Testbed-12 REST Architecture Engineering Report

Table of Contents

Publication Date: 2016-mm-dd

Approval Date: 2016-mm-dd

Posted Date: 2016-04-12

Reference number of this document: OGC 16-035

Reference URL for this document: http://www.opengis.net/doc/PER/t12-rest-er

Category: Public Engineering Report

Editors: Christoph Stasch, Simon Jirka

Title: **Testbed-12 REST Architecture Engineering Report**

OGC Engineering Report

COPYRIGHT

WARNING

LICENSE AGREEMENT

that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modifiedthe Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER’S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER

LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any

local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Abstract

REST interfaces facilitate the application of OGC standards in many novel application scenarios, e.g. implementing OGC clients on constrained devices, as they ease the implementation of service requests and simplify the interaction patterns. Thereby, REST serves as a complementary technology to the already existing SOAP/POX provided by most of the current OGC standards. This ER provides an overview on different REST service implementations in the Testbed-12 and in related activities. As a result, this ER can be used to develop recommendations on architecture guidelines for providing REST interfaces in the geospatial domain.

Business Value

Utilizing the REST architectural style for the exchange of geospatial information across the Web may ease development and deployment of both, clients and servers, and may have the potential to provide a simple, scalable and resilient way to exchange spatial information. Since REST principles are simple and relying on common Web standards, it may also link the OGC architecture to exisiting common interfaces and components, easing the integration of geospatial information with external (non-spatial) resources.

What does this ER mean for the Working Group and OGC in general

The REST Architecture ER describes the concepts for, implementations of, and experiences with providing REST APIs for different OGC services including WFS, WMS, WMTS, WCS and WPS. It also provides general considerations about REST APIs in the beginning of the ER and summarizes the findings made when conceptualizing and implementing the different REST APIs at the end of the ER resulting in general recommendations for OGC standards. As the Architecture DWG considers overarching architectural issues that are germane to multiple OGC(r) standards, we think that the REST Architecture ER is highly relevant for this group and that providing REST APIs and JSON encodings for geospatial resources broadens the field of potential applications and eases the implementation.

How does this ER relates to the work of the Working Group

The Architecture DWG considers overarching architectural issues that are relevant for several OGC standards. The REST Architecture ER describes general approaches for providing REST APIs for OGC services, which we consider being a general issue for all OGC services. One of the core aims of the Architecture DWG is also to provide encoding rules and best practices for (Geo-)JSON. We

consider this being closely related to the definition of REST APIs, as in most cases, REST APIs provide JSON encodings for the resources offered by the API. We hence propose that the Architecture DWG reviews the ER and comments on it.

Keywords

testbed-12, Resource-oriented Architecture, OGC REST Bindings, REST, engineering report, http

Proposed OGC Working Group for Review and Approval

This engineering report shall be submitted to the Architecture DWG for review and comment.

Chapter 1. Introduction

1.1. Scope

This OGC Engineering Report (ER) is a deliverable (A005-1) in the Linked Data and Semantics (LDS) thread of the OGC Testbed 12 activity. It describes the different REST services developed in this thread as well as considerations and recommendations for a RESTful OGC architecture.

1.2. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Table 1. Contacts

Name	Organization
Christoph Stasch (Editor)	52°North GmbH
Simon Jirka (Editor)	52°North GmbH
Benjamin Pross	52°North GmbH
Jeff Harrison	Carbon
Peter Vretanos	CubeWerx
Alex Mircea Dumitru	Jacobs University

1.3. Future Work

The following work items have been identified as future work items:

- Provide a common way for describing RESTful OGC services.
- Provide formal, machine-readable definitions of spatial associations (e.g. topological relationships such as within, intersects, etc.) under supervision of the OGC naming authority that can be utilized in resource-oriented architectures and linked data.
- Provide REST binding specifications for OGC Sensor Observation Service, OGC Web Processing Service, and OGC Web Coverage Service as well as for the OGC Pub/Sub standards. The RESTful services described in this document may serve as a basis for these standards.
- Provide a common way how to encode links in hypermedia (e.g. FeatureCollections) and non-hypermedia responsens (Tiles)

1.4. Forward

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any

relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 2. References

The following documents are referenced in this document. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies.

- OGC 06-042, OpenGIS® Web Map Server Implementation Specification
- OGC 06-121r9, OGC® Web Services Common Standard
- OGC 07-057r7, OpenGIS® Web Map Tile Service Implementation Standard
- OGC 09-025r1, OGC® Web Feature Service 2.0 Interface Standard
- OGC 10-004r3, OGC Abstract Specification Geographic information — Observations and measurements
- OGC 10-025r1, Observations and Measurements - XML Implementation
- OGC 12-000, OGC® SensorML: Model and XML Encoding Standard
- OGC 12-006, OGC® Sensor Observation Service Interface Standard
- OGC 15-018r2, OGC® WaterML2.0: Part 2 - Ratings, Gaugings and Sections
- OGC 15-033, OGC WaterML2.0 part 2 – RESTful API and JSON encoding
- OGC 14-065, OGC® WPS 2.0 Interface Standard
- OGC 15-052r1, OGC® Testbed 11 REST Interface Engineering Report
- OGC 15-078r6, OGC SensorThings API Part 1: Sensing
- OGC 15-100r1, OGC Observations and Measurements – JSON implementation

Chapter 3. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard [OGC 06-121r9] and in OGC® Abstract Specification Topic TBD: TBD shall apply. In addition, the following terms and definitions apply.

3.1. Application Programming Interface (API)

An interface definition that permits invoking services from application programs without knowing details of their internal implementation.

3.2. Hypermedia

Hypermedia is an extension of the term hypertext. In addition to the general media information, e.g. graphics, audio, or video, hyperlinks are provided that enable browsing between different information items.

3.3. Representational State Transfer (REST)

Representational State Transfer (REST) offers a clean, simple and easy-to-understand method of discovering, accessing and updating geospatial information. There is no special software to install. A web browser, a web application or a mobile app can be used to access the service directly - using standard HTTP methods (e.g. GET, PUT, POST, DELETE).

3.4. Representations

Representations describe the current or intended state of a resource at a certain point in time. A representation consists of a sequence of bytes and metadata to describe those bytes. As an example, a geographic feature resource may be represented by a GeoJSON document describing the spatial location and other feature properties or by a JPEG image providing a rendered representation of the feature. The media type definition provides the metadata needed to encode or decode the different representations.

3.5. Resource

Following Fiedling [1], "any information that can be named can be a resource". Thereby, a resource is considered as a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at a particular point in time. Examples of resources in the geospatial domain are map tiles (served by a WMTS), features (served by a WFS), feature types (served by WFS), or observations (served by an SOS).

Chapter 4. Conventions

4.1. Abbreviated terms

- API Application Program Interface
- HTTP Hypertext Transfer Protocol
- IDL Interface Definition Language
- JSON JavaScript Object Notation
- OGC Open Geospatial Consortium
- REST Representational State Transfer
- RMM Richardson Maturity Model
- SOA Service-oriented Architecture
- ROA Resource-oriented Architecture
- UML Unified Modeling Language
- URI Uniform Resource Identifier
- URL Uniform Resource Locator
- WCS Web Coverage Service
- WFS Web Feature Service
- WMS Web Map Service
- WMTS Web Map Tile Service
- WPS Web Processing Service
- XML Extensible Markup Language

Chapter 5. Overview

This ER provides an overview on the different REST implementations of the OGC Testbed-12, as shown in Figure 1. It also provides guidance and recommendations on which future specification activities of OGC standards can rely on to ensure a consistent specification of REST interfaces including common architecture guidelines for providing REST interfaces in the geospatial domain.

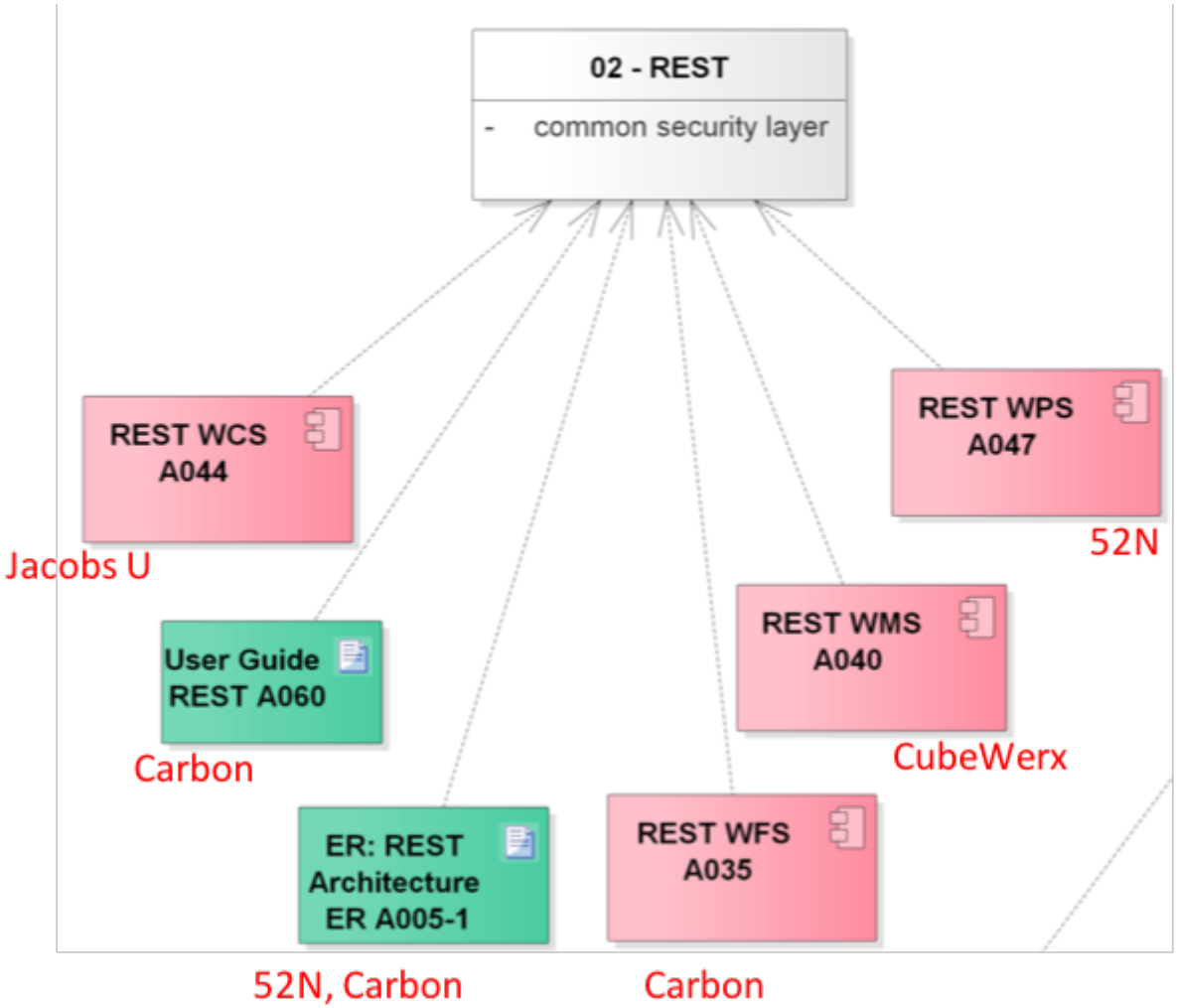


Figure 1. Overview on the REST deliverables in the Linked Data and Semantics (LDS) thread of Testbed 12

The ER is structured as follows: At first, clause 6 provides a general introduction in the concept of RESTful architectures and summarizes the advantages and disadvantages of using RESTful servers instead of e.g. POX-based Web Services. It also briefly sketches how existing OGC services may be mapped to RESTful APIs. The next five clauses describe the different REST server implementations realized in Testbed-12 for WFS (clause 7), WMS (clause 8), WMTS (clause 9), WCS (clause 10) and WPS (clause 11).

Afterwards, clause 13 summarizes the commonalities and differences of the different REST server implementations, before final recommendations are given in clause 14.

Chapter 6. General Considerations

The OGC activities related to REST architectures go back to 2009 when the first Engineering Report about REST and SOAP bindings for the Web Map Tile Server (WMTS) was published as result of an OWS-6 testbed activity. Since then, several activities have been implemented within and beyond OGC testbeds. Previous OGC documents that deal with REST are listed in Annex A.

Currently, the Web Map Tile Service (WMTS) specification (OGC 07-057r7) is the only OGC implementation specification explicitly defining a REST binding for an existing OGC service. Recently, the Sensor Things API (OGC 15-078r6) has been released as another official implementation standard defining a REST API. However, though it is relying on some concepts of the Sensor Observation Service, it has a different focus and there is no corresponding service specification.

This clause hence aims to lay the foundation for a common understanding of REST principles and to describe what needs to be done to build REST bindings for existing OGC Web Services. The open issues and recommendations are given in Clause 14.

6.1. REST Principles.

Representational State Transfer (REST) has been defined as an architectural style for distributed hypermedia systems by Roy Fielding in his doctoral dissertation[1]. The architectural style is defined by a set of constraints, which are described below. The core idea is that components of a REST-based architecture communicate primarily through the transfer of resource *representations* and additional control data which defines the actions upon these resources. In other words, the control data defines a set of pre-defined operations (usually HTTP verbs) to exchange and/or manipulate these representations, resulting in a uniform interface.

6.1.1. The Core Principle: Uniform Interface

The uniform interface is the central feature that distinguishes REST architectures from others. It is defined by four principles: indentification of resources, manipulation of resources through representations, self-descriptive messages and hypermedia as the engine of application state (HATEOS). Before explaining these constraints, we briefly define resources, identifiers and representations.

According to Fielding, a *resource* can be everything that can be named. Examples of resources in the geospatial domain are map tiles (served by a WMTS), features (served by a WFS), feature types (served by WFS), or observations (served by an SOS). Each resource needs to be identifiable by an *unique identifier*, usually a Uniform Resource Identifier (URI).

A *representation* describes the current or intended state of a resource at a certain point in time. Its encoding is defined by a media type. As an example, a geographic feature resource may be represented by a GeoJSON document describing the spatial location and other feature properties or by a JPEG image providing a rendered representation of the feature. The media type definition provides the metadata needed to encode or decode the different representations. Using a resource identifier and a media type, a client should be able to retrieve a representation of the resource.

The following principles apply now to the uniform interface:

- **Identification of resources:** Each resource should be identifiable by a URI and each REST server should be located under an Unified Resource Locator (URL). Together, the URI and the URL allow to uniquely identify a resource (comparable to a street name and a housenumber in a post address)
- **Manipulation of resources through representations:** If a client holds a resource representation, it has enough information to manipulate this representation by an additional request to the server, given it has the rights for manipulation.
- **Self-descriptive messages:** In a REST architecture, a message consists of control data, a resource identifier and an optional representation. This is enough information to process the request without additional information.
- **Hypermedia as the engine of application state (HATEOS):** This constraint requires that the client is simply following links in hypermedia, provided as URLs, to navigate from one system state to another. In other words, the client interacts with the server completely through hypermedia. The hypermedia may be dynamically changed by the server. In theory, no a-priori knowledge about interacting with the server is needed besides general knowledge of hypermedia.

6.1.2. Additional Principles of REST

In addition to the principles regarding the uniform interface described above, there are additional restrictions on the architecture. These are as follows:

- **Client-Server**: The user interface concerns (client) are seperated from the data storage concerns (server).
- **Stateless**: Communication between clients and servers should be stateless, i.e. each request from client to server must contain all necessary information to understand the request and should not rely upon previous requests.
- **Cache**: Responses should be labeled as cacheable or non-cacheable and hence enable caching of rather static resource representations.
- **Layered System:** A REST architecture should define a layered system. Component providers can hence just offer a single layer and hide other components of other layers.
- **Code-on-demand (optional)**: Servers may provide scripts within resource representations and hence allow for more light weight clients, e.g. embedding JavaScript code in an HTML representation.

NOTE	As an architectural style that abstracts from the WWW, REST does not require a specific protocol or, more precisely, the HTTP protocol. However, since the common protocol of the WWW is HTTP and most of the existing OGC Web Service specifications rely on HTTP, the remainder of this ER is focusing HTTP as the application protocol.

