, and between services); Linked Data (RDF) uses a self-describing semantic model beyond the relatively simple link semantics in most REST deployments.

From a Linked Data perspective, this presents an opportunity for more sophisticated description and navigation of links in representations, and through this the application of stronger semantics (with a common underlying model) for application state transitions and the development of true RESTful application development using Linked Data, beyond the current polarisation around SPARQL endpoints.

4.6. Summary of Design Principles for High-Level APIs

Based on our evaluation of suitable architectures to support mashup development and comparison REST and Linked Data, we propose the following design principles for High-Level API development:

1. Agile development of lightweight mashups is best supported by Resource Oriented service architectures. Reduce complexity for mashup developers through the simplification of access methods espoused by REST. To develop a good API of this type requires careful and successful identification and design of *resources* by the service provider.

2. Identification of resources must be undertaken within the context of the domain of the data. Use Domain Driven Design as a flexible and suitable methodology to ensure that the knowledge of domain experts is drawn upon during the iterative design and development process that is identifying service resources.

3. Use Semantic Web data structures and ontologies (RDF, RDFS, and OWL) for canonical representations of resources; they share a common architectural heritage that makes them particularly suitable for use with REST. This enables development of a common domain model with self-describing link semantics beyond the relatively simple structures found in traditional REST deployments.

4. Identify resources to support both the domain model *and* the API. Provide Linked Data through content negotiation and a SPARQL endpoint, but also identify resources to enable RESTful application where hypertext is the engine of application state.

5. RESTfully provide other representations, derived from the domain model, to enrich the service for easy application development, as identified through the Domain Driven Design process.

5. Applying the Design Principles to Semantic Sensor Grids: Design of a High-level API for Observations

In chapter 4 we described the similarities and differences between the REST and Linked Data service architectural styles. Based upon this analysis we identified a 'best fit' approach that draws upon the strengths of each, and proposed a way forward to better serve the development of domain driven applications through a set design principles.

The overarching theme across the principles is the application of domain driven design to the High-Level API; for Semantic Sensor Grids this means the domain of sensor network data, and the domains relevant to the measurements sensed.

In this chapter we apply the principles in the design of a High-Level API for Observations, suitable for adoption by any Semantic Sensor Grid. To demonstrate its use by example, and to provide the domain driven basis required to apply the principles, we refine and specialise this API for a specific use-case and associated domain: the Channel Coastal Observatory sensor network.

The Channel Coastal Observatory (CCO) is the data management centre for the Regional Coastal Monitoring Programmes of England. Over a period of more than 5 years, the GeoData Institute has designed, built from the top down, and operated the data management infrastructure to run this programme. This includes software to manage and transmit real-time data from the largest network of coastal sensors in the UK; a data management infrastructure to manage data and metadata for over 65,000 environmental surveys of different types amounting to terabytes of storage; and a website to deliver real time and surveyed data to a public audience though highly complex dynamic map and data visualisation interfaces, serving over a million hits per month.

In each describing the design of a High-Level API for the CCO, we focus on four aspects through which the principles are applied:

1. the **Domain Model** (application of principles 1, 2 and 3)
2. identification of **Resources** (principles 1, 2, 3 and 4)
3. suitable **Representations** (principles 1, 3 and 5)
4. the **Web API** (principles 1, 2, 3, 4 and 5)

Beyond the principles, it is also worth recalling that any API is provided to support a *domain developer*, and the motivation for doing so is to enable semantic mashup applications that combine observation data from the API with other domain information retrieved from the linked data and RESTful web services (e.g. land use, transport).

5.1. The Domain Model

The Domain Driven Design "knowledge crunching" process was adopted both within the SemSorGrid4Env project (with domain experts in GeoData) and with local potential users and collaborators (see [D7.1v2]) to ensure the API exposes a model that is both pragmatic programmatically and representative of the domain.

Having surveyed existing data models, and in consultation with domain experts, it became clear that two different, but complementary, high-level approaches can be applied to sensor networks and their associated data (introduced in chapter 3):

> A **provider-centric** approach orientated around and primarily describing the *processes* undertaken by sensors, structured networks of sensors, and constituent elements of sensors. Data is a product of the described sensor network. From the OGC standards this approach is adopted by the SensorML GML application schema and the SWE Sensor Planning Service.
>
> A **consumer-centric** approach orientated around and primarily describing the *observations and measurements* – i.e. the data, the results – captured by sensors rather than the sensors themselves (although the provenance of observations is modelled through an associated process). The OGC Observations and Measurements (O&M) model and GML application schema apply this approach, as used by the SWE Sensor Observation Service (SOS).

While the former might be applicable to provisioning, deploying, and managing sensor network themselves, our domain users (and the domain developers supporting them) are engaged in activities – such as emergency response planning and management – which instead require manipulation of the data collected by the sensor network.

We therefore take a data-(consumer-)centric approach to the sensor data and adopt the Observations and Measurements (O&M) model [Cox2007] through both its GML Application Schema and, by including some key concepts (figure 5-1) in an ontology:

- The measured value/result
- The observed property (e.g. wave height)
- The process that made the observation (e.g. a sensor)
- The time at which the observation was asserted
- The time over which the sampling leading to the observation was taken
- The (domain specific) feature of interest that is being observed (e.g. the ocean)
- A mechanism for grouping observations (an observation collection).

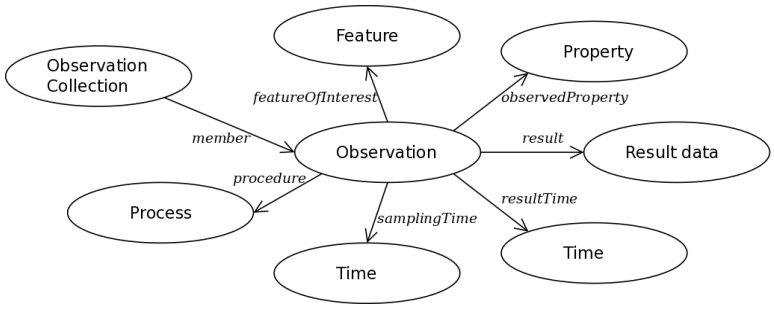

Figure 5-1: The basic concepts in an Observation model (from [Pag2009])

(While ontological evaluation has identified weaknesses in the O&M model [Pro2006], our primary focus is in its application for linking data and representations through a High-Level API, for which it is entirely sufficient.)

The ontology encapsulating O&M was then further developed as part of the SSN Ontology (figure 5-2) by the W3C Semantic Sensor Networks Incubator Group[6], which includes several members of the SemSorGrid4Env consortium amongst its membership (through Work Packages 4 and 5).

It is also included in the SemSorGrid4Env Ontology Suite [D4.3v2] where – through the observedProperty and featureOfInterest – it provides the crucial link between the measurements and the domain concepts the observations are capturing (specific examples of which are part of the domain model for the CCO API).

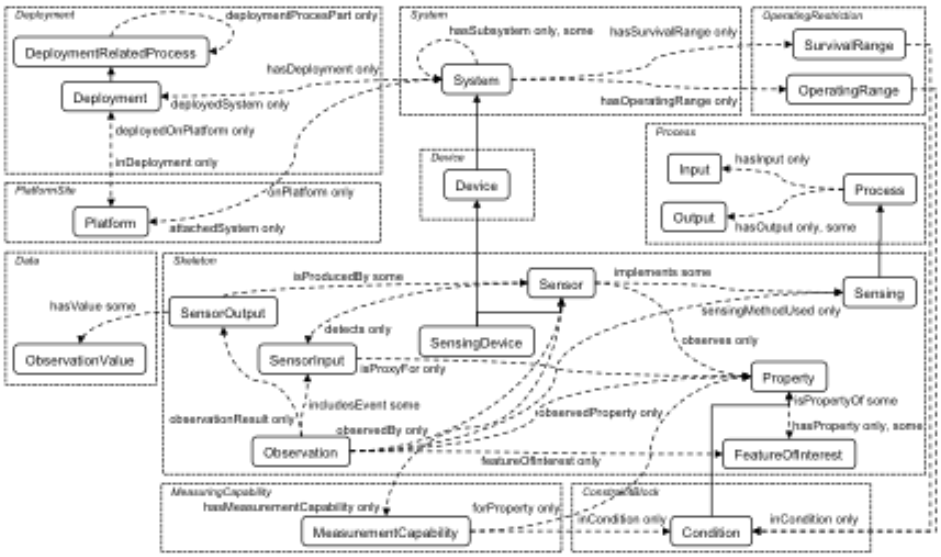

Figure 5-2: Overview of the SSG4Env ontologies (from [D4.3v2])

[6] http://www.w3.org/2005/Incubator/ssn/

5.2. Resources

Our approach so far has identified the generic model suitable for representing sensor data, and having adopted the O&M model it is clear that many of our resources will be *Observations*. But to construct a high-level API for particular sensors grids and the data they provide we must continue our domain-driven design – combining the principles and practice of REST APIs and Linked Data – and assess more detailed aspects of the specific domain for which we are constructing the API.

In this example implementation of a High-Level API for Observations, we focus on publishing data from the CCO network of marine and coastal sensors monitoring:

- wave height
- sea surface temperature
- wave period
- wave spread
- wave direction
- tide height.