6.2. RESTful APIs

Web Service APIs complying to the REST constraints described above, are called RESTful APIs [3]. The vast majority of RESTful APIs available in the WWW are based on HTTP.

HTTP-based RESTful APIs are defined by:

- the base URL of the resources
- media type(s) used for the representation of the resources
- the HTTP methods that are applicable

Table 2 illustrates how HTTP methods are typically used in RESTful APIs.

Table 2. Usage of HTTP verbs for RESTful APIs (modified from [5])

Uniform Resource Locator (URL)	**GET**	**PUT**	**POST**	**DELETE**
Collection, such as http://api.example.com/resources/	Retrieve representation of collection	Replace entire collection	Create new entry in collection	Delete entire collection
Single resource representation, such as http://api.example.com/resources/resource_xy	Retrieve representation of resource	Replace representation	-	Delete resource

HTTP GET is used for the retrieval of information. PUT can be used to update existing resource representations or collections. POST is used to create new resource representations and DELETE is used for deleting resources. In some cases, POST is used to create a new resource representation for a specific URL. However, this requires, that the client is (i) allowed to specify the resource identifiers and (ii) knows that no other resource is identified by the request URL.

Instead of specifying specific response codes, RESTful APIs utilize the common HTTP Status Codes (e.g. 200 for OK, 400 for bad request, etc.).

6.3. Richardson Maturity Model

The REST APIs available in the Web are varying regarding the constraints that are supported. For example, not all REST APIs are supporting hypermedia. The Richardson Maturity Model (RMM) defines different levels indicating how strict a Web service is following the REST principles ranging from level 0 (not strict) to level 3 (strict). The different levels are listed in Table 3.

Table 3. The levels of Richardson's REST Maturity Model

Level	Properties
0	uses XML-RPC or SOAP; service is identified by single URI; uses single HTTP method (often POST)
1	uses different URIs and resources; uses single HTTP method (often POST)
2	uses different URIs and resources; uses several HTTP methods
3	supports HATEOAS and thus uses hypermedia for navigation; uses different URIs and resources; uses several HTTP methods

6.4. Advantages and Disadvantages of using REST

The advantages and disadvantages of using REST are liste in Table 4.

Table 4. Advantages and Disadvantages of REST architectures

Advantages	Disadvantages
Simplicity; Reliability; Scalability; Performance; Caching; Visibility of Communication; Portability	Simplicity; Increased Network Traffic; Custom APIs and clients

The probably most significant advantage of implementing REST APIs with hypermedia support is simplicity due to the uniform interface constraint. As most REST APIs utilize the HTTP verbs and JSON as hypermedia encoding, usual Web browsers supporting JSON formatting may provide a first entry point. More sophisticated thin Web clients may easily be build, because JSON is the default serialization format for JavaScript applications. However, keeping the interface specification quite general may also result in a variety of different REST APIs for the same purpose and dedicated clients for these APIs resulting in decreased interoperability. To address this issue, a set of recommendations has been derived in Clause 14.

Being stateless, REST improves reliability, as requests resulting in errors may be simply re-sent, and scalability, as no additional status information needs to be stored and managed. As a potential drawback, network traffic may be increased due to additional requests.

Another advantage is the ability for clients to cache representations improving performance and decreasing network traffic. However, if the representations are dynamically changing, this may become a drawback since information on the client side may be outdated or the update of representations may require additional client requests.

6.5. From an Service-oriented to a RESTful OGC Architecture

The geospatial services specified by the OGC are following a Service-oriented Architecture (SOA). The specifications thus describe the access and management of spatial data in terms of service operations. Resource-oriented Architectures (ROA) are defined from a different viewpoint: Instead of specifying service operations, a ROA focuses on the resources that should be managed by a server. In order to retrieve, create, update or delete these resources, the general HTTP methods (GET, POST, PUT, DELETE) are applied to these resources .

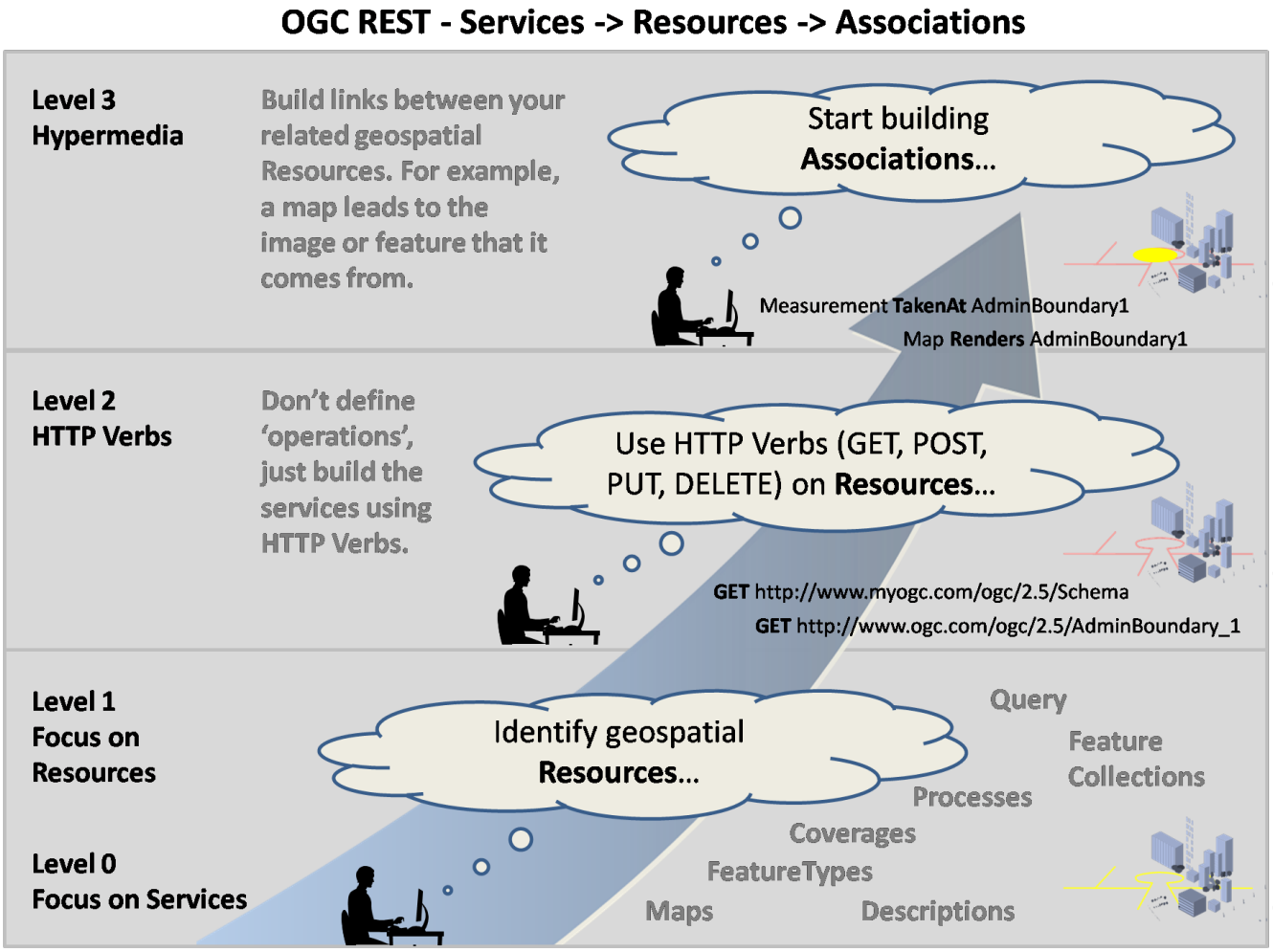


Figure 2. OGC REST - From Services to Resources

The transition from spatial data that is served by OGC Web Services towards resources offered in a OGC REST architecture is illustrated in Figure 2. In a first step, the service interfaces need to be translated to REST APIs, which includes the transition from Level 0 to Level 2 in Richardson Maturity Model. This boils down to to the following steps:

1. Identifying resources and associations between them: What resources should be managed? What kind of associations exist between these resources and how could they be represented?
2. Specification of the API: What (HTTP) operations are applicable on the resources? What is the main service endpoint and how are the resources accessible from this endpoint?

Unfortunately, as we will see in the chapters below, the translation is not as straight-forward as it may appear, since the OGC services have been specified from a service-oriented viewpoint focusing on operations rather than on resources. Therefore, step 1 is essential and needs to be executed with care. In order to illustrate the required steps, we use the OGC Sensor Observation Service (SOS; OGC 12-006), which provides access to sensor observations and sensor descriptions and transactional operations on them, as an example. In case of a consequent usage of hypermedia, the services as known in the SOA are no longer needed. This is explained in the last subsection of this chapter.

6.5.1. Identification of Resources

The SOS provides access to observations based on the Observations & Measurements (O&M) model (OGC) and sensor descriptions based on the Sensor Model Language (SensorML; OGC 12-000).

Therefore, the questions regarding the resources and associations in step 1 may be answered by analyzing these underlying data models. Figure 3 shows a strongly simplified version of the basic resource model underlying the SOS.

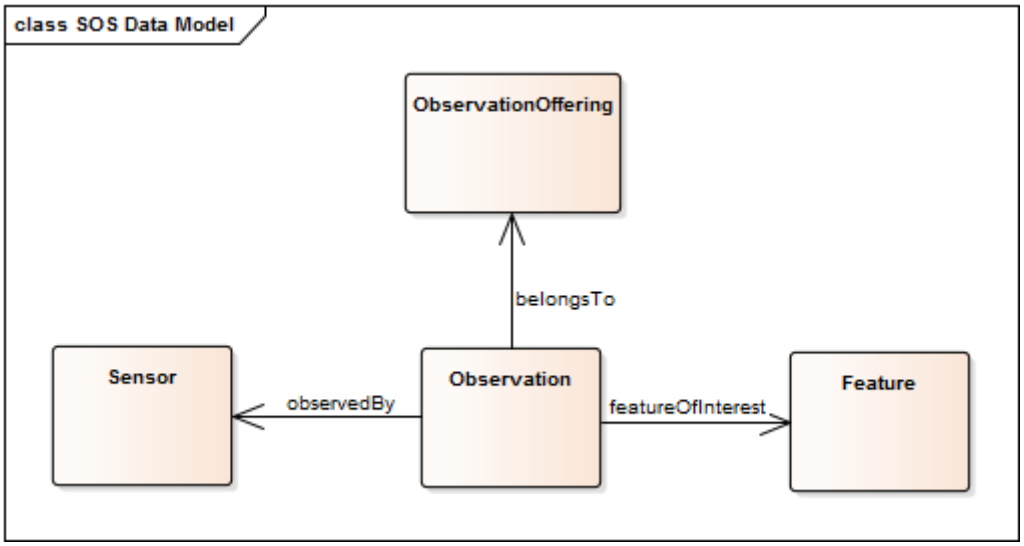


Figure 3. Simplified resource model of the Sensor Observation Service

In a nutshell, an observation is made on a feature of interest and has been observed by a certain sensor. An observation belongs to an ObservationCollection which aggregates single observations.

The supported media types may depend on the encodings that are specified for the data models. In case of O&M, an XML binding has been specified (OGC 10-025r1) and a JSON encoding is currently available as a discussion paper (OGC 15-100r1). For SensorML, only an XML encoding is currently specified.

6.5.2. Specification of the API

Once the core resources that should be offered are identified, the RESTful API needs to be specified. This involves identifying which HTTP methods can be applied to which resources and defining the base URL of the endpoint of the API. Let the endpoint of the a RESTful SOS API be as follows:

```
https://my.sos.url/rest-api
```

In order to ensure that the functionality offered by the original OGC service is available, it may be useful to take a look on the operations that are specified for the OGC service interface. In case of SOS they look as shown in Figure 4. For illustration purposes, we focus on the core and transactional operations.

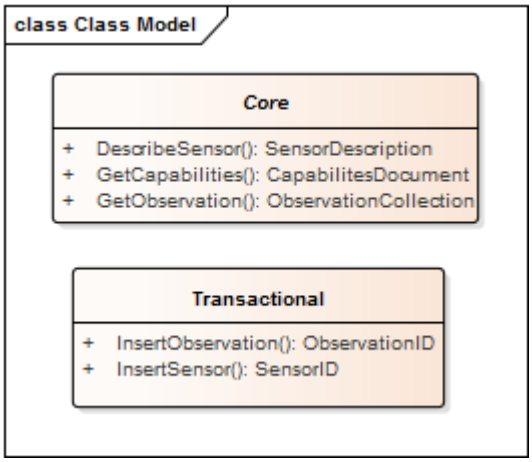


Figure 4. Operations of Core and Transactional profile of Sensor Observation Service

The SOS core profile defines the operations for observation and sensor description retrieval. The Transactional profile specifies the operations for the insertion of new sensors and observations. In order to provide this functionality in the REST API, we thus need to specify, which HTTP operations need to be applied on which resources.

Table 5 illustrates how the DescribeSensor and GetObservation operations can be realized using HTTP methods on observation and sensor resources and corresponding collections. Since the operations only define the retrieval of resources, the only HTTP method used is HTTP GET.

Table 5. HTTP methods on resources for implementing a RESTful API for the SOS Core profile

HTTP Method	resource endpoint	parameters	description
GET	https://my.sos.url/rest-api	NA	returns the Capabilities document
GET	https://my.sos.url/rest-api/observations	NA	returns an observation collection
GET	https://my.sos.url/rest-api/observations/obs_id_1	NA	returns a single observation representation
GET	https://my.sos.url/rest-api/sensors	NA	returns a sensor collection
GET	https://my.sos.url/rest-api/sensors/sensor_id_1	NA	returns a single sensor representation

How can the feature of interest now be retrieved? The XML encoding of O&M already allows for the usage of XML links. Hence, the feature of interest is already referenced from an observation and may be served by a Web Feature Service. As such, O&M XML may already be considered as hypermedia.

The transactional operations of SOS enable the insertion of new sensors and observations. Thus the HTTP method used is HTTP POST and should be applied on the sensor or observation collections.

Table 6. HTTP methods on resources for implementing a RESTful API for the SOS Transactional profile

HTTP Method	resource endpoint	parameters	description
POST	https://my.sos.url/rest-api/observations	observation representation	inserts new observation and returns assigned URI for new observation
POST	https://my.sos.url/rest-api/sensors	sensor representation	inserts new sensor and returns assigned URI

NOTE As can be seen in the following clauses, there are still open issues left for defining REST bindings for the OGC services. For example, the Capabilities cannot be easily used to describe the REST APIs, as the operations metadata section was originally designed for service operations and currently only HTTP GET and POST are supported. The issues and recommendations are listed below in Clause 14.

6.5.3. Hypermedia replacing the OGC services?

If all geospatial resources would be provided as hypermedia in an OGC REST architecture, the classical division in OGC services providing operations for different data types, i.e. WCS providing operations on coverages, WMS on maps, WFS on features, and so forth, would no longer be needed. Coming back to our example of the provision of observations, the observations may be linked to other spatial resources as illustrated in Figure 5. In a traditional service-oriented OGC architecture, the observations, observation collection and sensor metadata would be served by a SOS (grey classes), the feature by a WFS (blue class), the spatial coverage representing an interpolated field from the observation collection would be served by a WCS (yellow class), and the observations may be rendered with the feature of interest on a map served by a WMS (orange classes). Another map would be rendered from the spatial coverage and the features of interest.