As a RESTful Linked Data system, the high-level API is defined by its resources and the representations of those resources – in this case by the observations of the phenomena measured by the CCO above. In defining a resource we must necessarily create a globally unique identifier for it – a URI – the creation of which is frequently referred to as "*minting"* a URI. As noted in earlier sections, the URI for the resource should be treated as an opaque string when it is accessed through the API; while the implied structure within the URI is helpful when designing and maintaining the web service (and perhaps for developers when writing clients), client applications must navigate to and between resources using links between those resources. Use of the API must not rely on encoding of semantics within the URI – this clearly violates REST principles.

Identification and structuring of resources can be very dependent on the data (the resources) being exposed. For example, our primary observation resources are of the form:

http://id.channelcoast.org/observations/boscombe/Hs/20090801#140500

where the individual observation is dereferenced by retrieving the resource (which is an observation collection):

http://id.channelcoast.org/observations/boscombe/Hs/20090801

In this case, the observation of wave height (Hs) made by the Boscombe sensor on 01/08/2009 at 2.05pm is asserted within an observation collection of all wave height measurements taken by the Boscombe sensor on 01/08/2009.

This strikes a balance between the size of the retrieved resource representation and the number of links the client must retrieve *for this particular data set*. In this case the observations of *Hs* at the *Boscombe* sensor are taken half-hourly, so the resource that must be retrieved to dereference any one observation will contain 48 observations (all

the observations for the day). This grouping of resources (and the associated dereferencing) would not make sense if there were many observations per second.

Once more, note that the semantics that have clearly been used to structure the minting of the URIs – by our design – are not exposed through, or necessary for the operation of, the API. Relationships between resources must be expressed in the representations returned by the API, not within the syntax of the URI.

Nor does this primary statement of an observation constrain its use in other observation collections - an RDF model can be declared and reused across several resources by linking between statements. Using the example above, a collection of all measurements of wave height across the sensor network on 01/08/2009 would be identified by:

http://id.channelcoast.org/observations/Hs/20090801

and the primary statement of the observation above would be linked in by reference.

Figure 5-3 shows a number of relationships between key URIs, again focusing on the interface for retrieving significant wave height. Resources for other observed properties follow a similar structure, and examples of how these relationships are encoded in specific representations follows in a later section.

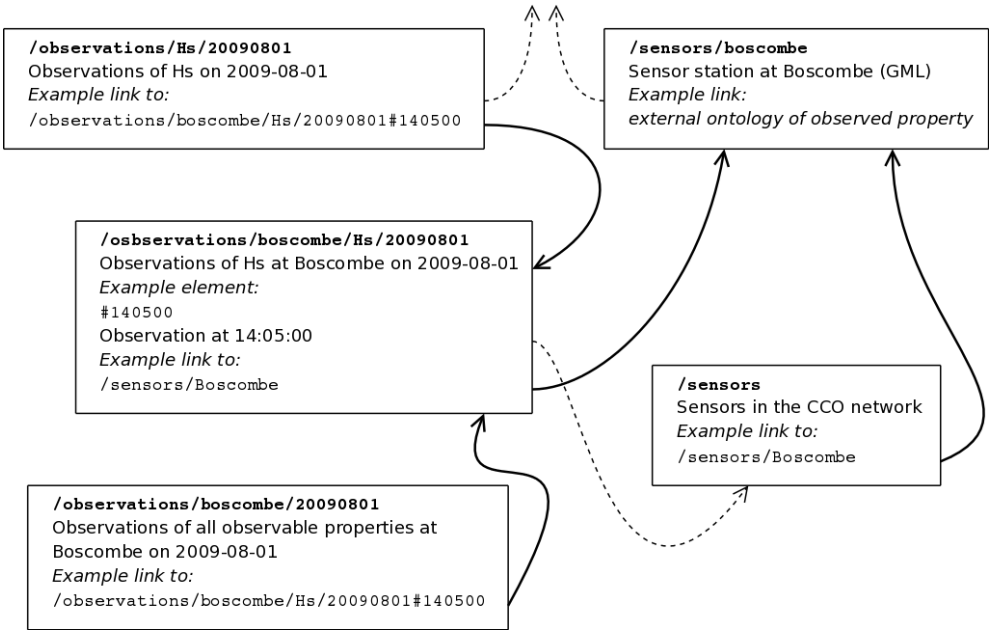

Figure 5-3: Relationships between some key URIs in the CCO system (from [Pag2009])

5.3. Representations

For each non-information observation resource (such as the wave height observation introduced above), the API provides several *representations* through a common

information resource and a further representation for backwards compatibility (with existing GIS systems) through a separate information resource.

The first representation is in **RDF XML**, and using the observations ontology introduced earlier. Here links are also made to domain ontologies for features of interest and observed properties.

The second representation conforms to the **O&M GML** schema. While the XML returned is very similar to that provided by the SOS GetObservation function, here we support a RESTful interaction by navigation between resources. This is made possible by the extensive support for XLink in GML and an underlying object-property-value model which closely resembles RDF.

We also note that the O&M GML representation of resources (and SensorML for appropriate resources) is compatible with the O&M GML and SensorML returned by the Sensor Observation Service (SOS)[SOS] (particularly the GetObservation and DescribeSensor functions). The design described herein can also be considered a partial implementation of a RESTful SOS; we hope that this will allow adaption of SOS clients to work with our API.

The third representation is in **HTML** and is a human browsable hyperlinked interface to the observation resources.

The fourth representation conforms to the **WFS GML** schema (XML). This representation provides compatibility with existing web GIS mapping tools (e.g. OpenLayers). The nature of these tools requires all the required data points (observations) to be provided in a single "layer" which can be overlaid onto a map; this flattened data structure is incompatible with a the other representations so must be provided through a separate information resource.

Further representations should be provided as appropriate to the domain and application, e.g. GeoJSON, KML.

When moving between representations it is important to note that the resources – the URIs – remain constant. As such, using this API a client application can move seamlessly between RDF and GML representations, taking advantage of the semantic linking provided in linked data, while being able to retrieve established encodings for Web GIS applications when required. Conversely an application can use a GML identifier as a jumping off point into the linked data web.

While there are clear benefits to alternately returning GML and Linked Data representations of a resource, each of these representations has particular constraints – be that conceptual model, design principle, or XML serialisation – and though they are in general complementary, a specific interactions can bring up incompatibilities between representations: we illustrate this with the following example to demonstrate that, with representation as well as resources, there are some design decisions which must be made for a particular domain and use case.

With regard to aggregation and nesting of observation collections, the O&M GML Application Schema is relatively constrained: more specialised observation collections

have typed constituents that are too specific to be transcluded in more general, otherwise specialised, or multiply nested, collections. While this is not an issue in existing O&M applications (e.g. using an SOS), the Linked Data principles lead us to both uniquely identify individual observations (i.e. avoiding duplication of a single observation at several URIs and creating duplication in the graph) and the identification and publication of aggregate resources that by their nature include observations that have already been 'used'/published as, or as a part of, another resource (i.e. we must use linking rather than duplication).

For example, it would be desirable to have the URI:

http://id.channelcoast.org/observations/boscombe/Hs

represent all the observations of *Hs* at *Boscombe*, and this to be an aggregation in the form of an *ObservationCollection*, where each member is in turn an *ObservationCollection* such as:

http://id.channelcoast.org/observations/boscombe/Hs/20090801

(there would be as many references to other *ObservationCollections* as there are days on which observations have been made).

While this is relatively simple to implement in RDF - ObservationCollection as a subClass of Observation, and member a TransitiveProperty with rdfs:range Observation - this is not possible using the O&M GML Schema. This leaves two possibilities for the O&M GML representation:

- Do not return a GML representation and return a 406 Not Acceptable code.
- 303 redirect straight to an information resource content negotiated for application/xml, without a common information resource shared with the RDF representation. The body returned contains a 'flattened' ObservationCollection with xlinks directly to the Observations required; this is the similar to the approach taken for WFS compatibility.

While the current CCO design takes the latter approach, without nested aggregations the number of Observations returned is potentially very high, and as such a compromise is made to redefine the resource as e.g. Observations of Hs at Boscombe over the last week (rather than all time). A hybrid approach might combine the original resource definition with a 406 for GML with a second resource limited to collections that are reasonable to return in GML (e.g. the observations from the last week).

5.4. Web API

In the previous sections we have outlined the domain model (in RDF) and identified key resources and representations. In combination, these form the core of the High-Level API for Observation applied to the CCO, but exposing the observation resources RESTfully as Linked Data with alternative representations for data formats useful to the domain developer.

To finish the API, and to satisfy the 4th Design Principle (section 4.6), we must consider any other functionality and identify any other resources to support the *domain developer*.

The first task, to complete Linked Data provision, is the implementation of a SPARQL query endpoint. This will, of course, utilise the same information encoded in the RDF representation for identified resources, structured according to the domain ontology already outlined.

The second task is to provide resources specifically in support of the API, that is, such that RESTful applications and mashups (driven by HATEOAS) can be written. Here we list some examples provided for the CCO implementation of the High-Level API, which were used to create the mashup examples described in chapter 7:

- /latest – relative within each observation collection, a resource that is always the most recent observation.
- "next" and "previous" – for each observation, a reference to the prior and next observations of that class.
- /summary – for each observation collection, a summary resource containing information about that collection, e.g. maximum/minimum values, frequency, averages, units of measurements, and descriptive metadata (this can be used by clients to calibrate visualisations and provide annotations).
- broader temporal collections appropriate to the data set (e.g. month) containing links to the constituent (e.g. daily) observation collections.
- links from constituent collections to the broader collections ("up") to enable better navigation through the data.
- /sensors – a collection of links to all procedures that generate observations (i.e. sensor platforms).
- For each sensor, a list to the "top-level" (temporally broadest) observation collections generated by that sensor platform.

5.5. API walk-through: the Channel Coastal Observatory

Exposing sensor data according to linked data principles and practice is a first and necessary step to enabling linked data sensor grid applications. In the sections prior to this we have introduced the building blocks of a dual purpose API design that combines the provision of linked sensor data with a RESTful interface to existing standardised data encodings such as OGC O&M and WFS, and in doing so have applied the design principles introduced in chapter 4.

In this final section describing design of the High-Level API for Observations we return to the example observation collection previously introduced and examine how a client using the API accesses a resource and negotiated for its representations (illustrated in figure 5-4).

The (non-information) resource:

http://id.channelcoast.org/observations/boscombe/Hs/20090801

is the set of all observations of Hs (significant wave height) from the Boscombe sensor on August 1st 2009.

As noted in earlier sections, the URI for the resource should be treated as an opaque string; while the implied structure within the URI is helpful when designing and maintaining the web service (and perhaps for developers when writing clients), client applications must navigate to and between resources using links between those resources.

When a client attempts to dereference this resource (e.g. through an HTTP GET), the web server responds with HTTP code "*303 See Other*" and the information resource:

http://data.channelcoast.org/observations/boscombe/Hs/20090801

Content negotiation is then used by the client to retrieve a suitable representation:

- *application/rdf+xml* returns an RDF representation.
- *application/xml* returns a GML representation.
- *text/html* returns an HTML rendering for viewing in a traditional web browser.

In each case the web server responds with code "*200 OK*" and sets the *Content-Location* header to the resource of the negotiated representation, e.g.

http://rdf.channelcoast.org/observations/boscombe/Hs/20090801

http://om.channelcoast.org/observations/boscombe/Hs/20090801

http://pages.channelcoast.org/observations/boscombe/Hs/20090801

followed by the appropriate representation in the HTTP body.

The intermediate stage of redirecting to a common information resource URI indicates to the client that the following content negotiation is for different representations of the same information. For example, this means that if the client has reached the resource through RDF representations, but needs to plot the data using an OGC compliant tool (e.g. within a mapping layer) it can request the GML representation knowing it is plotting the *same* information.

Other representations might, by necessity of the encoding, be of closely related but different information. An earlier incarnation of the CCO server implementation returned a GML representation using the WFS schema to create a MapServer compliant layer, and while we wish to preserve this functionality, the 'flattened' nature of the data in the WFS layer means this representation contains different information to those based on O&M (described below).

In this situation we use the content type *application/vnd.ogc.wfs* to enable a client to retrieve a backwards-compatible representation; when the client requests the non-information resource:

http://id.channelcoast.org/observations/boscombe/Hs/20090801

with content type *application/vnd.ogc.wfs*, the server performs a 303 redirect directly to the WFS GML (setting the *Content-Location* header accordingly):

http://wfs.channelcoast.org/observations/boscombe/Hs/20090801

In redirecting directly to the WFS information resource through content negotiation (rather than through the common information resource and then performing content negotiation), the web server has indicated that this is a *different* (but related) information resource, not a different representation of the same information resource.

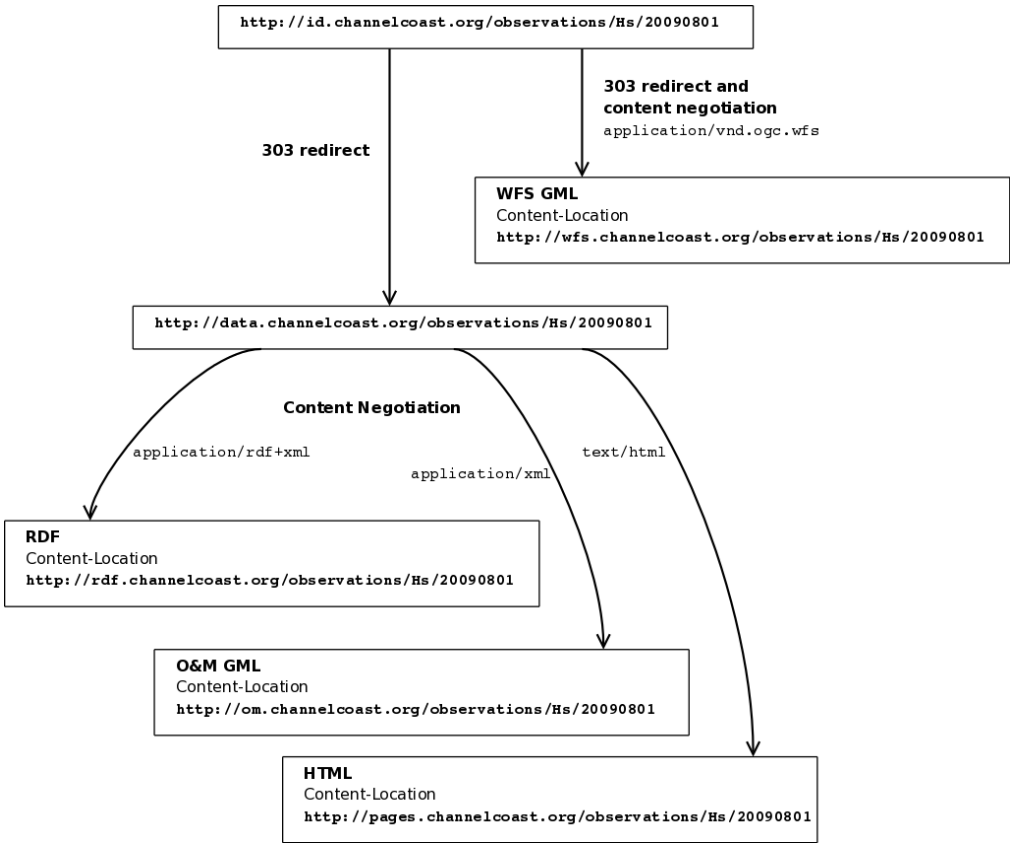

Figure 5-4: Resource representations in the high-level API (from [Pag2009])

As noted in earlier sections, the URI for the resource should be treated as an opaque string; while the implied structure within the URI is helpful when designing and maintaining the web service (and perhaps for developers when writing clients), client applications must navigate to and between resources using links between those resources.

The following XML fragments demonstrate the content (and similar structure) of the O&M GML and RDF representations of:

http://id.channelcoast.org/observations/boscombe/Hs/20090801

The O&M GML encoding is returned for *application/xml* :

```
<?xml version="1.0" encoding="UTF-8"?>
<om:ObservationCollection gml:id="this"
  xmlns:om="http://www.opengis.net/om/1.0"
  xmlns:gml="http://www.opengis.net/gml"
[...] >
  <gml:description>Wave height observations at Boscombe on 2009-08-01
      </gml:description>
  <om:member>
    <om:Observation gml:id="140500">
      <om:resultTime>
        <gml:TimeInstant gml:id="T140500">
          <gml:timePosition>2009-08-01T14:05:00</gml:timePosition>
        </gml:TimeInstant>
      </om:samplingTime>
      <om:samplingTime>
[...]
      </om:samplingTime>

      <om:procedure xlink:href=
          "http://id.channelcoast.org/sensors/boscombe"/>
      <om:observedProperty xlink:href=

        "http://marinemetadata.org/2005/08/ndbc_waves#Wind_Wave_Height"/>
      <om:featureOfInterest xlink:href=
          "http://www.eionet.europa.eu/gemet/concept?cp=7495"/>
      <om:result
        xsi:type="gml:MeasureType" uom="urn:ogc:def:uom:OGC:m">
         0.28
      </om:result>
    </om:Observation>
  </om:member>
  <om:member>
    <om:Observation gml:id="143500">
[...]
    </om:Observation>
  </om:member>
</om:ObservationCollection>
```

The RDF representation is returned for *application/rdf+xml* :

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:om2="http://rdf.channelcoast.org/ontology/om_tmp.owl#"
[...]
>
  <rdf:Description rdf:about=
      "http://rdf.channelcoast.org/observations/boscombe/Hs/20090801">
    <rdfs:label>
      An RDF representation of wave height observations at Boscombe on 2009-08-01
    </rdfs:label>
  </rdf:Description>
  <om2:ObservationCollection
      rdf:about="http://id.channelcoast.org/observations/boscombe/Hs/20090801">
    <om2:member>
      <om2:Observation rdf:about=
          "http://id.channelcoast.org/observations/boscombe/Hs/20090801#140500">
        <om2:resultTime>
          <om2:TimeInstant rdf:about=
            "http://id.channelcoast.org/observations/boscombe/Hs/20090801#T140500">
[...]
          </om2:TimeInstant>
        </om2:resultTime>
[...]
        <om2:procedure rdf:resource="http://id.channelcoast.org/sensors/boscombe"/>
        <om2:observedProperty rdf:resource=
            "http://marinemetadata.org/2005/08/ndbc_waves#Wind_Wave_Height"/>
        <om2:featureOfInterest rdf:resource=
            "http://www.eionet.europa.eu/gemet/concept?cp=7495"/>
        <om2:result [...]
[...]
      </om2:Observation>
    </om2:member>
  </om2:ObservationCollection
</rdf:RDF>
```

Consistent use of URIs is maintained in other observation collections. For example:

http://id.channelcoast.org/observations/Hs/20090801

contains parallel fragments in GML:

```
<om:ObservationCollection gml:id="this" [...]>

[...]