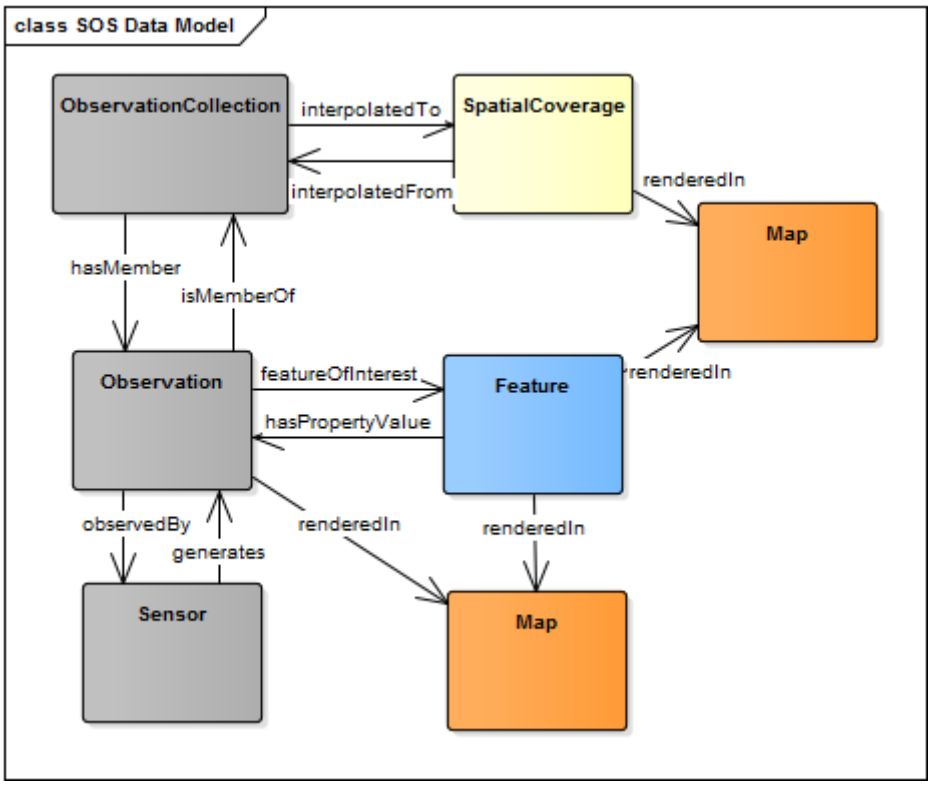


Figure 5. Observations and linked spatial resources provided as hypermedia.

Now, if all resources would be served as hypermedia, a single entry point, e.g. to observations would be sufficient to follow the links to the other resources provided which would implement the HATEOAS constraint. For example, one would be able to navigate from the observation to the collection it belongs to, from the collection to the spatial coverage and from the coverage to the map to which the coverage is rendered to. The underlying data may be serialized in RDF or JSON-LD (see also the JSON ER and User Guide provided in this testbed).

However, as we will see below, most of current OGC REST implementations are implementing RMM Level 2, meaning that no hypermedia is provided. Hence, in the long term it may be possible to go beyond the borders between services in future OGC REST architectures, we consider translating the OGC services to REST APIs as a sensible intermediate step.

Chapter 7. WFS REST Server

This section presents Testbed 12 experience using open geospatial REST with WFS and JSON. The resulting JSON WFS REST API (A035) was able to represent real world phenomena as open geospatial features. The features were deployed as open geospatial Resources that can be accessed and updated using the language of the World Wide Web.

This section consists of the following sub-sections:

- Background - WFS, NAS GML and GeoJSON
- GeoJSON WFS REST

7.1. Background

Technology Integration Experiments in this segment of Testbed 12, participants brought together three aspects of Testbed 12 -

- WFS
- NAS GML
- GeoJSON

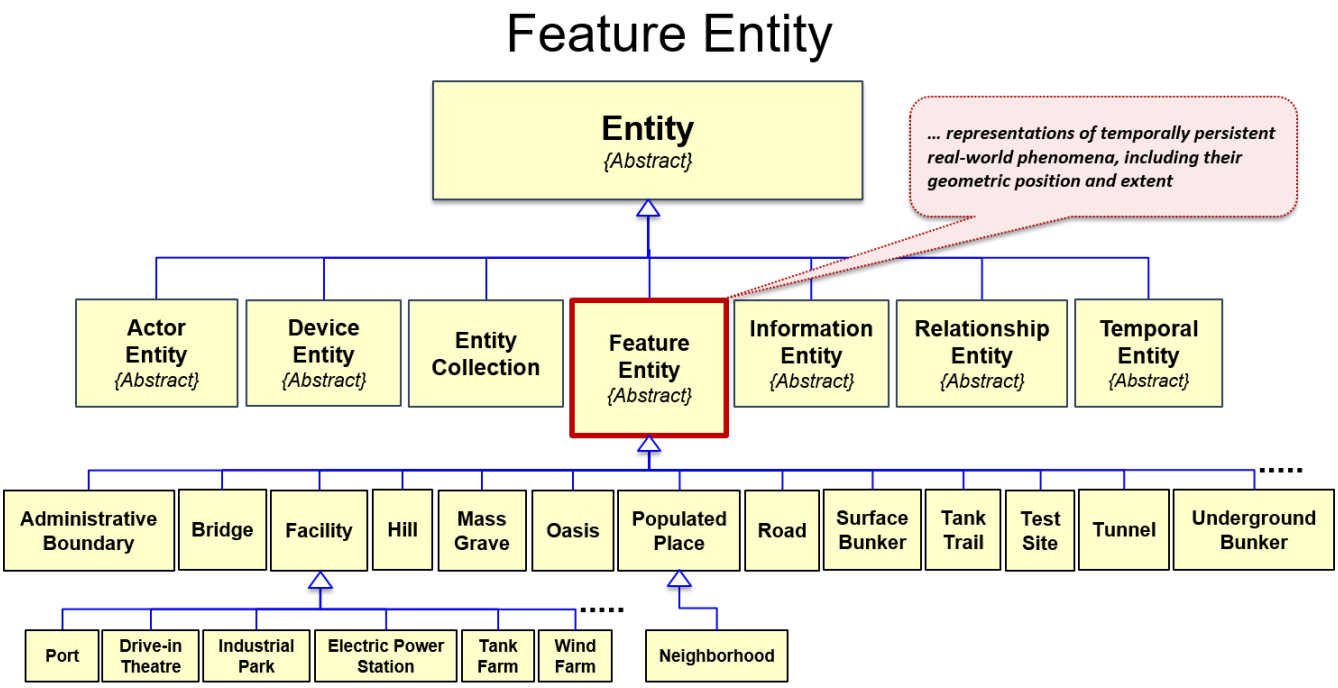
7.1.1. WFS

The OGC Web Feature Service (WFS) Implementation Specification allows a client to retrieve geospatial data encoded in Geography Markup Language (GML) and other formats from multiple Web Feature Services. The specification defines operations for data access and manipulation operations on geographic features, using HTTP as the distributed computing platform. Via these interfaces, a Web user or service can combine, use and manage geodata — the feature information behind a map image. Relevant WFS operations for this segment of Testbed 12 include -

- Query operations allow features or values of feature properties to be retrieved from the underlying data store based upon constraints, defined by the client, on feature properties.
- Transaction operations allow features to be created, changed, replaced and deleted from the underlying data store.

7.1.2. NAS GML

NSG Application Schema (NAS) is a Platform Independent Model that defines the GEOINT exchange semantics for the US National System for Geospatial-Intelligence (NSG). For Testbed 12, NAS GML was provided to test the ability of interoperable components to support the model, in particular the Entity portion of NAS -



For Testbed 12 Adminstrative Boundary features in NAS GML were provided to participants for testing.

7.1.3. GeoJSON

GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON). It defines several types of JSON objects and the manner in which they are combined to represent data about geographic features, their properties, and their spatial extents. GeoJSON uses a geographic coordinate reference system, World Geodetic System 1984, and units of decimal degrees.

GeoJSON object may represent a region of space (a Geometry), a spatially-bounded entity (a Feature), or a list of features (a Feature Collection). GeoJSON supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Features in GeoJSON contain a geometry object and additional properties, and a Feature Collection that contains a list of features.

GeoJSON features are this combination of geometry and properties:

```
{

 "type": "Feature",
 "geometry": {
  "type": "Point"'
  "coordinates": [0, 0]

 },

 "properties": {
  "name": "admin boundary"

 }
}
```

The properties attached to a feature can be any kind of JSON object.

7.2. WFS REST

7.2.1. Resources to be provided

The resources that are provided by a WFS REST server are listed in Table 7 below and include the capabilities document of the server, features, queries, transactions and certain metadata.

Table 7. Resources provided by the WFS REST server

Resource Class	Description	Acccess Path
Capabilities	The complete service metadata document.	{WfsRESTBaseURL}
Schema	The complete application schema offered by the server.	{schema URL}
Feature Type	A feature type (i.e. a named collection of features with the same schema)	{ftype schema URL}
Feature	A feature (i.e. a member of a feature type)	{feat URL}
Property of a feature type	A named property from the schema of a feature type	{ftype URL}/{prop} (See note 1)
Property of a feature	A named property from the schema of a feature	{feat URL}/{prop} (See note 1)
Query	A complex query resource	/query
Transaction	A complex transaction resource	/transaction

Resource Class	Description	Acccess Path
Notations: 1. The property name is appended before any query parameters 2. {schema URL}: The URL to the application schemas that the service offers; this URL is specified in the capabilities document of the service using an atom:link element with rel="describedBy" 3. {ftype schema URL}: The URL to a schema document declaring the schema of the specified feature type; this URL is specified at the first nesting level within the wfs:FeatureType element in the capabilities document using an atom:link element with rel="describeBy" 4. {ftype URL}: The URL to the collection of feature of this type; it is specified at the first nesting level within the wfs:FeatureType element in the capabilities document using an atom:link element with the rel="collection" 5. {feat URL}: The URL of a feature; it is specified in the response to a query within the feature container element (i.e. wfs:member element for an XML encoded response) using an atom:link element with rel="self" 6. {prop}: The name of a property of a feature or feature type; appended to the resource URL before any query parameters.		

7.2.2. Operations on resources

Table 8. Operations on resources provided by WFS REST server

HTTP Operation	Access Path	Equivalent WFS Operation
GET	/?...	GetCapabilities
	{schema URL}	
	{ftype schema URL}	DescribeFeatureType
	{ftype URL}?...	GetFeature
	{query URL}?...	GetFeature
POST	{ftype URL}?...	Transaction
PUT	(feat URL}?...	Transaction
	{feat URL}/{prop}?...	Transaction
DELETE	{feat URL}?...	Transaction
	{feat URL}/{prop}?...	Transaction

It is important to note that emerging WFS drafts define four REST conformance classes:

- Simple Query
- Simple Transaction
- Complex Query
- Complex Transaction

The "Simple" conformance classes handle single feature type operations where the resource is the feature collection (i.e. the feature type) and the operations are GET, POST, PUT & DELETE.

All classes are optional so people who don't need or want complex query and transaction handling don't have to implement those Resources … but if you want to support joins and atomic transactions you will need to.

7.3. Implementation

In Testbed 12 Open Geospatial REST, WFS, NAS GML and GeoJSON were integrated by The Carbon Project to deploy a GeoJSON WFS REST.

- **GML**

http://ows12.azurewebsites.net/wfs/featuretypes/AdministrativeSubdivision

- **JSON**

http://ows12.azurewebsites.net/wfs/featuretypes/AdministrativeSubdivision?outputFormat=json

Note: In most JSON WFS deployments there may be only JSON output format and the request would be directed to the Administrative Subvision feature Resource. Simple query and transactions were tested with a focus on an initial assessment of NAS GML as GeoJSON.

7.3.1. Request GeoJSON

Basic request

```
GET /featuretypes/AdministrativeSubdivision?outputFormat=json
HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type:
application/vnd.geo+json

{ "type": "FeatureCollection", "features": [{"type":"Feature","_id":"1",,"geometry":{
"type": "Polygon", "coordinates": [ [ [-122.438548585768, 37.7527347148009], [-
122.438712218419, 37.7542478576584], [-122.437694850195, 37.75428723271], [-
122.437588133248, 37.7528134664936], [-122.438548585768, 37.7527347148009] ] ]
},"properties":{
  "nas:uniqueEntityIdentifier": "UUID-1-132904",
  "nas:place": {
    "nas:FeaturePlaceInfo": {
      "nas:uniqueEntityIdentifier": "UUID-1-132904",
      "nas:place": {}
    }
  },
  "nas:restriction": {
    "nas:RestrictionInfo": {
      "nas:uniqueEntityIdentifier": "UUID-1-here",
      "nas:commercialCopyrightNotice": {
        "nas:TextLexUnconMeta": {
```

```
          "nas:valueOrReason": "Copyright National Geospatial-Intelligence Agency"
        }
      },
      "nas:securityAccessGroup": {
        "nas:RestrictionInfoSecurityAttributesGroupMeta": {
          "nas:valueOrReason": {
            "attributes": [
              {
                "icism:classification": "U"
              },
              {
                "icism:ownerProducer": "USA"
              }
            ],
            "value": null
          }
        }
      }
    }
  },
  "nas:area": {
    "nas:RealNonNegMeta": {
      "nas:restriction": {
        "nas:RestrictionInfo": {
          "nas:uniqueEntityIdentifier": "UUID-1-here",
          "nas:commercialCopyrightNotice": {
            "nas:TextLexUnconMeta": {
              "nas:valueOrReason": "Copyright National Geospatial-Intelligence Agency"
            }
          },
          "nas:securityAttributesGroup": {
            "nas:RestrictionInfoSecurityAttributesGroupMeta": {
              "nas:valueOrReason": {
                "attributes": [
                  {
                    "icism:classification": "U"
                  },
                  {
                    "icism:ownerProducer": "USA"
                  }
                ],
                "value": null
              }
            }
          }
        }
      }
    }
  },
  "nas:bgnAdminLevel": {
    "nas:AdministrativeSubdivisionBgnAdminLevelCodeMeta": {
```

```
      "nas:valueOrReason":
"http://api.nsgreg.nga.mil/codelist/BgnAdminLevel/firstOrder"
    }
  },
  "nas:designation": {
    "nas:AdminSubdivisionDesig": {
      "nas:uniqueEntityIdentifier": "UUID-1-132904",
      "nas:gencPreferredName": {
        "nas:TextLexUnconMeta": {
          "nas:valueOrReason": "California"
        }
      },
      "nas:gencShortUrnBasedIdentifier": {
        "nas:GencShortUrnBasedIdentifierTextMeta": {
          "nas:valueOrReason": "ge:GENC:3:3-3:US-CA"
        }
      }
    }
  },
  "nas:principalSubdivisionOf": {
    "nas:GeopoliticalEntity": {
      "nas:uniqueEntityIdentifier": "ge:GENC:3:3-3:USA",
      "nas:boundary": null,
      "nas:designation": {
        "nas:GeopoliticalEntityDesig": {
          "nas:uniqueEntityIdentifier": "UUID-1-132904",
          "nas:gencShortUrnBasedIdentifier": {
            "nas:GencShortUrnBasedIdentifierTextMeta": {
              "nas:valueOrReason": "ge:GENC:3:3-3:US-CA"
            }
          }
        }
      }
    }
  }
}}]}
```


In the example above, the response is a combination of geometry and properties, with the geometry type of Polygon.

Since this is GeoJSON only EPSG code 4326 should be returned.

7.3.2. NAS GML Attributes

There is currently no guidance for representing attributes in a GeoJSON WFS so they were handled in the following manner:

```
"nas:valueOrReason": {
 "attributes": [
  {
    "icism:classification": "U"
  },
  {
    "icism:ownerProducer": "USA"
  }
  ],
   "value": null
  }
```

7.3.3. Create GeoJSON FeatureType

```
POST /featuretypes/AdministrativeSubdivision?outputFormat=json
HTTP/1.1
Content-Type:
application/vnd.geo+json

{
   "type":"Feature",
   "geometry":{
      "type":"Polygon",
      "coordinates":[
         [
            [
               -122.478531156081,
               37.7394760665792
            ],
            [
               -122.478795146849,
               37.7409374031664
            ],
            [
               -122.477965461578,
               37.7409374031664
            ],
            [
               -122.477739183777,
               37.7394760665792
            ],
            [
               -122.478531156081,
               37.7394760665792
            ]
         ]
      ]
   },
   "properties":{
```

```
      "nas:uniqueEntityIdentifier":"UUID-1-132904",
      "nas:place":{
         "nas:FeaturePlaceInfo":{
            "nas:uniqueEntityIdentifier":"UUID-1-132904",
            "nas:place":{
               "nas:featureEntity":null
            }
         }
      },
      "nas:restriction":{
         "nas:RestrictionInfo":{
            "nas:uniqueEntityIdentifier":"UUID-1-here",
            "nas:commercialCopyrightNotice":{
               "nas:TextLexUnconMeta":{
                  "nas:valueOrReason":"Copyright National Geospatial-Intelligence
Agency"
               }
            },
            "nas:securityAttributesGroup":{
               "nas:RestrictionInfoSecurityAttributesGroupMeta":{
                  "nas:valueOrReason":null
               }
            }
         }
      },
      "nas:area":{
         "nas:RealNonNegMeta":{
            "nas:valueOrReason":"415836540000",
            "nas:restriction":{
               "nas:RestrictionInfo":{
                  "nas:uniqueEntityIdentifier":"UUID-1-here",
                  "nas:commercialCopyrightNotice":{
                     "nas:TextLexUnconMeta":{
                        "nas:valueOrReason":"Copyright National Geospatial-
Intelligence Agency"
                     }
                  },
                  "nas:securityAttributesGroup":{
                     "nas:RestrictionInfoSecurityAttributesGroupMeta":{
                        "nas:valueOrReason":null
                     }
                  }
               }
            }
         }
      },
      "nas:bgnAdminLevel":{
         "nas:AdministrativeSubdivisionBgnAdminLevelCodeMeta":{

"nas:valueOrReason":"http://api.nsgreg.nga.mil/codelist/BgnAdminLevel/firstOrder"
         }
```

```
      },
      "nas:designation":{
         "nas:AdminSubdivisionDesig":{
            "nas:uniqueEntityIdentifier":"UUID-1-132904",
            "nas:gencPreferredName":{
               "nas:TextLexUnconMeta":{
                  "nas:valueOrReason":"California"
               }
            },
            "nas:gencShortUrnBasedIdentifier":{
               "nas:GencShortUrnBasedIdentifierTextMeta":{
                  "nas:valueOrReason":"ge:GENC:3:3-3:US-CA"
               }
            }
         }
      },
      "nas:principalSubdivisionOf":{
         "nas:GeopoliticalEntity":{
            "nas:uniqueEntityIdentifier":"ge:GENC:3:3-3:USA",
            "nas:boundary":null,
            "nas:designation":{
               "nas:GeopoliticalEntityDesig":{
                  "nas:uniqueEntityIdentifier":"UUID-1-132904",
                  "nas:gencShortUrnBasedIdentifier":{
                     "nas:GencShortUrnBasedIdentifierTextMeta":{
                        "nas:valueOrReason":"ge:GENC:3:3-3:US-CA"
                     }
                  }
               }
            }
         }
      }
   }
}
```

```
HTTP/1.1 201 Created
```

Additional examples such as changing the value of a property would use PUT method etc.

7.4. Lessons Learned

Based on the experiences of Testbed 12, a GeoJSON WFS API may have significant potential to represent real world phenomena as open geospatial features as open geospatial REST Resources that can be accessed and updated using the language of the World Wide Web.

In the future, REST WFS may represent the Capabilities Resource as JSON Schema instead of current XML. There currently is no guidance for such metadata descriptions.

Chapter 8. WMS REST Server

The WMS (A040) work item for Testbed 12 involved investigating and implementing the use of GeoJSON as an output format for the WMS GetFeatureInfo operation as described in OGC 15-053, “Testbed 11 Implementing JSON/GeoJSON in an OGC Standard Engineering Report”. Specifically, this work item was concerned with recommendation 28 from OGC 15-053 which proposed including an encoding for GetFeatureInfo responses based on GeoJSON. Recommendation 28 further proposed replacing the geometry part by a marker of the position of the query and the position of the returned feature. If the returned objects correspond to simple features, an identifier is to be included in the response that allows recovering the geometry using an additional WFS query. The Testbed 12 RFP proposed using the proposed WMS 1.4 draft specification as the base service standard for this work. The state of the WMS 1.4 draft specification, however, is such that it would not be possible to implement a service with the WMS 1.4 API within the time and resource allocated for this work item in Testbed 12. Instead, the existing and more stable WMS 1.3 (OGC 06-042) server was proposed and used for this work item.

8.1. Resources to be provided

We can infer from the purpose of the GetFeatureInfo operation that the relevant resource is a “feature” and that a GeoJSON response would be considered a representation of that feature.

NOTE	The current draft WMS 1.4 specification (which seems to deprecate an existing draft WMS 2.0 version based on the modification dates of the documents) discusses REST only in a peripheral way and not related to the GetFeatureInfo operation at all.

8.2. Operations on resources

The WMS is a data portrayal service and as such, the only HTTP method used is the GET method which is used to retrieve the resources provided by the service (i.e. maps and feature info).

8.3. Implementation

A WMS 1.3 server supporting GeoJSON as a GetFeatureInfo output format was deployed for Testbed 12 here:

http://tb12.cubewerx.com/a040/cubeserv?SERVICE=WMS&VERSION=1.3.0&REQUEST=GetCapabilities

The following example retrieves GeoJSON output is response to a GetFeatureInfo request:

Example of HTTP GET request for retrieving a feature info resource (NOTE: Request has been line-wrapped for easier reading).

```
http://tb12.cubewerx.com/a040/cubeserv?SERVICE=WMS&
      VERSION=1.3.0&
      LANGUAGE=en-CA,en&
      REQUEST=GetFeatureInfo&
      CRS=EPSG%3A3857&
      BBOX=-13641185.826,4546529.053,-13601935.412,4575116.502&
      WIDTH=1027&HEIGHT=748&
      LAYERS=USGS.Struct_Point&
      FORMAT=image%2Fx-jpegorpng&
      QUERY_LAYERS=USGS.Struct_Point&
      FEATURE_COUNT=10&
      I=812&
      J=345&
      INFO_FORMAT=application%2Fjson
```

The following GeoJSON document is retrieved by this URL:

Example of a feature info resource encoded as GeoJSON.

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "fsetName": "Struct_Point",
      "geometry": {
        "type": "Point",
        "coordinates": [
          -122.2622829998311,
          37.87610699954206
        ]
      },
      "properties": {
        "PERMANENT_": "{F154F04D-FD02-4D8E-9B8E-E122331C931C}",
        "SOURCE_FEA": "115214",
        "SOURCE_DAT": "{B04D081F-2258-4584-B950-3659EE6688B8}",
        "SOURCE_D_1": "Schools ORNL nationwide schools dataset USGS provisional load",
        "SOURCE_ORI": "Oak Ridge National Laboratory, Geographic Information Sciences
and Technology Group",
        "DATA_SECUR": 5,
        "DISTRIBUTI": "E3",
        "LOADDATE": "2011-12-20 00:00:00.000 +00:00",
        "FTYPE": 730,
        "FCODE": 73006,
        "NAME": "Graduate Theological Union",
        "ISLANDMARK": 2,
        "POINTLOCAT": 1,
```

```
        "ADMINTYPE": null,
        "ADDRESSBUI": null,
        "ADDRESS": "2400 Ridge Rd",
        "CITY": "Berkeley",
        "STATE": "CA",
        "ZIPCODE": "94709-",
        "GNIS_ID": null,
        "FOOT_ID": null,
        "COMPLEX_ID": null
      }
    },
    {
      "type": "Feature",
      "fsetName": "Struct_Point",
      "geometry": {
        "type": "Point",
        "coordinates": [
          -122.2614299998325,
          37.87587699954241
        ]
      },
      "properties": {
        "PERMANENT_": "{D8BFD704-2104-4EC2-A725-6CB66A6E7B97}",
        "SOURCE_FEA": "112127",
        "SOURCE_DAT": "{B04D081F-2258-4584-B950-3659EE6688B8}",
        "SOURCE_D_1": "Schools ORNL nationwide schools dataset USGS provisional load",
        "SOURCE_ORI": "Oak Ridge National Laboratory, Geographic Information Sciences
and Technology Group",
        "DATA_SECUR": 5,
        "DISTRIBUTI": "E3",
        "LOADDATE": "2011-12-20 00:00:00.000 +00:00",
        "FTYPE": 730,
        "FCODE": 73006,
        "NAME": "Church Divinity School of the Pacific",
        "ISLANDMARK": 2,
        "POINTLOCAT": 1,
        "ADMINTYPE": null,
        "ADDRESSBUI": null,
        "ADDRESS": "2451 Ridge Road",
        "CITY": "Berkeley",
        "STATE": "CA",
        "ZIPCODE": "94709-1211",
        "GNIS_ID": null,
        "FOOT_ID": null,
        "COMPLEX_ID": null
      }
    },
    ...
  ],
  "bbox": [
    -122.2624979998308,
```

```
    37.875506999543,
    -122.2614299998325,
    37.87680799954097
  ]
}
```

NOTE	Normally, packed JSON output is generated from by the A040 server but the output presented here has been wrapped and truncated to make it easier to read.

8.4. Lessons Learned

Providing a GeoJSON response for WMS GetFeatureInfo requests makes a lot of sense, since more and more client applications are web-browser based and JSON is the more natural representation in this environment. However, OGC 15-053 imposes certain complications that were questioned, and ultimately discarded, in the implementation of A040.

OGC 15-053 states that "we are not recommending returning the full geometry description because we assume that this is the work of a WFS GetFeature operation using identifiers recovered by the GetFeatureInfo request". The justification for suggesting that returning a geometry is out of scope for a WMS GetFeatureInfo response was not clear. The entire point of having a WMS GetFeatureInfo request is to:

1. give a WMS client the ability to query information about a feature without having to implement any WFS logic
2. give a WMS server the ability to provide information about a feature without having to implement a companion WFS

If a WMS client is interested in the geometry of a feature, it's likely for the purpose of plotting it on the map to highlight the feature(s) that the user has clicked on. Wouldn't it be convenient if the GeoJSON GetFeatureInfo response could include the geometry as part of the response? Of course, GeoJSON can do exactly that so we saw no reason for needlessly complicating the situation by bringing a WFS into the picture.

Furthermore, returning a vector pointing from the user click location (which client already knows) to "a point that is in the interior or the border of the returned feature" is completely useless to the client. Then requiring the client to somehow determine the association to the appropriate WFS server and WFS feature type (see discussion below), formulate and issue an appropriate WFS GetFeature request, and be equipped to handle the WFS response, is a lot to ask of a simple WMS client especially in the case where the client simply wants to highlight a feature on the map.

The requirements imposed on a WMS server by this aspect of requirement 28 are also overly onerous. The requirement is basically suggesting that if a WMS wants to be able to serve geometries to its client — so that it can highlight them on a map for example — then the WMS must also include an implementation of a WFS server to handle that. This all seems very heavyweight for no good reason, especially since GeoJSON responses are naturally equipped to return the geometries right from the start.

If the concern is that returning geometries in a WMS GetFeatureInfo response uses unnecessary

bandwidth for client applications that don't require the geometries, then our position is that using a bit of extra bandwidth like this is a small price to pay for the simplicity of the interface. The addition of two optional GetFeatureInfo parameters (Table 8) is proposed to streamline this:

Table 9. Additional proposed GetFeatureInfo request parameters

Parameter Name	Values	Description
GET_GEOMETRIES	Boolean	Indicates whether or not geometries should be returned in the GetFeatureInfo response. If FALSE, then (at least in the case of GeoJSON output) the geometry should be returned as null. If unspecified, it's at the server's discretion. The server is not obligated to honour this parameter.
SIMPLIFY_GEOMETRIES	Boolean	Indicates whether or not geometries returned in GetFeatureInfo response should be simplified to the resolution of the corresponding GetMap request. If unspecified, the default value is FALSE. The server is not obligated to honour this parameter.

The deployed TB12 implementation of the GetFeatureInfo operation includes these two parameters.

Finally, none of this is to say that returning the WFS feature identifier of the feature as part of the WMS GetFeatureInfo response is necessarily a bad thing. In fact, it might be useful to complex clients that really do need full WFS access to the feature (for example, to perform updates on it, etc.). It's worth noting that a WFS feature identifier alone is not enough. The client application would have to also know:

1. The base URL of the association WFS server (it's not necessarily the same as the base URL of the WMS server being accessed)
2. The name of the associated WFS feature type (it's not necessarily the same as the name of the WMS layer being accessed)
3. The namespace of the associated WFS feature type (there is not way to deduce this from the WMS interface)

The mechanism(s) to allow the determination of these associations was certainly beyond the scope of OGC 15-053r1. However, OGC 16-043, "Testbed-12 Web Integration Service Engineering Report", proposes just such a mechanism in Clause 6, "GetAssociations operation".

8.5. Speculations on a RESTful API from WMS

8.5.1. Introduction

The question has been asked about whether the WMTS can be considered as the REST API for the WMS? The real question, however, is why is a WMS necessary now that we have a WMTS? For many situations the WMTS can indeed act as a RESTful replacement from the WMS. However, the answer to this question comes down to one of flexibility (WMS) versus scalability (WMTS).

The WMS can serve maps of arbitrary size, at arbitrary scales, in arbitrary coordinate system and with arbitrary styles and in a variety of output formats.

The WMST, on the other hand, serves images composed of tiles at very specific scales, with very specific tile boundaries, in very specific coordinate systems and in very specific output formats. In order to approach the flexibility of a WMS, a WMTS would need to generate a tile cache for each offered CRS, in each offered style, at each offered scale, for each offered output format, etc. Clearly, this quickly becomes an unmanageable combinatorial and space problem.

So, assuming that that WMTS is not the REST API for a WMS, the question then becomes, what does a RESTful API for WMS look like?

This clause does not attempt to answer this question but rather provides some speculations for consideration when designing a RESTful API for the WMS. Specifically, the discussion will speculate about:

1. the list of resources of a RESTful WMS
2. query parameters that may be used with those resources
3. discovery considerations (i.e. how clients should determine those resource URLs)

8.5.2. List of resources

The first task for designing a RESTful API is to establish what the resources are available. Some examples of possible basic resources might include:

1. service metadata
    - i.e. the capabilities document
2. layer
    - a basic unit of geographic information that may be requested as a map from a WMS
3. map
    - an order list of layers
4. view
    - a resource to control how a map is rendered onto a screen; (e.g. where the map is centered, what zoom level is being displayed what projection is being used, etc.)
5. feature
    - a resource for feature-information queries

This is by no means meant to be an exhaustive list of WMS resources. Other aspects of a WMS might also be meterialized as resource; for example, styles or map legends. This list is meant only to stimulate discussion in future test beds and in the WMS standards working group.

8.5.3. Query parameters

Once the set of resource have been identified the next task would be to identify the query parameters that may be used with those resource. Assuming the list of resources proposed in the previous clause, the following parameters might be relevant:

1. layer parameters
    a. style
2. map parameters
    a. output format
    b. transparent
    c. back ground colour
    d. exceptions format
    e. time
    f. elevation
3. view parameters
    a. bbox
    b. crs
    c. height
    d. width

8.5.4. Discovery

Once the resources and parameters have been defined, there will be a need to consider how a client should determine the resource URLs. Some possibilities might be:

- A URL-template solution similar to WMTS; the capabilities document would need to report the templates; the template variables and the **precise** syntax of their values would need to be specified.
- A well-known URL structures for each resource type.
- Some sort of opaque top-level map resource with hypermedia controls to various sub-resources and related resources; this would probably be the most RESTful approach, but might be the trickiest to define since there are so many variables (spatial window, coordinate system, layers, styles, extra dimensions, map size in pixels, image format, etc.) to take into consideration, and the mechanism would have to be as expressively powerful as the current KVP interface.
- A combination of one or more of the above.

Chapter 9. WMTS REST Server

The WMTS work item (A042) for Testbed 12 involved three tasks:

1. Deployment of the WMTS server, serving data for GeoPackage generation
2. An implementation of the TileJSON for the WMTS simple profile as per clause 10.3.1 or OGC 15-053
3. Support for the two tile matrix sets World Mercator (EPSG 3395) and Web Mercator (EPSG 3857 or EPSG 4326)

The first work item was completed very early on with the deployment of a WMTS server offering National Land Cover and Ortho Imagery data. The remainder of this clause describes the other two work items.

9.1. Resources to be provided

According to the WMTS Specification (OGC 07-057r7), the resources provided by a WMTS REST server are listed in Table 9.

Table 10. Resources provided by the WMS REST server

Resource Class	Description	Acccess Path
Capabilities	The complete service metadata document.	{WMTSBaseURL}/1.0.0/WMTSCapabilities.xml
Tile	A rectangular pictorial representation of geographic data, often part of a set of such elements, covering a spatially contiguous extent and sharing similar information content and graphical styling, which can be uniquely defined by a pair of indices for the column and row along with an identifier for the tile matrix.	{WMTSBaseURL}/{TileMatrixSet}/{TileMatrix}/{TileRow}/{TileCol}.png
FeatureInfo	Information related to a particular pixel of a map that refers to the geographic data portrayed on that area.	{WMTSBaseURL}/{TileMatrixSet}/{TileMatrix}/{TileRow}/{TileCol}/{J}/{I}.xml

Besides the standard resources provided by a WMTS (i.e. tiles), if we define a TileJSON document of a layer to be a resource in and of itself, then the resource provided by A042 is a "TileJSON document".

The TileJSON document is defined by an open specification that can be found here: https://github.com/mapbox/tilejson-spec/tree/master/2.1.0. The document defines a very simple encoding for describing the availability of a WMTS Simple profile layer in Web Mercator.

9.1.1. Operations on resources

The WMTS is a data portrayal service and as such, the only operation defined is the HTTP GET method which is used to retrieve the resources provided by the service (i.e. tiles and TileJSON documents).

9.1.2. Associations between resources

A WMS Capabilities document provides links to the TileJSON document of each layer in two redundant ways:

1. by advertising a service-oriented GetTileJSON operation which takes SERVICE, VERSION, OPERATION=GetTileJSON and LAYER parameters
2. by advertising a "TileJSON" ResourceURL for each layer

A TileJSON document in turn provides links to a set of tiles by means of tile-URL templates (as defined by the TileJSON specification).

9.2. Implementation

The TileJSON-generating WMTS server was deployed in Testbed 12 at the following URL:

http://tb12.cubewerx.com/a042/cubeserv/default/wmts/1.0.0/WMTSCapabilities.xml

The following is a RESTful example of accessing a TileJSON document from this server using the HTTP GET method:

```
http://tb12.cubewerx.com/a042/cubeserv/default/wmts/1.0.0/tileJSON/National_Land_Cover
.National_Land_Cover
```

The following TileJSON document is retrieved by this URL (NOTE: Normally, packed JSON output is generated from by the A042 server but the output has been wrapped in this example to make it easier to read):

```
{"tilejson":"2.1.0",
    "name":"National Land Cover",
    "tiles":["http://tb12-1.cubewerx.com/a042/OpenImageMap/tilesets/USGS/
              National_Land_Cover/National_Land_Cover/default/
              smerc/{z}/{y}/{x}.jop",
             "http://tb12-2.cubewerx.com/a042/OpenImageMap/tilesets/USGS/
              National_Land_Cover/National_Land_Cover/default/
              smerc/{z}/{y}/{x}.jop",
             "http://tb12-3.cubewerx.com/a042/OpenImageMap/tilesets/USGS/
              National_Land_Cover/National_Land_Cover/default/
              smerc/{z}/{y}/{x}.jop",
             "http://tb12-4.cubewerx.com/a042/OpenImageMap/tilesets/USGS/
              National_Land_Cover/National_Land_Cover/default/
              smerc/{z}/{y}/{x}.jop"],
    "bounds":[-2493045,1317885,-1713555,2497245]
   }
```

9.3. Lessons Learned

This exercise showed that the WMTS specification can be easily extended to support other tile-access strategies.

TileJSON essentially does what WMTS does, but in a more restrictive way, allowing only the Spherical Mercator coordinate system, a single layer, a single style, no extra dimensions, no feature-info querying, etc.. This makes TileJSON clients a bit easier to implement than WMTS clients (assuming that the restrictions of TileJSON are acceptable).

Adding TileJSON support to a WMTS could help allow a TileJSON client to use the WMTS. However, it's not a perfect plug-in solution for TileJSON clients, because the WMTS capabilities document still needs to be parsed (either by a human or automatically) in order to determine the TileJSON URLs.

9.4. Relationship of TileJSON to REST-WMS/WMTS

Now that is has been shown that a WMTS can generate a TileJSON document the question of whether all the necessary components are in place to build a REST-WMS/WMTS can be discussed.

The availability of TileJSON is orthogonal to the issue of REST. Its availability or not does not make a service any more or less RESTful. A TileJSON document is simply an alternative representation of a set of tile URL templates. The primary effects of the availability of TileJSON are:

- TileJSON clients will have an easier time communicating with a WMTS
- being able to access TileJSON from a WMTS will likely aid in the construction of browser-based WMTS clients because JSON is natively handled by Javascript

On the issue of building a REST-WMS/WMTS, the WMTS is (depending on how rigorous your requirements for RESTfulness are) already a RESTful service. Its primary resources (i.e. tiles) are individually addressable via URL and there is a means of navigating from the service root to those

resource via the tile URL templates advertised in the service metadata document (i.e. the capabilities document).

The WMS, on the other hand, cannot currently be considered a RESTful service. In some circumstances where the full flexibility of the WMS API is not required (e.g. building web-based geoportal applications) the WMTS can be considered a RESTful replacement for WMS. However, as already discussed in the "WMS REST Server" clause, the WMS is far more flexible and dynamic than the WMTS and would thus require its own RESTful API to be defined to properly cover all the requirements of a WMS.

Chapter 10. WCS REST Server

The WCS REST Server is used to serve multidimensional coverages (terrain data, satelite imagery, climate data) in a RESTful way that allows for easier client development using standard web libraries and clients (e.g. Postman, RestfulJs, the average browser).

The coverages that were used to exemplify our REST requests are provided by the NASA Earth Observatory and contain monthly-averaged temperature and radiance data with global coverage over a span of 10 years in two different formats. The first, contained in the AvgLandTemp coverage, exposes the direct values as floating points, the other, RadianceColorScaled contains radiance data which was processed using a color profile resulting in an multi-band rgb coverage.

The implementation used in the examples follows the WCS REST Protocol Binding draft extension that is described in the next section. The technical details of the implementation are discussed in the Implementation section.

10.1. WCS REST binding

Details about the WCS REST Protocol Binding can be found in the WCS REST extension draft. Synthesised, the protocol binding exposes all the functionality of the WCS service by mapping previously defined operations like GetCapabilities or DescribedCoverage to REST resources like Capabilities and CoverageDescription. Both the core and extension requests are covered by the protocol binding.

10.1.1. Provided resources

The resources that are provided by a WCS REST server are listed in Table 10 below and include the capabilities document of the server, coverages (including subsets) and corresponding metadata

Table 11. Resources provided by the WCS REST server

Resource Class	Description	Acccess Path	Example
Capabilities	The complete service metadata document. It contains a list of available extensions and all the available coverage resources	{WCSRestBaseUrl}/capabilities	http://ows.rasdaman.org/rasdaman/ows/wcs/2.0.1/capabilities
Coverage	Full coverage in the format negotiated by the client and server through the proper HTTP Headers (e.g. Accept)	{WCSRestBaseUrl}/coverage/{coverageId}	http://ows.rasdaman.org/rasdaman/ows/wcs/2.0.1/coverage/NN3_1
Coverage Description	Full metadata regarding one specific coverage in the negotiated format	{WCSRestBaseUrl}/coverage/{coverageId}/description	http://ows.rasdaman.org/rasdaman/ows/wcs/2.0.1/coverage/AvgLandTemp/description

Resource Class	Description	Acccess Path	Example
Subset of a coverage	A coverage derived on the fly from a subset operation applied to a persistent coverage. *The subset of a coverage is still a coverage	{WCSRestBaseUrl}/coverage/{coverageId}/subset({subset})	http://ows.rasdaman.org/rasdaman/ows/wcs/2.0.1/coverage/AvgLandTemp/subset(Lat(0:1))/subset(Long(0:1))
Range subset of a coverage	A coverage derived on the fly from a range subsetting operation applied to a persistent coverage *The range subset of a coverage is still a coverage	{WCSRestBaseUrl}/coverage/{coverageId}/rangesubset({subset})	http://ows.rasdaman.org/rasdaman/ows/wcs/2.0.1/coverage/RadianceColorScaled/subset(Lat(0:1))/subset(Long(0:1))/rangesubset(Red)

10.1.2. Operations on resources

Due to the protocol binding only GET operations can be applied on the resources mentioned in the section above. This follows the REST guidelines as all operations in WCS are idempotent and are mapped to concrete resources.

Since the first version of the protocol binding draft, the WCS-T extension of WCS was adopted as a standard in OGC, which provides the first non-idempotent operation that a service serving coverages could implement. It is our opinion that based on this new addition to the ecosystem two new operations should be defined in a future draft of the protocol that allow PUT and PATCH operations on a coverage resource. The table below contains this proposal as well.

Table 12. Operations provided by the WCS REST server

HTTP Operation	Acccess Path	Parameters	Description
GET	{wcsRESTBaseURL}	NA	Retrieval of Capabilites resource
PUT	{wcsRESTBaseURL}/coverage/MyCoverage	A valid coverage in one of the server supported formats	Creates a new coverage identified by the given url
PATCH	{wcsRESTBaseURL}/coverage/MyCoverage OR {wcsRESTBaseURL}/coverage/MyCoverage/subset(ansi("2012-01-01"))	A valid coverage in one of the server supported formats	Updates an existing coverage or a subset of it with the given input coverage

10.1.3. Associations between resources

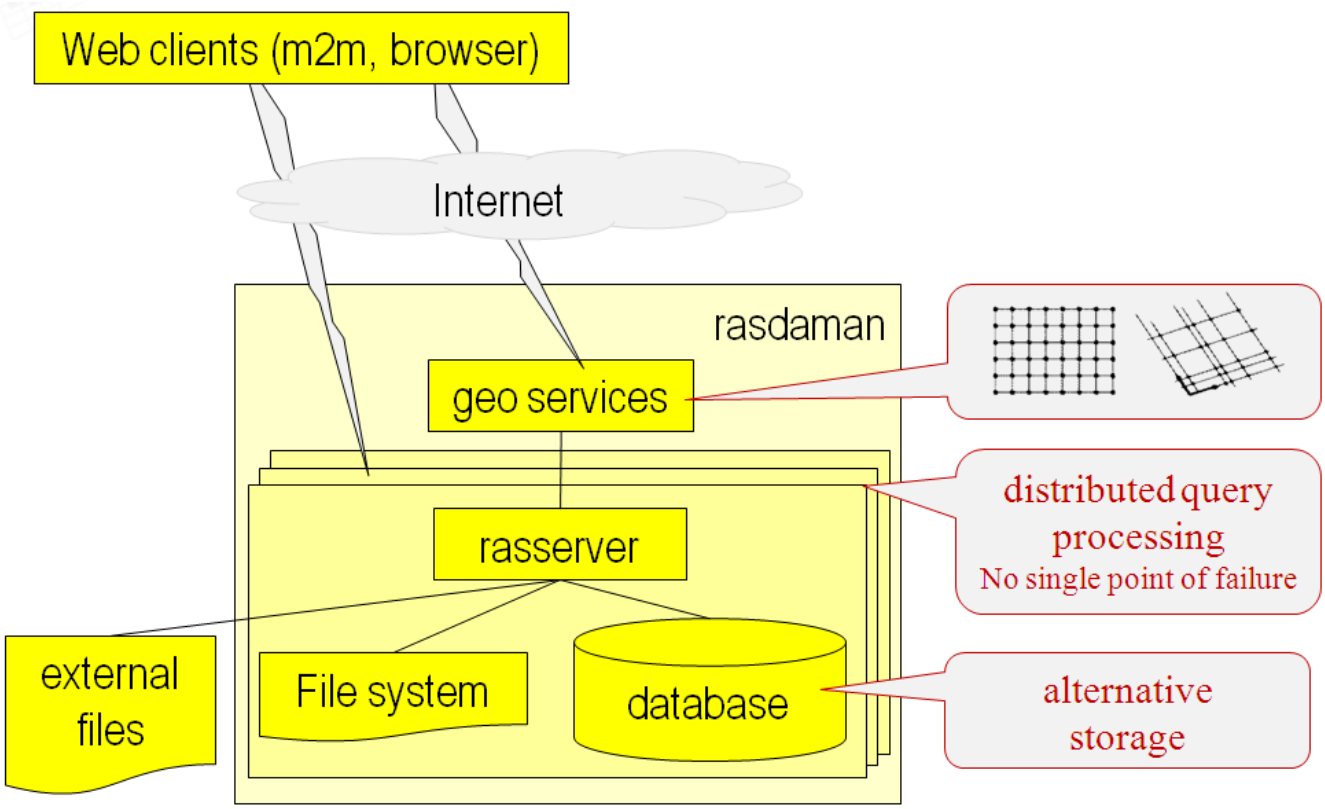
No direct associations are made between the resources as this is a Level 2 implementation according to Richardson's maturity model.