<om:member>

  <om:Observation xlink:href=

      "http://id.channelcoast.org/observations/boscombe/Hs/20090801#140500"/>

[...]
```

and in RDF:

```
<o2:ObservationCollection rdf:about=

    "http://id.channelcoast.org/observations/Hs/20090801">

[...]

<om2:member rdf:resource=

    "http://id.channelcoast.org/observations/boscombe/Hs/20090801#140500"/>

[...]
```

6. Implementation Patterns for the High-Level API Design

The previous chapters have presented the programming patterns and principles applicable to a high-level API for sensor grids, and the design for a specific API to publish sensor data from the Channel Coastal Observatory.

Taking the design in the previous section as an example, we present three implementation experiences realising this design in software for different scenarios:

1. exposing data using the API via a bespoke implementation based upon an existing web portal.
2. exposing data using the API by interfacing with the SemSorGrid4Env architecture and accessing sensor data through the architecture.
3. exposing data using the API provided by generic observation data sources (both via the SemsorGrid4Env architecture and from existing databases) using a semantic configuration utility and platform to automate API structuring and URI minting.

6.1. Bespoke Implementation of the API for an GIS web platform

6.1.1. Context

The Channel Coastal Observatory web portal (figure 6-1) is an established resource for users, exposing data through a web application with two major elements of functionality presented as separate options from the front page: “Realtime Data” and “Map Viewer and Data Catalogue”. As shown in figure 6-2 these are implemented independently, principally due to historical development and design decisions.

The map-viewing component is implemented using OpenLayers[7], a Javascript library for building web based geospatial applications. OpenLayers can present and integrate map data provided through several services and formats, including KML, Google Maps, Yahoo! Maps, and – as used by the CCO site – the WFS and WMS.

Session information – including data selection, preferences, and the “shopping basket” (which collates user selected sets data for ultimate download) – is handled by bespoke Ajax and server-side elements; the overall page is composed using the elements by PHP on the server.

[7] http://www.openlayers.org/

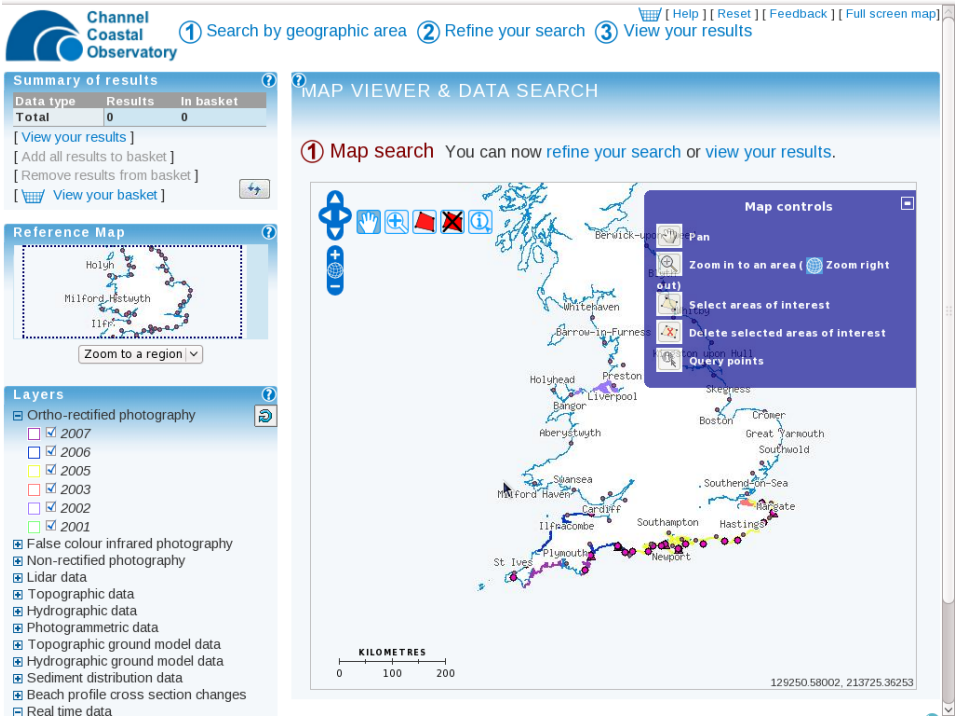

Figure 6-1: CCO Map Viewer and Data Catalogue

MapServer[8] is used to feed map data to the OpenLayers component using exposed WMS and WFS services. The MapServer instance is backed by a PostGIS[9] database storing the map and feature data (PostGIS spatially enables PostgreSQL through additional support of geographic objects).

Implementation of the high-level API must not cause regressions in the services already provided to users, and where possible is desirable (for maintenance reasons) to refactor and reuse code provided for existing interfaces (e.g. WFS).

6.1.2. Implementation

- The bespoke WSGI (Python) and PHP implementations for WFS and WMS are modified and extended to expose the new resources and their representations where possible (figure 4-2).
- A new map viewing component – the CCO API Explorer – is developed to showcase the API and demonstrate how its backwards compatibility representation allows it to be a drop in replacement for CCO WFS services where appropriate.

8 http://www.mapserver.org/

9 http://postgis.refractions.net/

- A partial implementation of the High-Level API for Observations is provided, limited by the canonical data representation being WMS/WFS – more sophisticated representations (such as Observation RDF) must be synthesised from this.
- Modifications to the underlying WMS/WFS data provision require corresponding updates to the implementation of other representations (e.g. RDF).

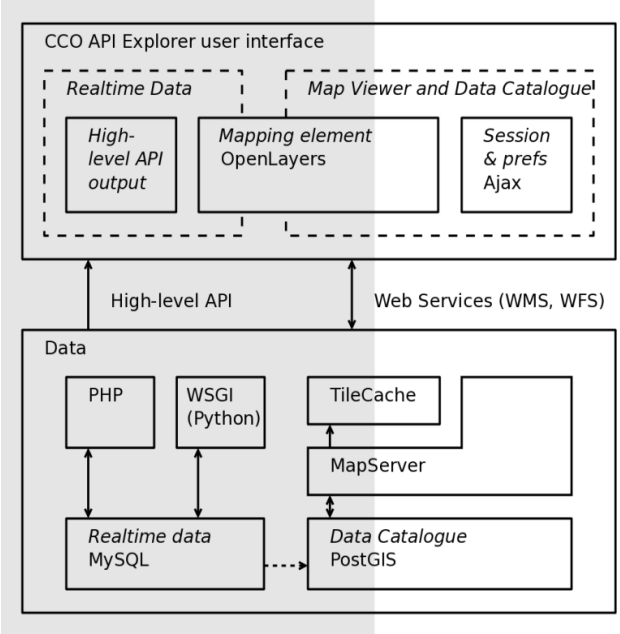

Figure 6-2: CCO components with elements altered by implementation of the API shaded grey

6.2. Adaptable Implementation for Specific Service Instances via a Sensor Web Architecture

6.2.1. Context

The phase I implementation of the SemSorGrid4Env architecture (figure 3-2) was an integration exercise to test the project architecture with instances of each of the different services: a data service, a semantic integration service, and a registry.

The scope of the demonstrator was again the Flood Use Case, and to support this CCO observations were exposed as a data source directly to the architecture. The web application constructed supports discovering and browsing data sources in support of coastal emergency response planning (e.g. flooding extend, asset inundation).

Application Tier development was required to enable the web application to access the CCO data now flowing through the architecture (through the Semantic Integration Service), principally by exposing a high-level API to the web application.

The Flood demonstrator application, though produced by developers involved as project partners in other aspects of the SemSorGrid4Env architecture, was designed and built as a standalone web application operating much like a "mash-up" could be constructed by a developer external to the project.

6.2.2. Implementation

The application tier libraries provide data to the web application via two interactions with the SemSorGrid4Env architecture (illustrated in Figure 6-3):

1. Semantic queries to the Registry. This library takes a SPARQL query from the web application, passes it to the semantic registry and, once the registry has located matching services, passes the endpoints back to the web application as a JSON array
2. **Data queries to the Integration Query Service.** The Integration Query Service (IQS) exposes data sources from the SemSorGrid4Env architecture through a SPARQL-STR interface. The application tier library makes the appropriate query to the IQS and as the resulting data is streamed back, converts it to GeoJSON files that can be loaded as a layer within the mapping component of the web application UI.

The second of these provided a limited implementation of the High-Level API for Observations. The primary representation in RESTful use of the API was GeoJSON, and a key difference from the previous bespoke implementation was the introduction of an internal data model for the observations, which was populated with the data returned from the IQS, and from which the GeoJSON used by the web application was generated. This alleviated much of the fragility associated with the bespoke implementation.