A Level 3 implementation should aim to associate the Subset and RangeSubet of a Coverage with

the original coverage in a meaningful way, most likely by using some standardized URL templates for describing the association. At this time there is no concrete REST solution that would allow a direct link from a coverage to all its subset coverages without exploding the coverage document in terms of size.

10.2. Implementation

The implementation is based on the WCS Server of the rasdaman array database and extends it by adding a new Protocol Binding. The architecture of a typical server is described in the following diagram:



The REST implementation is using the Java Servlet API to parse the URL into meaningful fragements that are grouped together into a WCS Internal request that is independent of any specific protocol. The request is then passed to a service that handles the request and provides a response that can be returned to the user.

10.3. Lessons Learned

The Web Coverage Service is an operation-oriented web service and this made encoding the operations in a RESTful way slightly cumbersome. Most operations, for example subsetting, rangesubsetting and scaling can be mapped to resources, as any operation on a coverage will result in a coverage with different properties. However, the representation of these ad-hoc resources obtained from the original coverage do not add anything in particular to the existing KVP protocol binding which uses the same communication protocol (HTTP), same way of addressing the end result (URLs), same way of building the requests (constructing URLs as opposed to navigation through association links).

Associations between resources are unfortunately impossible to implement as the number of

associations between certain resources is too large (a coverage has a virtually unlimited amount of subset coverage resources). Furthermore, no concrete method of describing an association between resources in the GML format is possible as far as we know.

Since the Capabilities are hard to use for describing the REST binding, the capabilities have been unchanged in the current implementation. However, a common way for describing the REST binding in a machine-readable manner needs to be developed in future versions.

Despite the shortcomings, the REST protocol binding proved simple to implement on top of an existing WCS platform and offers some advantages to users more accustomed to the REST approach in non-OGC services by defining resources that can be accessed through HTTP requests. The implementation done in this deliverable proves that the protocol is viable and can be implemented by service providers.

Chapter 11. WPS REST Server

This section describes the REST Binding and corresponding implementation of a Web Processing Service (WPS) offering conflation processes (A047).

The WPS REST Conflation Service was used to conflate road datasets in an area-of-interest. The reference layer, i.e. the layer that should be used as base, was the USGS Trans_RoadSegment layer. As geometry snap layer, delivering updated/new features, a current snapshot of OpenStreetMap data in the area of interest was chosen. Figure 6 shows the Trans_RoadSegment data with a OpenStreetMap Background:

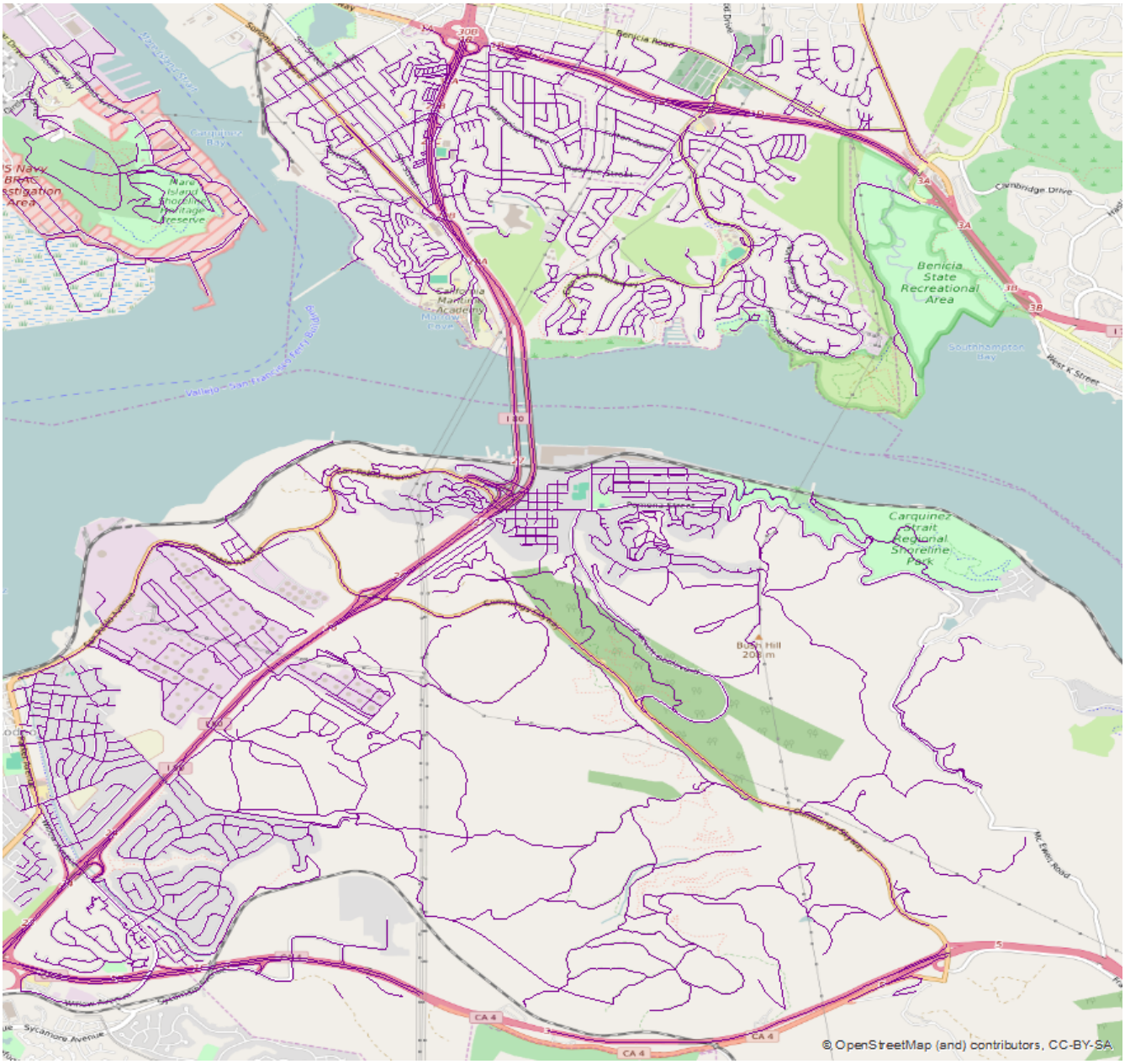


Figure 6. Trans_RoadSegment data with OpenStreetMap background

The OSM roads are shown in Figure 7 below.

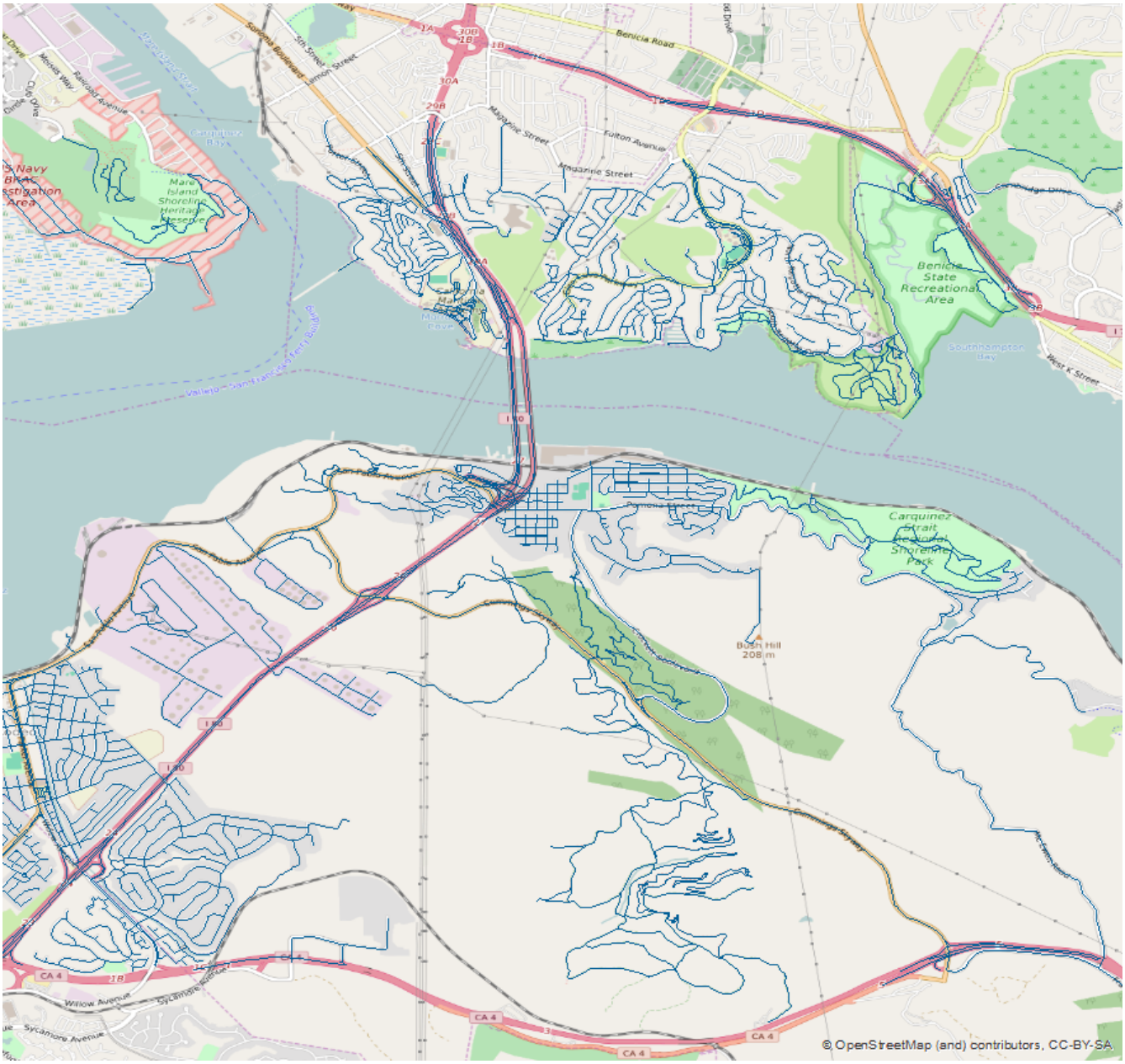


Figure 7. OSM Roads data

You can see that the Trans_RoadSegment layer is already complete to a high degree. The OSM layer is missing some features of the Trans_RoadSegment layer but has more detail in other areas.

The conflation process was performed using Hootenanny with the standard parameters. The output is shown in Figure 8:

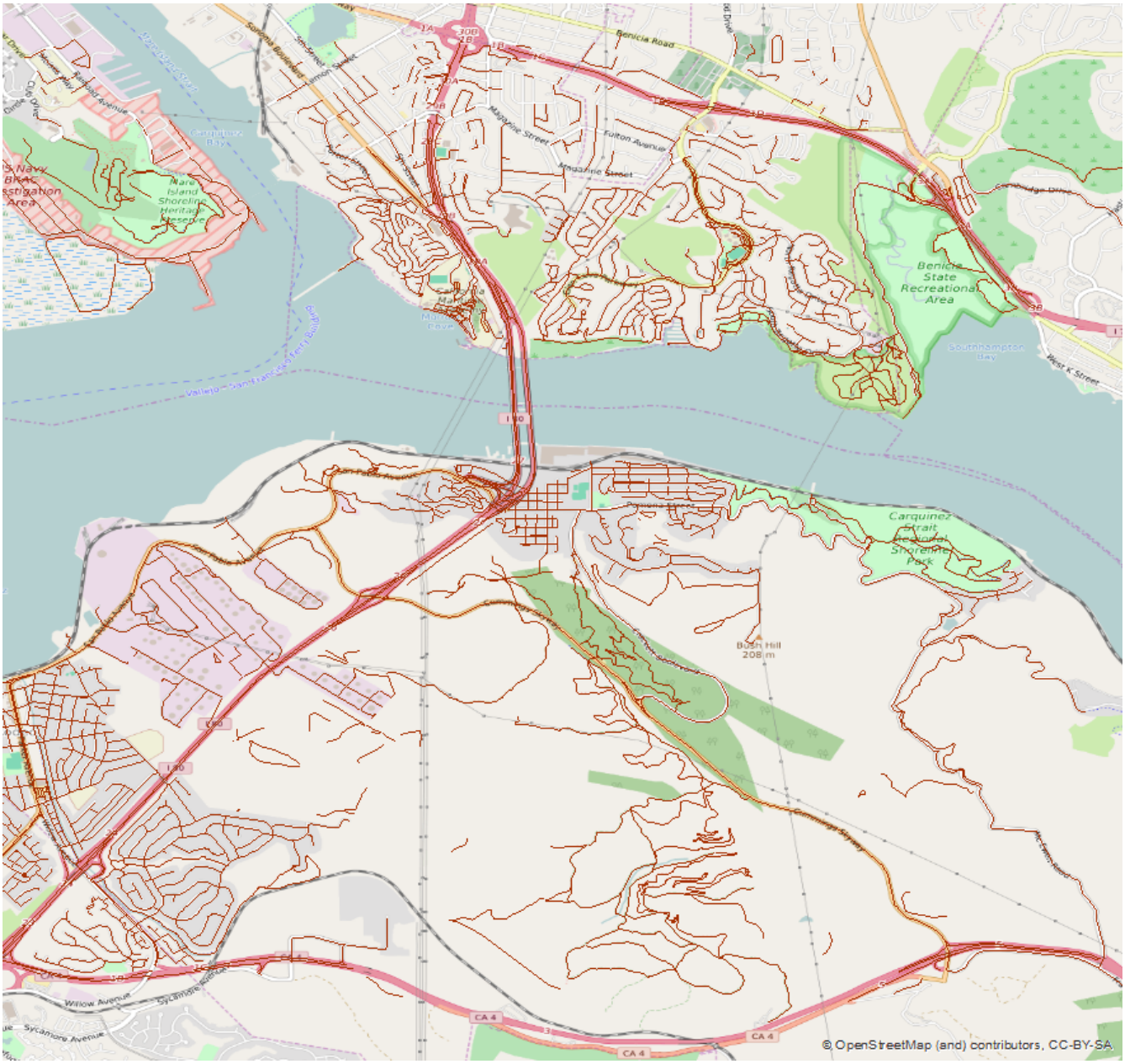


Figure 8. Resulting road data set that was generated by the conflation process

The conflation results will be discussed in detail in The WPS Conflation Service Profile ER (OGC 16-022).

11.1. Resources to be provided by WPS

As there is not yet an official REST binding of the WPS available, the REST binding has been specified in Testbed 12. The resources that are provided by a WPS REST server are listed in Table 12 below and include the capabilities document of the server, the list of processes available (ProcessCollection and Process), jobs (running processes) and outputs of processes.

Table 13. Resources provided by the WPS REST server

Resource Class	Description	Acccess Path	Example
Capabilities	The complete service metadata document.	{WpsRESTBaseURL}	http://geoprocessing.demo.52north.org:8080/wps-proxy

Resource Class	Description	Acccess Path	Example
ProcessCollection	List of processes available	{WpsRESTBaseURL}/processes	http://geoprocessing.demo.52north.org:8080/wps-proxy/processes
Process	Detailed process description of a single process	{WpsRESTBaseURL}/processes/{process-id}	http://www.maps.bob/etopo2/default/WholeWorld_CRS_84/10m/1/3/86/132.xml
JobCollection	List of jobs of a process	{WpsRESTBaseURL}/processes/{processID}/jobs/{job-id}	NA
Job	Representation of a job (execution of a process) containing status information	{WpsRESTBaseURL}/processes/{processID}/jobs/{job-id}	NA
Process Output Data	Resource containing the different process outputs inline or as reference.	{WpsRESTBaseURL}/processes/{processID}/jobs/{job-id}/outputs	NA
Notations: {WpsRESTBaseURL}: The base URL of the WPS REST endpoint. {process-id}: identifier of a process. {job-id}: identifier to a job.			

11.2. Operations on WPS resources

In general, the HTTP GET operation is used to provide access to the resources described above. However, in order to create a new job, the HTTP POST method is used to post a new job by sending a new job resource represented by an execute request to the server. This results in the operations listed in Table 13 below.

Table 14. Operations on resources provided by the WPS REST server

HTTP Operation	Acccess Path	Parameters	Description
GET	{WpsRESTBaseURL}	NA	Retrieval of Capabilities resource
GET	{WpsRESTBaseURL}/processes	NA	Retrieval of ProcessCollection resource
GET	{WpsRESTBaseURL}/processes/{process-id}	NA	Retrieval of a Process resource (process description)
GET	{WpsRESTBaseURL}/processes/{processID}/jobs	NA	Retrieval of list of jobs of a specific process (JobCollection)

HTTP Operation	Acccess Path	Parameters	Description
GET	{WpsRESTBaseURL}/processes/{processID}/jobs/{job-id}	NA	Retrieval of a single Job resource
POST	{WpsRESTBaseURL}/processes/{processID}/jobs	Execute request (contained in body)	Execution of a process
Notations: {WpsRESTBaseURL}: The base URL of the WPS REST endpoint. {process-id}: identifier of a process. {job-id}: identifier to a job.			

11.3. Associations between WPS resources

As stated above in the listing of resources (Table 12), the basic resources managed by the WPS REST server are processes, jobs, outputs and its corresponding collections. An overview on the associations between the resources is given in Figure 9.

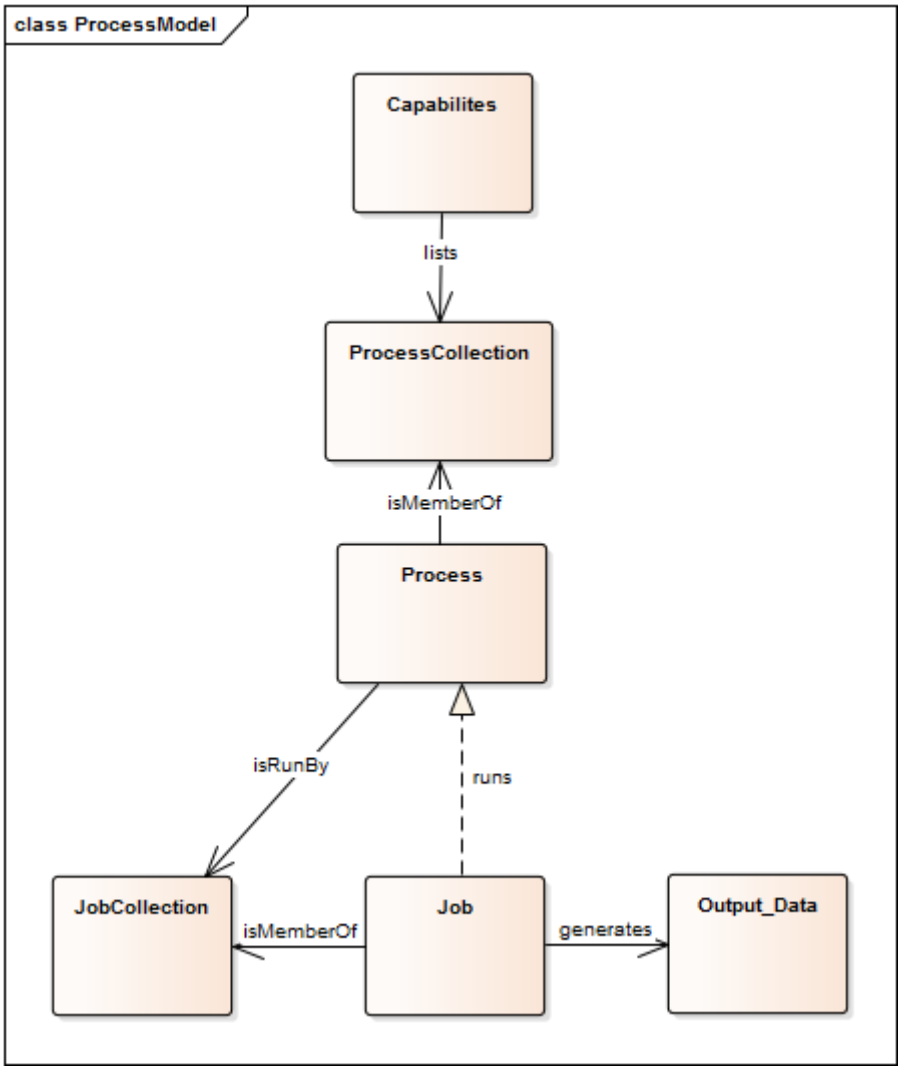


Figure 9. Resource model of the WPS REST server

The Capabilities has an association to the ProcessCollection that aggregates the single Processes

offered by the Web Processing Server. The process has an association to the collection of jobs (JobCollection) that aggregates single jobs, i.e. instances that run a certain process.

11.4. Implementation

The architecture of the 52°North WPS REST PI implementation is shown in Figure 10. The 52°North WPS REST API was realized as proxy to be able to test it with different WPS implementations during the specification process. The proxy was written in Java using the Spring framework and jackson libraries for XML and JSON handling. The WPS requests and responses are handled using the 52°North WPS 2.0 XMLBeans. The underlying WPS was a 52°North WPS 2.0 instance, wrapping the Hootenanny conflation software.

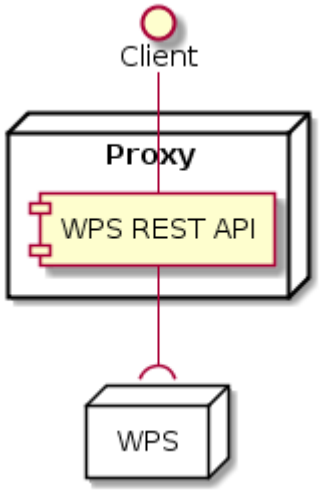


Figure 10. Architecture of the 52°North WPS REST API implementation

In the following, example requests and responses for the different operations and resources will be shown.

Example of HTTP GET request for retrieving JSON WPS Capabilities.

```
http://tb12.dev.52north.org/wps-rest
```

Example of WPS Capabilities encoded as JSON.

```
{
    "Capabilities": {
        "ServiceIdentification": {
            "Title": "52°North WPS 4.0.0-SNAPSHOT",
            "Abstract": "Service based on the 52°North implementation of WPS 1.0.0",
            "ServiceType": "WPS",
            "ServiceTypeVersion": ["1.0.0",
            "2.0.0"],
            "Fees": "NONE",
            "AccessConstraints": "NONE"
        },
        "ServiceProvider": {
            "ProviderName": "52North",
            "ProviderSite": {
                "HRef": "http://www.52north.org/"
            },
            "ServiceContact": {
                "IndividualName": "Your name",
                "ContactInfo": {

                }
            }
        },
        "Contents": {
            "ProcessSummaries": [{
                "identifier": "testbed12.fo.DouglasPeuckerAlgorithm",
                "title": "testbed12.fo.DouglasPeuckerAlgorithm",
                "_processVersion": "1.0.0",
                "_jobControlOptions": "sync-execute",
                "_outputTransmission": "value",
                "url": "http://tb12.dev.52north.org:80/wps-
rest/processes/testbed12.fo.DouglasPeuckerAlgorithm"
            },...
            ]
        },
        "_service": "WPS",
        "_version": "2.0.0"
    }
}
```

The process summaries in the contents-section contain links to the process description of the respective process. The standalone list of processes can be requested as follows:

Example of HTTP GET request for retrieving the list of offered processes encoded as JSON.

```
http://tb12.dev.52north.org/wps-rest/processes
```

Example of WPS Capabilities encoded as JSON.

```
{
  "ProcessSummaries": [
    {
      "identifier": "testbed12.fo.DouglasPeuckerAlgorithm",
      "title": "testbed12.fo.DouglasPeuckerAlgorithm",
      "_processVersion": "1.0.0",
      "_jobControlOptions": "sync-execute",
      "_outputTransmission": "value",
      "url": "http://tb12.dev.52north.org:80/wps-
rest/processes/testbed12.fo.DouglasPeuckerAlgorithm"
    },...
    ]
}
```

Example of HTTP GET request for retrieving the process description of a process encoded in JSON. (NOTE: Request has been line-wrapped for easier reading).

```
http://tb12.dev.52north.org/wps-rest/processes/
testbed12.fo.DouglasPeuckerAlgorithm
```

Example of WPS Capabilities encoded as JSON.(NOTE: Some inputs and formats have been left out for easier reading).

```
{
  "ProcessOffering": {
    "Process": {
      "Title": "Hootenanny Conflation Process",
      "Identifier": "testbed12.lsa.HootenannyConflation",
      "Input": [
        {
          "Title": "INPUT1",
          "Identifier": "INPUT1",
          "ComplexData": {
            "Format": [
              {
                "_default": "true",
                "_mimeType": "application/x-zipped-shp"
              },...
            ]
          },
          "_minOccurs": "1",
          "_maxOccurs": "1"
        },
        {
          "Title": "INPUT1_TRANSLATION",
          "Identifier": "INPUT1_TRANSLATION",
          "ComplexData": {
```

```
            "Format": [
              {
                "_default": "true",
                "_mimeType": "text/x-script.phyton"
              },
              {
                "_default": "false",
                "_mimeType": "text/plain"
              }
            ]
          },
          "_minOccurs": "0",
          "_maxOccurs": "1"
        },
        {
          "Title": "INPUT2",
          "Identifier": "INPUT2",
          "ComplexData": {
            "Format": [
              {
                "_default": "true",
                "_mimeType": "application/x-openstreetmap+xml"
              }...
            ]
          },
          "_minOccurs": "1",
          "_maxOccurs": "1"
        },
        {
          "Title": "CONFLATION_TYPE",
          "Identifier": "CONFLATION_TYPE",
          "LiteralData": {
            "Format": [
              {
                "_default": "true",
                "_mimeType": "text/plain"
              },
              {
                "_default": "false",
                "_mimeType": "text/xml"
              }
            ],
            "LiteralDataDomain": [
              {
                "AnyValue": null,
                "DataType": {
                  "_reference": "xs:string"
                }
              }
            ]
          },
```

```
          "_minOccurs": "0",
          "_maxOccurs": "1"
        },...
      ],
      "Output": [
        {
          "Title": "CONFLATION_OUTPUT",
          "Identifier": "CONFLATION_OUTPUT",
          "ComplexData": {
            "Format": [
              {
                "_default": "true",
                "_mimeType": "application/x-zipped-shp"
              },...
            ]
          }
        },
        {
          "Title": "CONFLATION_REPORT",
          "Identifier": "CONFLATION_REPORT",
          "ComplexData": {
            "Format": [
              {
                "_default": "true",
                "_mimeType": "text/plain"
              }
            ]
          }
        }
      ]
    },
    "_processVersion": "1.0.0",
    "_jobControlOptions": "sync-execute async-execute",
    "_outputTransmission": "value reference",
    "execute-url": "http://tb12.dev.52north.org:80/wps-
rest/processes/testbed12.lsa.HootenannyConflation/jobs"
  }
}
```

Example of HTTP GET request for getting a list of jobs of a process. (NOTE: Request has been line-wrapped for easier reading).

```
http://tb12.dev.52north.org/wps-rest/processes/
testbed12.fo.DouglasPeuckerAlgorithm/jobs
```

Example of a list of jobs for a process encoded as JSON.

```
{
  "Jobs": [
    "1317c058-cb4d-4ab4-ad21-b78e51229a17",
    "1319d2fc-cac8-4e8d-8039-2c511f55a9d3"
  ]
}
```

Example of HTTP POST request for executing a process. (NOTE: Request has been line-wrapped for easier reading).

```
http://tb12.dev.52north.org/wps-rest/processes/
testbed12.fo.DouglasPeuckerAlgorithm/jobs
```

By default, the proces will be executed asynchronously. If the process supports synchronous execution, this can be achieved by appending the following URL-parameter:

Example of HTTP POST request for synchronously executing a process. (NOTE: Request has been line-wrapped for easier reading).

```
http://tb12.dev.52north.org/wps-rest/processes/
testbed12.fo.DouglasPeuckerAlgorithm/jobs?sync-execute=true
```

Example of WPS Execute request encoded as JSON.

```
{
    "Execute": {
        "Identifier": "testbed12.lsa.HootenannyConflation",
        "Input": [
            {
            "Reference": {
                    "_mimeType": "application/x-zipped-shp",
                    "_href":
"http://geoprocessing.demo.52north.org:8080/data/Trans_RoadSegment-aoi.zip"
            },
            "_id": "INPUT1"
            },
            {
            "Reference": {
                    "_mimeType": "text/x-script.phyton",
                    "_href":
"http://geoprocessing.demo.52north.org:8080/data/TNM_Roads.py"
            },
            "_id": "INPUT1_TRANSLATION"
            },
            {
            "Reference": {
                    "_mimeType": "application/x-openstreetmap+xml",
                    "_href":
"http://geoprocessing.demo.52north.org:8080/data/sf_only_roads-aoi.osm"
            },
            "_id": "INPUT2"
            }
        ],
        "output":[{
            "_mimeType": "application/x-zipped-shp",
            "_id": "CONFLATION_OUTPUT",
            "_transmission": "reference"
        },{
            "_mimeType": "text/plain",
            "_id": "CONFLATION_REPORT",
            "_transmission": "reference"
        }],
        "_service": "WPS",
        "_version": "2.0.0"
    }
}
```

The direct response to a asynchronously executed process is HTTP status code 201 (created) and the URL to obtain status information and finally the result. The URL will be returned in a HTTP header named *Location*. For synchronous execution, the result document will be returned after the process has finished.

Example of HTTP GET request for retrieving status information about a asynchonously executed process (NOTE: Request has been line-wrapped for easier reading).

```
http://tb12.dev.52north.org/wps-rest/processes/
testbed12.fo.DouglasPeuckerAlgorithm/jobs/
c731d14b-1de6-499c-9317-20224e056012
```

Example of WPS StatusInfo response encoded as JSON. The process is still running.

```
{
  "StatusInfo": {
    "JobID": "c731d14b-1de6-499c-9317-20224e056012",
    "Status": "Running",
    "Progress": 0
  }
}
```

After the process has finished, the progress element will be replaced by the URL to obtain the outputs

Example of WPS StatusInfo response encoded as JSON. The process has finished.

```
{
  "StatusInfo": {
    "JobID": "c731d14b-1de6-499c-9317-20224e056012",
    "Status": "Succeeded",
    "Output": "http://tb12.dev.52north.org/wps-
rest/processes/testbed12.lsa.HootenannyConflation/jobs/c731d14b-1de6-499c-9317-
20224e056012/outputs"
  }
}
```

Example of WPS Result response encoded as JSON.

```
{
  "Result": {
    "JobID": "c731d14b-1de6-499c-9317-20224e056012",
    "Output": [
      {
        "ID": "CONFLATION_OUTPUT",
        "Reference": {
          "_mimeType": "application/x-zipped-shp",
          "_href":
"http://tb12.dev.52north.org:80/wps/RetrieveResultServlet?id=c731d14b-1de6-499c-9317-
20224e056012CONFLATION_OUTPUT.b1172b1c-c9aa-495a-aa8b-62220ac93605"
        }
      },
      {
        "ID": "CONFLATION_REPORT",
        "Reference": {
          "_mimeType": "text/plain",
          "_href":
"http://tb12.dev.52north.org:80/wps/RetrieveResultServlet?id=c731d14b-1de6-499c-9317-
20224e056012****CONFLATION_REPORT.220391a6-4357-44e2-b5f3-c0a0983cedae"
        }
      }
    ]
  }
}
```

11.5. Lessons Learned

The WPS REST API simplifies the interaction between servers and clients. The HTTP methods with its clear semantics ease to understand the interface and its communication pattern. The usage of hypermedia encoded as JSON allows clients to browse from the Capabilities to the processes available, to navigate from these processes to jobs (running instances of these processes) and, once a job is finished, to browse to the resulting outputs of a job. We hence submitted a change request (Request 396: Specify a REST binding for WPS 2.0) to the WPS SWG for defining a WPS REST binding based on the REST component developed in this testbed.

One difficulty encountered is the description of the API using the Capabilities document. In general, in order to interact with an HTTP based RESTful API, clients need to know

- which resources are offered and
- which HTTP methods need to be applied in order to retrieve or manipulate these resources

While the information about the resources offered can be easily embedded in the Contents section of the Capabilities document, the integration of the description about which HTTP methods are applicable on which resources is not straightforward. Usually, the OperationsMetadata section would be used for describing the operations offered, the endpoints and the operation parameter. However, since the OperationsMetadata have been defined for Web Services in Service-oriented

Architectures, the description of REST APIs is not straightforward for the following reasons:

- Only HTTP GET and POST are supported.
- There is no way to describe on which resources the HTTP methods, e.g. HTTP GET, can be applied.

For this reason, we omitted the OperationsMetadata section and described the application of the HTTP methods in the specification draft. In future, it would be desirable to also offer a machine-readable description of the API relying upon existing description language, e.g. the OpenAPI specification [4].

Chapter 12. OGC REST Components Outside Testbed 12

This clause describes other REST components that have been not directly addressed in Testbed 12. NOTE: This clause still needs to be completed. It will provide short overviews on the different components and references to existing documentation, e.g. previous ERs or specifications.

12.1. WaterML REST API

The WaterML REST API has been published as a best practice document (OGC 15-033) resulting from the interoperability experiment testing the information model part 2 of WaterML (OGC 15-018r2). The REST API hence provides access to the following resources:

Table 15. Resources provided by the WPS REST server

Resource Class	Description	Acccess Path	Example
ResourceListing	Listing of resources.	{BaseURL}	http://waterml2.csiro.au/rgs-api/v1/
Gauging		{BaseURL}/gauging	http://waterml2.csiro.au/rgs-api/v1/gauging/
Conversion	Relationship between two phenomena at a moment in time as defined by an equation, table of paired values or other form.	{BaseURL}/conversion	http://waterml2.csiro.au/rgs-api/v1/conversion/
MonitoringPoint	primary location for conducting observations	{WpsRESTBaseURL}/monitoring-point	http://waterml2.csiro.au/rgs-api/v1/monitoring-point/
Notations: {BaseURL}: The base URL of the REST endpoint.			

The RESTful API defined is currently a read-only service and thus only implements the HTTP GET method for resources.

An implementation of the REST API is available at