However, the Application Tier Libraries took the form of a component which, while adaptable, needed to be re-coded to work with different input interfaces and to serialise to different representations – it could not simply be reconfigured without rebuilding the software.

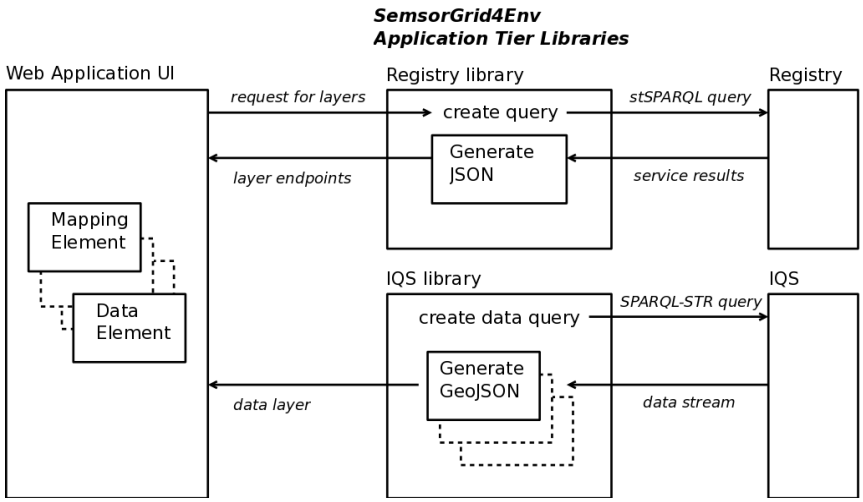

Figure 6-3: Application tier libraries (from [D7.4v1])

6.3. Implementation of a Generic High-Level API for Observations platform

6.3.1. Context

While the previous two implementations proved the utility of the High-Level API design, and that it could be used by application developers as intended, they did not prove the practicality of deploying the API: in both cases, aspects of the API service had to be modified at the code level specifically for the implementation at hand.

To allow the High-Level API to be deployed over a greater range of data sources without the need to re-code the software each time, a new service was developed to take advantage of the semantics encoded in the domain models so as to move all of the deployment specific detail into configuration files which can be setup by the domain expert and system administrator as appropriate.

6.3.2. Implementation

The HLAPI system has been designed to expose the general HLAPI design for generic data sources, as described in chapter 5. An overview of the system is shown in Figure 6-5. To achieve this, and achieve tractable configurability, incoming data is transformed into the known observation model. When data arrives in the system – either through a database insert, or through the SemsorGrid4Env architecture – the corresponding event trigger is activated, and determines what to do with the data. If the data represents an observation that we wish to serialise, the event trigger sends the data to the Processor to be turned into an RDF representation of an observation. If the data does not represent an observation, it is ignored. The generated RDF observation forms the canonical representation of the observation, as it is the most flexible and fully featured representation. All other serialised outputs are lower-information representations, and are therefore derived from the RDF.

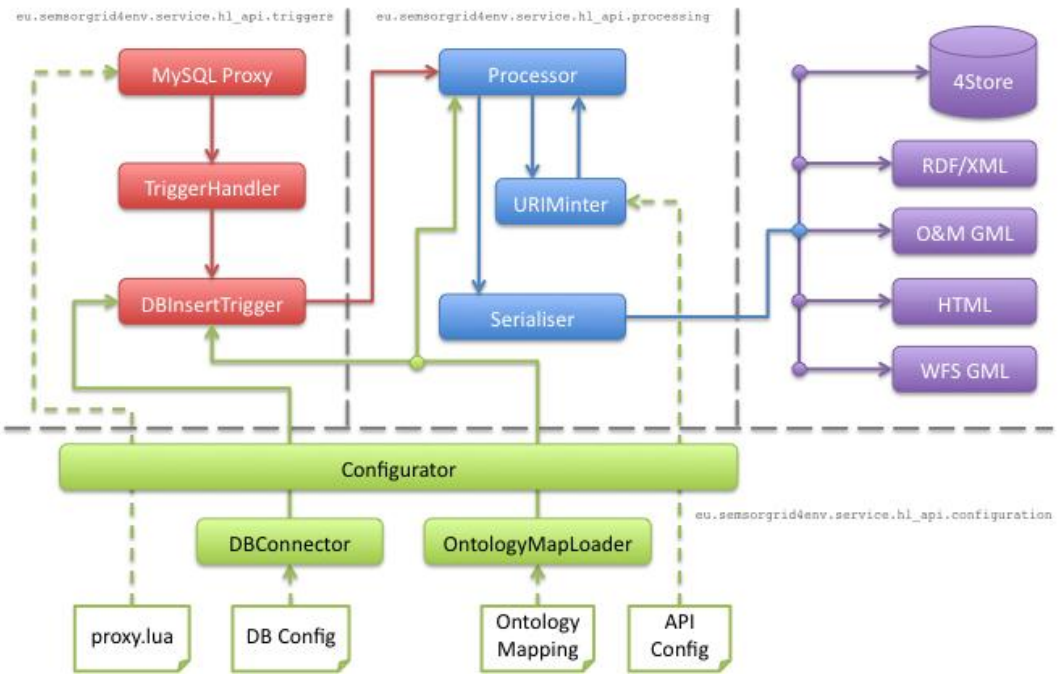

Figure 6-4: System overview of the HLAPI for Observations engine

The outputs to be serialised are determined using the API configuration file. This file defines the observation collections that the current observation should appear in, the formats in which to serialise them, and what the corresponding URIs should be. This configuration file is kept separate from the ontology mapping file, in order to separate the administrative concerns of different users; a domain expert is able to configure the mapping of the data source into the observation model, while the system administrator is able to handle to configuration of the exposed APIs.

Full details of the HLAPI engine implementation can be found in [D5.2v2]; they can be summarised as:

- Adopts the Observation Model as the core (and assumed) data structure
  - Further domain specific mappings (e.g. for Features of Interest and Observed Properties) can be configured in RDF by a domain expert (and separately from the systems administration).
- Event-driven, and can be triggered by streaming data sources (e.g. a SemSorGrid4Env architecture service) or live database inserts
  - A system administrator can configure how data sources are mapped to the Observation model and domain model. Standard triggers are provided for SemSorGrid4Env architecture services and MySQL databases.
- A number of standard representations are supported "off the shelf" by the engine: RDF, WFS, O&M GML, GeoJSON, HTML.
- The API is realised independently through a configuration file which lists resources: these are the canonical observations, different observation collections, and sensors.

    - Using this API configuration, the domain ontology mapping, and trigger configuration, the HLAPI engine automatically generates the resources required and applicable representation.
- RDF resources are automatically populated in a SPARQL endpoint.
- REST API extensions are automatically populated when data is available and relationships between resources exist.
- Provides a full implementation of the High-Level API for Observations.

7. Use Cases and Example Semantic Mashups

Having introduced a High-Level API for Observations and implemented services to expose datasets using the HLAPI, in this chapter we briefly outline how a *domain developer* might use the HLAPI to develop semantic web applications and mashups.

It is worth noting that:

- These semantic mashups demonstrate the potential for positive unintended re-use of sensor data. While they do not make full use of all of the features of all of the data (this is better demonstrated in the full SemSorGrid4Env Flood Application), they show that a little amount of semantically annotated sensor information can prove useful to many different use cases.
- By their nature of being a *mashup*, these web applications make use of other (often Linked Data) information sources in addition to those exposed by the HLAPI. *Linking from and in the HLAPI is crucial in this regard*, via domain ontologies and instances.
- The mashups were each coded by a single web developer, unfamiliar with the CCO data sources (but with some general semantic web familiarity) in a short period of time.

7.1. Recreational re-use: sea state and linked amenities for surfers

One of the CCO sensors is based near Boscombe, where the UK's first artificial surf reef has been constructed. This mashup shows the surfer the size of swell, received from the sensor network, and should the surfer then decide to visit, details of local amenities (pubs, car parks) and relevant information (road safety) all generated from Linked Data.

Scripting language and libraries

This example uses the PHP[10] scripting language. For Sparql queries and RDF manipulation it uses the Arc2[11] library and, for ease of coding and readability, Graphite[12]. The Google Chart API[13] is used for charts, and the Google Static Maps API[14] and Openlayers[15] for mapping.

[10] http://php.net

[11] http://arc.semsol.org/

[12] http://graphite.ecs.soton.ac.uk/

[13] http://code.google.com/apis/chart/

[14] http://code.google.com/apis/maps/documentation/staticmaps/

[15] http://openlayers.org/

Another useful tool is an RDF browser such as the Q&D RDF Browser[16].

First we load in the Arc2 and Graphite libraries and set up Graphite with a list of namespaces for coding simplicity.

```
require once "arc/ARC2.php";
require once "Graphite.php";
$graph = new Graphite();
$graph->ns("id-semsorgrid", "http://id.semsorgrid.ecs.soton.ac.uk/");
$graph->ns("ssn", "http://purl.oclc.org/NET/ssnx/ssn#");
$graph->ns("DUL", "http://www.loa-cnr.it/ontologies/DUL.owl#");
$graph->ns("time", "http://www.w3.org/2006/time#");
```

This continues for other useful namespace prefixes. The `id-semsorgrid` prefix is added for further code brevity.

Displaying a map of all wave height sensors

One of the observation serialisations available from the CCO deployment of the HLAPI is a GeoJSON format. This serialisation, which shows the locations of all wave height readings made in a particular time frame, can be rendered by various mapping engines including Openlayers.

The markup to display the map, given the path to a GeoJSON file, is very simple and fully documented by Openlayers.

Depending on how the HLAPI is configured, the resource for of wave height readings for a particular hour may be at:

```
http://id.semsorgrid.ecs.soton.ac.uk/observations/cco/Hs/20110215#01
```

which would then be content-negotiated to the OpenJSON representation:

```
http://geojson.semsorgrid.ecs.soton.ac.uk/observations/cco/Hs/20110215/00
```

[16] http://graphite.ecs.soton.ac.uk/browser/

Given this URL a map such as the following may be generated:

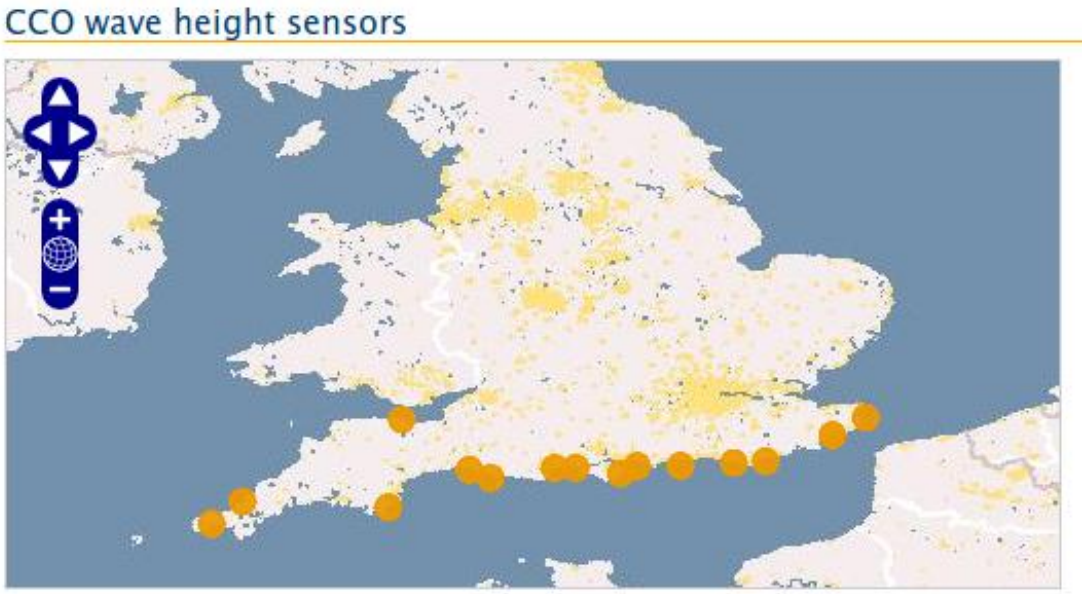

Figure 7-1 – Example of sensor and wave heights retrieved from the HLAPI via an OpenLayers map interface

Getting the day's wave height readings and the sensor metadata

In the case of the CCO deployment, the current day's wave height readings for the Boscombe sensor are identified by

```
id-semsorgrid.ecs.soton.ac.uk:observations/cco/boscombe/Hs/latest
```

We can direct Graphite to load the resources into a graph – Graphite and the HLAPI will automatically negotiate a content type that can be used. We're using the namespace we defined above for brevity.

```
$graph->load("id-semsorgrid:observations/cco/boscombe/Hs/latest");
```

Graphite allows the graph to be rendered directly as HTML to quickly visualise what is available. The same can be achieved by using a dedicated RDF browser.

```
echo $graph->dump();
```

The beginning of the output is something like the following:

```
id-semsorgrid:observations/cco/boscombe/Hs/20110215 -> rdf:type ->
   DUL:Collection -> DUL:hasPart ->
      id-semsorgrid:observations/cco/boscombe/Hs/20110215#000000,
      id-semsorgrid:observations/cco/boscombe/Hs/20110215#003000,
      id-semsorgrid:observations/cco/boscombe/Hs/20110215#010000

id-semsorgrid:observations/cco/boscombe/Hs/20110215#000000 ->
   rdf:type -> ssn:Observation -> ssn:observedBy ->
   id-semsorgrid:sensors/cco/boscombe -> ssn:featureOfInterest ->
```

```
   http://www.eionet.europa.eu/gemet/concept?cp=7495 ->
   ssn:observedProperty ->
   http://marinemetadata.org/2005/08/ndbc waves#Wind Wave Height ->
   ssn:observationResult ->  :arce2d5b1 ->
   ssn:observationResultTime ->  arce2d5b3 <- is DUL:hasPart of
   <- id-semsorgrid:observations/cco/boscombe/Hs/20110215
```

The bnodes are also shown, and their IDs can be traced to see which properties are available on each node.

A lot of useful information such as the sensor's coordinates is attached to the sensor's URI, which is linked from each `ssn:Observation` node. It's easy to get the URI, simply by getting `ssn:Observation` nodes, and then collecting the first found `ssn:observedBy` property of any of them. It's important to handle the case where there are not yet any results.

```
$sensor = $graph->allOfType("ssn:Observation")->
   get("ssn:observedBy")-> distinct()->current();
if ($sensor->isNull())
   die("No results yet today");
$sensorURI = $sensor->uri;
```

To get the sensor's coordinates we ask Graphite to dereference the sensor's URI and load its triples, then traverse the expanded graph to fetch the required values. The traversals here can once again be visualised by first dumping the graph or exploring the graph in any RDF browser.

```
$graph->load($sensorURI);
$location = $graph->resource($sensorURI)->get("ssn:hasDeployment")->
   get("ssn:deployedOnPlatform")->get("sw:hasLocation");
$coords = array(floatVal((string) $location-> get("sw:coordinate2")->
   get("sw:hasNumericValue")), floatVal((string) $location->
      get("sw:coordinate1")-> get("sw:hasNumericValue")),);
```

To collect all wave height observations we query the graph for all nodes of type `ssn:Observation` and skip over those whose `ssn:observedProperty` property is not that which we are looking for (just in case we have other observation types in our graph).

Each observation corresponds to a particular time interval so we need to collect the time (in this example we'll associate the end of the time interval – `time:hasEnd` – with the reading) as well as the wave height observation itself. The code snippet below also skips any observations whose `ssn:observationResultTime` property doesn't point to a node of type `time:Interval`, but it would be trivial to also parse nodes of different time classes.

Finally in this snippet the array of observations is sorted by time.

Again, to see how the traversal is built up it is easiest to inspect the graph visually.

```
$observations = array();
foreach ($graph->allOfType("ssn:Observation") as $observationNode) {
   if ($observationNode->get("ssn:observedProperty") !=
     "http://marinemetadata.org/2005/08/ndbc waves#Wind Wave Height")
   continue;
$timeNode = $observationNode->get("ssn:observationResultTime");
if (!$timeNode->isType("time:Interval"))
   continue;
```

```
$time = strtotime($timeNode->get("time:hasEnd"));
$observations[$time] = floatVal((string) $observationNode->
   get("ssn:observationResult")->get("ssn:hasValue")->
      get("DUL:hasDataValue")); }
ksort($observations, SORT_NUMERIC);
```

Visualising the data

The array resulting from the code above can be used to produce a chart of the wave heights. Explaining the snippet below is beyond the scope of this document, but it uses the Google Chart API to produce a line graph of wave height against time.

```
// organise data
$keys = array keys($observations);
$start = array shift($keys);
$end = array pop($keys);
$period = $end - $start;
$datax = $datay = array();
$maxheight = ceil(max($observations) * 10 * 1.2) / 10;
foreach ($observations as $time => $height) {
   $datax[] = ($time - $start) * 100 / $period;
   $datay[] = $height * 100 / $maxheight;
}

// x axis labels
$axisx = array();
for ($time = $start; $time <= $end; $time += $period / 6)
   $axisx[] = date("H:i", $time);

// parameters for Google Chart API
$chartparams = array(
"cht=lxy", //line x-y
"chs=340x200", //size
"chco=0066cc", //data colours
"chm=B,99ccff,0,0,0", //fill under the line
"chd=t:" . implode(",", $datax) . "|" . implode(",", $datay), //data
"chxt=x,y,x", //visible axes
"chxr=0,0,100|1,0," . $maxheight, //x and y axis ranges
"chxl=0:|" . implode("|", $axisx) . "|2:|Time", //custom labels for
axes, evenly spread, also axis titles
"chxp=2,50|3,50", //positions of axis titles
"chf=bg,s,ffffff00", //transparent background );

// output chart
echo '<img src="http://chart.apis.google.com/chart?' . implode("&",
$chartparams) . '">';
```

It's easy to show a map with the sensor's position highlighted, too: the following uses the Google Static Maps API to do this.

```
echo '<img
src="http://maps.google.com/maps/api/staticmap?size=300x200&center=' .
$coords[0] . ',' . $coords[1] .
'&zoom=8&maptype=hybrid&sensor=false&markers=' . $coords[0] . ',' .
$coords[1] . '">';
```

Fetching related data from other data sources

We can get the name of a nearby place and the nearest post code from the web services provided by Geonames[17]. Geonames returns XML that is easy to parse with PHP. Again, explaining how the external API call works is beyond the scope of this document.

```
// get nearby place name $placenameXML =
simplexml_load_file("http://ws.geonames.org/findNearbyPlaceName?lat={$
coords[0]}&lng={$coords[1]}"); $placename = array_shift($placenameXML-
>xpath('/geonames/geoname[1]/name[1]'));  // get nearby postcode
$postcodeXML =
simplexml_load_file("http://ws.geonames.org/findNearbyPostalCodes?lat=
" . $coords[0] . "&lng=" . $coords[1]); $postcode =
array_shift($postcodeXML->xpath('/geonames/code[1]/postalcode[1]'));
```

The postcode is used in the surf status mashup to fetch the British region name from Ordnance Survey, which in turn is used to fetch population and traffic accident data from Eurostat.

Data is also collected from Linked Geodata[18] to get the whereabouts of nearby facilities. For instance, to get parking facilities within five kilometres of the sensor, its SPARQL endpoint is queried as follows:

```
$store = ARC2::getRemoteStore(array("remote store endpoint" =>
   "http://linkedgeodata.org/sparql/"));
$rows = $store->query("
   PREFIX lgdo: <http://linkedgeodata.org/ontology/>
   PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
   PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
   SELECT * WHERE {{ ?place a lgdo:Parking . } UNION
      { ?place a lgdo:MotorcycleParking . } UNION
      { ?place a lgdo:BicycleParking . }
      ?place a ?type ;
      geo:geometry ?placegeo ;
      rdfs:label ?placename .
      FILTER(<bif:st_intersects>
      (?placegeo, <bif:st_point> ($coords[1], $coords[0]), 5)). }",
   "rows");
```

The returned results include the coordinates of each parking facility (`placegeo`), from which the distance to the sensor can be calculated.

Similar queries can be used to get data on other types of nearby amenities – the surf status mashup also locates nearby pubs, cafés and shops.

[17] http://www.geonames.org/

[18] http://linkedgeodata.org/

Finished mashup

The finished mashup, once styled, looks something like the screenshot shown in Figure 7-2 (with only three readings so far that day).

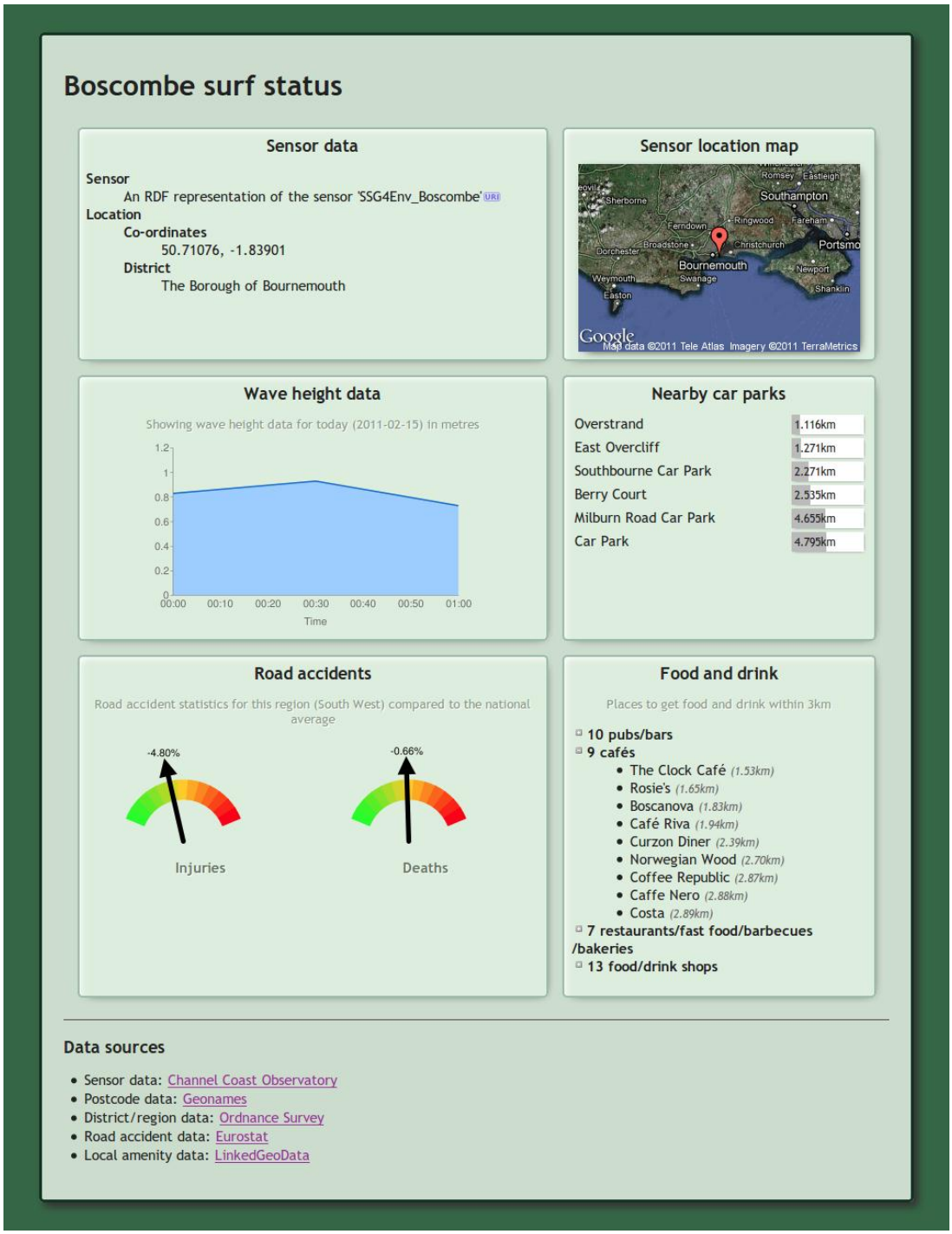

Figure 7-2 – Example mashup using HLAPI serialised data sources

7.2. Flood Gate status for the Coastal Defence Partnership

The Coastal Defence Partnership (CDP) is an alliance of three local governments along the Solent on the South Coast of the UK (Portsmouth City Council, Gosport Borough Council, and Havant Borough Council).

One of the responsibilities of the CDP Coastal Team is the co-ordination of flood protection barrier erection in the old town of Portsmouth should a flood event occur. Installation of flood barriers is dependent on tide and wave levels, and is sequenced.

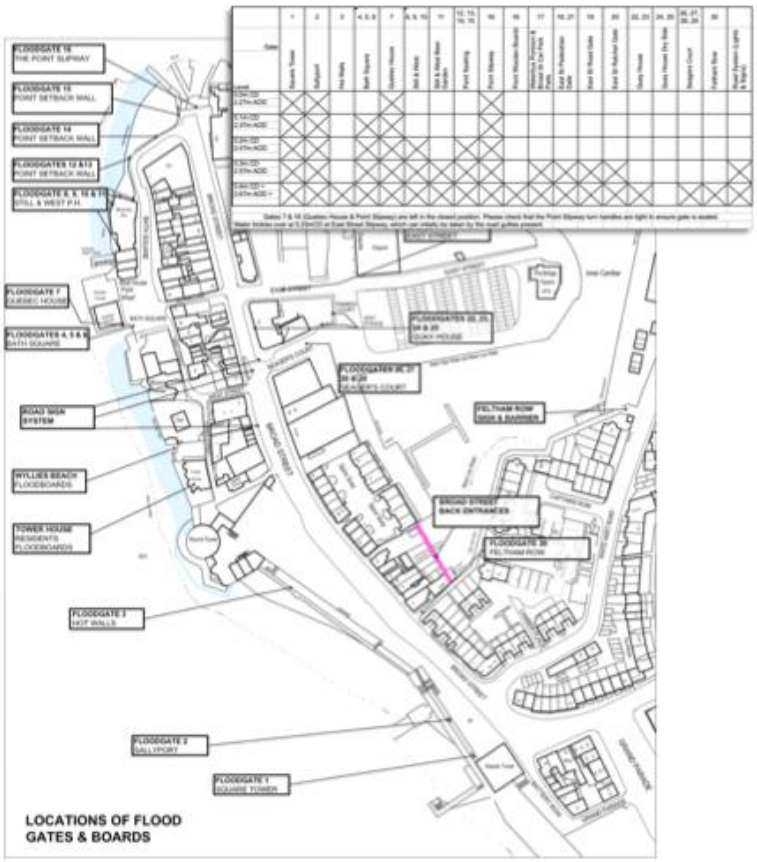

Figure 7-3 Locations of floodgates in Portsmouth, shown with the sequence in which they are closed during a flood (photograph courtesy CDP)

Flood barriers as well as flood water can block access to roads and facilities, so it can be useful to have an overview of which barriers are in place, which need to be erected next, and any relevant utilities that might be affected. The co-ordination is undertaken by team members on site, and it is plausible that, in the future, they may be equipped with internet enabled tablet devices – this is the scenario the prototype mashup looks to address.

Figure 7-4 A flood waters surround Portsmouth old town (photograph courtesy CDP)

Figure 7-5 A flood barrier deployed in Portsmouth (photograph courtesy CDP)

Mashup implementation

Development of the mashup follows the same basic pattern as the first example, starting with tide height measurements from the CCO sensor closest to the flood barrier location:

```
http://id.semsorgrid.ecs.soton.ac.uk/observations/cco/portsmouth/TideH
eight/latest
```

Rather than finding recreational amenities linked to the sensor location, the mashup finds linked data for critical services (police stations, hospitals, trunk roads), but using the same techniques.

The completed mashup is shown in figure 7-5.

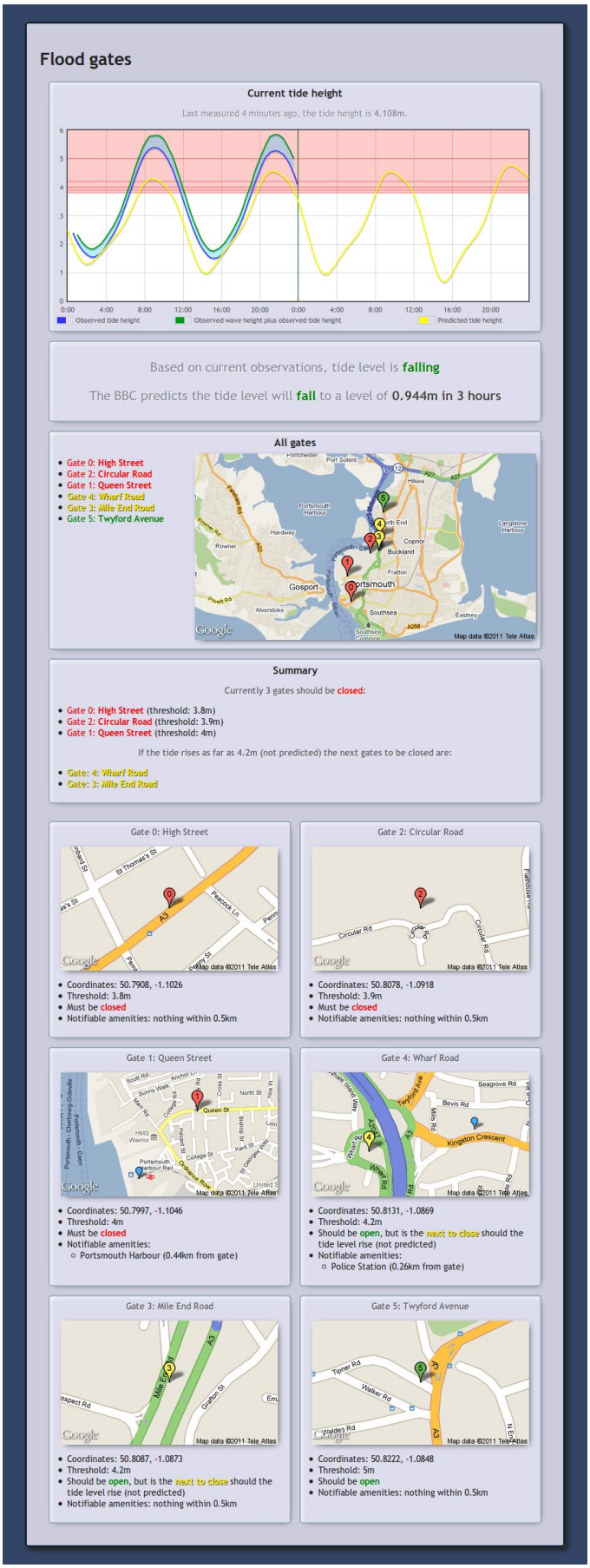

Figure 7-5: Mashup showing sea levels and flood gate status

8. Summary

High-Level APIs enable the quick and simple development of lightweight Web applications and mashups. We have explored the requirements for a high-level API for semantic sensor grids through domain driven design of a specific API for the Channel Coastal Observatory sensor data. Our objectives in developing the API can be summarised as:

1. To publish the sensor data from the CCO as linked data, enabling Semantic Web client applications that can combine these observations with other domain information retrieved from the linked data web (e.g. land use, transport)
2. To RESTfully publish sensor data to clients that support existing GML schema
3. To allow the development of hybrid clients which can transpose between linked data and GML (or other) representations, taking best advantage of both

In doing so we have developed best-practice principles for developing High-Level APIs, taking a Resource Oriented approach to simplify application development while semantically structuring domain data that forms the core of useful software. This combines the best of REST and Linked Data experience to support domain developers with lightweight, self-descriptive HLAPI, enabling them to quickly build bespoke applications using data in new and previously unforseen ways.

Our aim throughout is to encourage the use of High-level APIs to generate a new class of rapidly developed applications in support of sensor grids. Traditional GIS systems are often large and complex – adding support for a new use case implies enlargement of the functionality of the application. We believe the lower barrier of entry in developing simple web applications and mashups, with the associated shorter lead time and lower costs, will encourage the development of many varied and specialised web based applications suited to individual tasks that are not practical propositions in a monolithic GIS environment.

References

[Ala2010] Alarcon, R., Wilde, E. (2010) *"Linking Data from RESTful Services".* In proc. LDOW 2010, Raleigh, North Carolina

[D1.4v1] Aparicio, J., Gray, A., Kyzirakos, K., Sadler, J., and Page, K. (2010) "Reference SemsorGrid4Env Implementation - Phase I", Deliverable D1.4v1, SemSorGrid4Env

[Bec2000] Beck, K. (2000) *Extreme Programming Explained: Embrace Change* 2nd. ed. Addison-Wesley, 2000 pp. 54

[Ben2008] Benslimane, Djamal; Schahram Dustdar, and Amit Sheth (2008). "Services Mashups: The New Generation of Web Applications". *IEEE Internet Computing, vol. 12, no. 5*. Institute of Electrical and Electronics Engineers. pp. 13–15.

[Ber2006] T. Berners-Lee. Linked Data, Design Issues. http://www.w3.org/DesignIssues/LinkedData.html, July 2006.

[Biz2009] Bizer, C., Heath, T. and Berners-Lee, T. (2009) *"Linked Data - The Story So Far,"* Int. Journal on Semantic Web and Information Systems, Special Issue on Linked Data

[Bot2007] Botts, M. *et al.* (2007) *"OGC Sensor Web Enablement: Overview and High Level Architecture".* White Paper, published by Open Geospatial Consortium Inc.

[D7.1v2] Hutton, C., Sadler, J., Page, K., Clark, M., Newman, R. and Roe, S. (2010) "Flood user requirements specification", Deliverable D7.1v2, SemSorGrid4Env

[Cox2007] Cox, S. (2007) *"Observations and Measurements - Part 1 - Observation schema (OpenGIS Implementation Standard OGC 07-022r1)"*. Technical report, Open Geospatial Consortium Inc.

[Eva2003] Evans, E. (2003) "Domain-driven Design: Tackling Complexity at the Heart of Software", 1st Edition, Addison Wesley, ISBN 978-0321125217

[Fie2000] Fielding, R. T. (2000) *Architectural Styles and the Design of Network-Based Software Architectures*, PhD dissertation, Dept. of Computer Science, Univ. of California, Irvine, California

[Fra2011] Frazer, A. J., De Roure, D., Martinez, K., Nagel, B., Page, K. R., Sadler, J. (2011) "Implementation and Deployment of a Library of the High-level Application Programming Interfaces", Deliverable D7.2v2, SemSorGrid4Env

[D4.3v2] Garcia-Castro, R., Hill, C., Corcho, O. (2011) "Sensor network ontology suite", Deliverable D4.3v2, SemSorGrid4Env

[Ger2006] Gerber, AJ, Barnard, A & Van der Merwe, Alta (2006) *"A Semantic Web Status Model"*. Integrated Design & Process Technology, Special Issue: IDPT 2006

[D1.3v2] Gray, A. J. G., Galpin, I.,Fernandes, A. A. A., Paton, N. W., Page, K., Sadler, J., Koubarakis, M., Kyzirakos, K., Calbimonte, J., Corcho, O., Garcia, R., Diaz, V., Liebana, I. (2010) "SemSorGrid4Env architecture – phase I", Deliverable D1.3v2, SemSorGrid4Env

[D7.1b] Hutton, C., Sadler, J., and Newman, R. (2010) "Flood user requirements specification update", Deliverable D7.1b, SemSorGrid4Env

[Lac2005] Lacy, L. W. (2005) "*OWL: Representing Information Using the Web Ontology Language"*. Victoria, BC: Trafford Publishing. ISBN 1-4120-3448-5.

[SML2010] OpenGIS Sensor Model Language (2010) available online at *http://www.opengeospatial.org/standards/sensorml*

[D5.1] Page, K. R., De Roure, D.C., Martinez, K., and Sadler, J. (2009) "Specification of high-level application programming interfaces", Deliverable D5.1, SemSorGrid4Env

[Pag2009] Page, K. R., De Roure, D.C., Martinez, K., Sadler, J. and Kit, O. (2009) *"Linked Sensor Data: RESTfully serving RDF and GML"*. In proc. 2nd International Workshop on Semantic Sensor Networks, Washington DC, 2009

[Pal2002] Palmer, S. R., Felsing, J. M. (2002) *A Practical Guide to Feature-Driven Development*. Prentice Hall. ISBN 0-13-067615-2

[Pro2006] Probst, F. (2006) *"Ontological Analysis of Observations and Measurements"*. In: Geographic Information Science, 4th International Conference (GIScience 2006).

[Pro2010] programmableweb.com (2010) "Frequently Asked Questions", available online at http://www.programmableweb.com/faq

[Rai2010] Y. Raimond, T. Scott, S. Oliver, P. Sinclair, and M. Smethurst. Use of Semantic Web technologies on the BBC Web Sites. In D. Wood, editor, Linking Enterprise Data, pages 263–283. Springer, 2010.

[RDF1999] Resource Description Framework (RDF) Model and Syntax Specification (1999) available online at *http://www.w3.org/TR/PR-rdf-syntax*

[Ric2007] Richardson, L., Ruby, S. (2007) *RESTful Web Services*. 1st ed. O'Reilly. ISBN 0-596-52926-0

[Rou2010] D. De Roure, C. Goble, S. Aleksejevs, S. Bechhofer, J. Bhagat, D. Cruickshank, et al. The evolution of myexperiment. In IEEE International Conference on eScience, pages 153–160. IEEE Computer Society, 2010.

[Sau2008] Sauermann, L., Cyganiak, R. (2008) *"Cool URIs for the Semantic Web"*. W3C Semantic Web Education and Outreach Interest Group Note

[Sch2004] Schwaber, Ken (2004) *Agile Project Management with Scrum*. Microsoft Press. ISBN 978-0-735-61993-7

[RDFS2004] W3C RDFS Specification (2004) available online at *http://www.w3.org/TR/rdf-schema*

[Wil2009] E. Wilde and M. Hausenblas. RESTful SPARQL? You name it!: aligning SPARQL with REST and resource orientation. In Proceedings of the 4th Workshop on Emerging Web Services Technology, pages 39–43. ACM, 2009

[WFS2010] WFS Implementation Specification (2010) available online at *http://www.opengeospatial.org/standards/wfs*

[WMS2010] WMS Implementation Specification (2010) available online at *http://www.opengeospatial.org/standards/wms*