```
http://waterml2.csiro.au/rgs-api/v1/
```

Since the resources returned contain links to other resources the RMM level is level 3.

12.2. Sensor Things API

The OGC SensorThings API Part 1: Sensing (15-078r6) is intended as a standard for connecting Internet of Things (IoT) devices as well as related data and applications via the Web. Currently, Part 1 of this standard is available which covers sensor observations. It supports the retrieval of observations of IoT devices and their related metdata. Tasking functionality will be covered by a separate part of this standard.

The following table provides an overview of the ressources offered by the Sensor Things API:

Table 16. Resources provided by the Sensor Things API

Resource Class	Description	Acccess Path	Example
ResourceListing	Listing of resources.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/
Thing	Individual physical or information world object.	{BaseURL}/v1.0/Things/	http://example.sensorup.com/v1.0/Things/
Location	Entity describing the last known loacation of a thing.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/Locations/
HistoricalLocation	Entity describing a past location of a thing (at a certain time).	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/HistoricalLocations/
Datastream	Collection of Observations of the same ObservedProperty by the same Sensor.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/Datastreams/
Sensor	Instrument observing an ObservedProperty to generate Observations.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/Sensors/
ObservedProperty	Phenomenon of an Observation.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/ObservedProperties/
Observation	Act of determining the value of an ObservedProperty at a certain time.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/Observations/
FeatureOfInterest	Geospatial object to which an observation is associated.	{BaseURL}/v1.0/	http://example.sensorup.com/v1.0/FeaturesOfInterest/
Notations: {BaseURL}: The base URL of the REST endpoint.			

It is important to note, that the Sensor Things API is, like the Sensor Observation Service (SOS),

based on the general ISO/OGC Observations and Measurements model. As the functionality of the sensing part of the Sensor Things API is widely identical to the core operations of the SOS standard, a stronger harmonisation would be recommended. In order to avoid confusion among potential operators and users, an idea could be to advance the Sensor Things API to become a REST binding of the SOS standard.

Chapter 13. Commonalities and Differences between the different REST Servers

The previous clauses show that the different REST server implementations for WMS, WMTS, WCS and WFS, differ regarding the compliance to the REST principles. This results in different levels of the Richardson Maturity Model, as shown in Table 16.

13.1. RMM Levels (Support of Hypermedia)

As can be seen in Table 16, hypermedia is not always returned due to the very different nature of geospatial information that is provided. While, for example, features served in GeoJSON may also easily include links to related features, it is more difficult to embed links in an image that is returned from a WMS server. Also, as stated in the lessons learned section of the WCS REST server description (Clause 10), in some cases there may be an infinite set of possible associations, e.g. a coverage resources has an unlimited number of associations to subset coverage resources. Since the subsetting is depending on the client and application, it does not make sense to provide only a subset of those associations.

Table 17. Maturity Levels of the different OGC Testbed 12 REST servers

Component	RMM Level	Comment
REST WCS (A044)	2	No hypermedia returned.
REST WMS (A040)	2	No hypermedia returned.
REST WMTS (A042)	3	If TileJSON is returned, hypermedia is returned, otherwise no hypermedia for the tile resources.
REST WFS (A035)	2	In TB 12 implementation, no hypermedia returned. In TB 11 example with comprehensive usage of hypermedia (see OGC 15-052r1, section 6.4).
REST WPS (A044)	3	Hypermedia supported.

13.2. Complexity and Implementation Efforts

In the lessons learned, all developers of the REST servers in Testbed 12 mentioned the simplicity of the implementations. This may be considered as a major improvement since especially the support of complex XML schemas with circular references was considered to be a major implementation task for the POX or SOAP based OGC services. Instead, requests are simply URLs that are invoked with different HTTP methods.

13.3. Service Capabilities

Since all OGC services inherit from the OWS Common specification (OGC 06-121r9), they need to provide a Capabilites document stating:

- the service type, e.g. WMS, WCS or WPS (section ServiceIdentification)
- information about the provider, e.g. contact, institution, etc. (section ServiceProvider)
- the supported service operations and parameters, e.g. GetMap with mandatory parameters (section OperationsMetadata)
- information about the content provided, e.g. layers in WMS, feature types in WFS
- options on filtering content, e.g. which spatial filters are supported (optional; not specified in OWS Common, but in the different W*S service specifications)

Hence, the Capabilities document is considered as an additional resource offered by the different OGC REST servers. However, the way how the Capabilities document is offered differs in the different implementations. WPS, WFS, and WCS are providing the Capabilities document at the base URL of the REST server, whereas the WMTS is providing it under a fixed subdomain (/capabilities), as specified in the WMTS specification (OGC 07-057r7).

In order to utilize a RESTful API, clients need to know

- which resources are offered
- how too access and/or manipulate them, i.e. which HTTP methods can be applied to the resources

While the first information can be embedded in the Contents section of a Capabilities document (as shown, for example for the WPS REST server), the way how to access these resource or manipulate them cannot be simply mappped to the operations metadata section of the Capabilities. Hence, in case of WPS, for example, the decision was made to not provide the OperationsMetadata section (see also Clause 14).

13.4. JSON Encodings

Except the WCS REST server, all of the services are providing JSON encodings for at least some resources. Similarly to the support of hypermedia, the provision of resource representations as JSON is depending on the nature of the resources offer. While the encoding of geospatial features with vector geometries can be easily done using GeoJSON, a JSON encoding of Coverages or Map images provided by a WMS is not meaningful.

A special role is the encoding of the Capabilities document. So far, a dedicated mapping from the Capabilities elements to JSON does not exist. Rules how a mapping may be realized are currently defined in the JSON as an activity in this testbed. The WPS REST server provides an example how a JSON-encoded Capabilities document may look like (Clause 11).

13.5. Associations

There are three different types of associations in OGC REST APIs:

- within resource offered by a single REST API:
  - Example: link from feature collection to single feature in RESTful WFS
- across different REST APIs:

    - Examples: link from catalog entry to a feature type in a RESTful WFS; link from an observation in a SOS to a featureOfInterest (observed feature) in a WFS
- to external resources
    - Example: features in a WFS may be linked to additional external information, e.g. data for facility management maintained in a separate service

Furthermore a specific subtype may be links from a whole resource to subresources that are part of a larger resources. The association between coverages in coverage subsets may be consider as such an association.

The associations given in the resource representations are currently either given as embedded URLs in JSON (see Clause 11 for examples) or as ATOM links in RESTful WFS (see OGC 15-052r1). REST APIs supporting RMM Level 2 (WMS and WCS) are not providing the associations explicitly as links within data, but rahter explicitly through their hierarchical URL templates.

A discussion has been held how to provide links for tile resources encoded as binary media types in a WMTS. A first suggestion was to define additional HTTP header paramters. Since there is already a Link parameter, it was agreed that this may be the right way to go, though this was not demonstrated by an implementation in Testbed 12.

13.6. URL Templates vs. HATEOAS

The WMTS and WCS REST bindings are defining URL templates in order to construct the URLs for specific tiles or coverage subsets by replacing certain parts of an URL. For example, the URL template for retrieving a specific tile in WMTS looks like:

```
./{TileMatrixSet}/{TileMatrix}/{TileRow}/{TileCol}.png
```

The parameter names between the curly brackets ('{' and '}') need to be replaced by a client in order to retrieve a specific tile, resulting in an URL like:

```
https://my.wmts.org/rest/tileMatrixSetXY/TileMatrixYZ/41/42.png
```

Since these APIs are not returning hypermedia (RMM Level 2), the URL templates are needed to specify the access to resources. In contrast, if, for example, the REST API is providing hypermedia, the Capabilities may serve as an entry point to the resources and the client can then navigate through the different resources and states following the HATEOAS principle.

The usage of URL templates is disputed controversely in the Web. The advantage of the approach is that there is a clear pre-defined structure for the resource identifiers and there is no need for hypermedia. Hence, the usage of URL templates appears to be appropriate for APIs at RMM level 2.

However, the approach also comes with disadvantages: The URL structure cannot be easily changed by the server, if, for example, also the resource model has changed, since the client is expecting the structure defined by the template. In addition, the approach contradicts the HATEOAS principle, since the URLs are not opaque and the links to resources are not given in hypermedia, but the a-

priori specified URL template. Finally, there are also security concerns, since the URLs may be easily hackable to resources that are not officially published.

In the end, similarly to the support of JSON and hypermedia, the usage of URL templates depends on the nature of the data that is provided via the APIs: In case of coverages (WCS), maps (WMS), and map tiles (WMTS), the usage of URL templates is reasonable. In case the resource representations can be provided as hypermedia, there is no need for a pre-defined URL structure, if there is a pre-defined entry point to the resources, e.g. the Contents section of the Capabilities.

13.7. Request Parameters & Filtering

Several options are available to filter on resources in a RESTful API and to implement additional query parameters. The traditional way is to utilize the Key-Value-Pair approach, i.e. appending parameters to the URL. For example, when searching for features within a Bounding Box, the bbox parameter may be appended to the resource URL, applying the filter on the resource provided.

```
https://my.wfs.org/rest/featureCollection?bbox=7.0,52.0,8.0,53.0
```

Another option would be to encode the bbox parameters in the URL and hence consider the result set laying in the bounding box as an additional resource offered by the REST server. This may look like:

```
https://my.wfs.org/rest/featureCollection/7.0/52.0/8.0/53.0
```

However, this appears to be somewhat cumbersome since no template for encoding the bounding box parameters (minx, miny, maxx, maxy) currently exists. Furthermore, another open question would be, which resource is offered at a shorter path, e.g.

```
https://my.wfs.org/rest/featureCollection/7.0
```

Similarly, several approaches exist to query the resource representations in a specific format, i.e. content negotiation. The common way to implement content negotiation is using the Accept parameter in the HTTP header. However, some other approaches exists in the current OGC REST APIs: The WFS REST server implemented in Testbed 12 utilizes the outputFormat parameter of the URL:

```
http://ows12.azurewebsites.net/wfs/featuretypes/AdministrativeSubdivision?outputFormat
=json
```

The WMTS specification suggests to append the file format extension to the URL, which may look like the following URL containing the ".xml" suffix at the end indicating that the XML media type is requested:

```
http://my.wmts.org/rest/Capabilites.xml
```

To sum up, no common approach for applying filters on the resources and additional parameters, e.g. for content negotiation currently exist.

13.8. Security

Since there were no special requirements for security for the REST components, the only security method implemented has been HTTP Basic Authorization.

Chapter 14. Recommendations

As a result from the discussions held about RESTful APIs for OGC services during the Testbed 12, a couple of recommendations have been derived that are listed below.

14.1. Suggested RMM Level

The current REST APIs implementing WMS, WMTS, WCS, WFS, and WPS vary in the RMM level that is supported. As described in the implementation sections (Clauses 7 to 11) and the previous discussion about the commonalities and differences (Clause 13), depending on the nature of the information that is offered, e.g. coverages vs. features, hypermedia may be returned or not.

Recommendation 1: If possible, OGC REST APIs should support RMM level 3.

14.2. Identification of Resources

The resources offered by a specific service type are defined in the service specification or data models referenced from the service specification. For example, as illustrated in Clause 6, the resources offered by a Sensor Observation Services (SOS) are defined in the Observations & Measurements (O&M) and Sensor Model Language (SensorML) standards. Similarly, the Web Feature Service offers features compliant to the General Feature Model.

Recommendation 2: The resources offered by an OGC W*S RESTful API should be compliant to the data specifications of the corresponding service specifications.

Furthermore, the resources should be idenifiable by a unique identifer.

Recommendation 3: All OGC REST implementations shall follow the general REST requirement that resources are identifiable by an unique URI.

In addition, there should be an unique entry point for the resources. This recommendation is given below in the section about the API description and Capabilities document.

14.3. Associations between Resources

The associations given in the resource representations are currently either given as embedded URLs in JSON (see Clause 11 for examples), as ATOM links in RESTful WFS (see OGC 15-052r1) or using XML links in O&M XML. REST APIs supporting RMM Level 2 (WMS and WCS) are not providing the associations explicitly as links within data, but rather explicitly through their hierarchical URL templates.

Resulting from the varying encodings of associations, we define the following recommendations for REST APIs supporting RMM Level 3:

For each media type, a common way how to encode associations should be defined.

Recommendation 4: A unique way how to encode associations should be defined for different media types of OGC REST APIs.

In case the resource representation cannot be used to provide the associations inline, e.g. binary encoding of an image, associations to related resources should be given in the HTTP header using the Link parameter.

Recommendation 5: If links cannot be encoded in the resource representation returned, the Link param of the HTTP header should be used.

Currently, the semantics of associations are defined in the specification documents. One strength of hypermedia and linked data is to utilize explicitly defined semantics of the associations, e.g. in form of dictionaries or ontologies, that can be embedded in different linked data serializations, e.g. RDF or JSON-LD.

A couple of common vocabularies are already existing. One of the most prominent examples is the IANA Link Relations vocabulary [2].

Recommendation 6: OGC REST APIs should re-use existing vocabularies for describing link relations.

However, some of the special spatial relations like topological relations (e.g. within, intersects, etc.) are not covered by these vocabularies. Hence, these need to be specified and maintained by the OGC Naming Authority.

Recommendation 7: Associations representing spatial relationships should be specified by OGC and maintained by the OGC Naming Authority.

14.4. Description of API & Discovery of Resources

As shown in the previous Clause 13, there is currently no single way how the OGC REST APIs are described. Some are using plain XML documents without links to resources, others are embedding links to resources in the XML Capabilites, and others are providing a JSON encoding of the Capabilities with embedded links to resources. There is hence a need for harmonization of the different approaches.

One approache is to utilize the Capabilities document as an entry point to the resources. The WPS REST server developed in Testbed 12 (Clause 11) provides an example how this may look like. In addition the retrieval of the Capabilities document differs between the different REST APIs.

Recommendation 8: The Capabilities document should be provided at the root of the API endpoint.

Recommendation 9: The Capabilities document should provide an entry point to the resources offered by the API.

As there currently is not a concrete JSON encoding of the Capabilities document specified, the encodings vary as well. Hence, there is a need for a specification of a JSON encoding of the Capabilities document.

Recommendation 10: A JSON encoding of the Capabilities document should be specified.

If REST servers and clients would follow completely the HATEOAS principle, no further information would be needed, since requesting the allowed HTTP methods on a particular resource via the

HTTP OPTIONS method would be sufficient. However, as in reality most of the REST APIs are not providing a full HATEOAS support, this information is needed in an API description. As stated above the OperationsMetadata is difficult to apply for the REST API description, in particular which HTTP methods need to be applied in order to retrieve or manipulate the resources offered by the OGC REST API. We hence suggest to also specify the provision of an additional API description file that is based on common Web standards already in use. One possible approach would be to utilize the OpenAPI specification and provide best practices how OpenAPI descriptions can be provided for OGC REST APIs.

Recommendation 11: The OGC should specify how an additional API description can be provided that defines how HTTP methods can be applied to the resources offered in order to retrieve or manipulate them.

14.5. Usage of HTTP Verbs

As already stated in Clause 13, HTTP GET is used for retrieving resource representations. To create new resources, PUT or POST are usually used and DELETE for deleting resources. The REST binding for the WCS suggest to use the HTTP PATCH method for an partial update of a resource. This results in the following usage of HTTP methods:

Recommendation 12: OGC REST APIs should utilize the HTTP methods as shown in Table 17.

Table 18. Recommended HTTP verbs for OGC RESTful APIs

Uniform Resource Locator (URL)	**GET**	**PUT**	**PATCH**	**POST**	**DELETE**
Collection, such as http://api.example.com/resources/	Retrieve representation of collection	Replace entire collection	Update parts of collection	Create new entry in collection	Delete entire collection
Single resource representation, such as http://api.example.com/resources/resource_xy	Retrieve representation of resource	Create or replace representation	Replace parts of representation	-	Delete resource

14.6. Usage of HTTP Status Codes

As common in REST APIs and illustrate already in the previous clauses, HTTP Status Codes should be used (e.g. 200 for OK, 400 for bad request, etc.).

Recommendation 13: OGC REST APIs should utilize the HTTP Status Codes in their responses.

14.7. Filter Parameters and Content Negotiation

A common use case is to apply spatial filters on collections of resources, e.g. the bounding box filter. As it is already common practice to utilize URL parameters for this purpose, it is also recommended to utilize them for applying filters to resource collection, for example:

```
https://my.wfs.org/rest/featureCollection?bbox=7.0,52.0,8.0,53.0
```

Recommendation 14: Filters that should be applied to the resources should be given in the URL as URL parameters.

In the implementation of OGC REST APIs, different options mechanism for content negotiation are implemented. As using the ACCEPT parameter of the HTTP header is already commmon practice, we recommend to specify this option as mandatory.

Recommendation 15: Content negotiation should be implemented by OGC REST APIs using the ACCEPT parameter of HTTP headers.

Appendix A: List of OGC documents dealing with REST

This listing shows the relevant OGC documents that specify REST inerfaces. It extends the previous listing presented in OGC15-052r1.

Date	Title	OGC#	Status
2009-08	OWS-6 DSS Engineering Report - SOAP/XML and REST in WMTS	09-006	Approved Engineering Report
2010-04	OpenGIS Web Map Tile Service Implementation Standard	07-057r/	Implementation Standard
2014-04	OGC RESTful Encoding of Ordering Services Framework For Earth Observation Products	13-042	Best Practices Document
2014-07	RESTful encoding of OGC Sensor Planning Service for Earth Observation satellite Tasking	14-012r1	Best Practices Document
2015-08	OGC® Testbed 11 Implementing JSON/GeoJSON in an OGC Standard Engineering Report	15-053r1	Approved Engineering Report
2015-09	OGC® Testbed 11 REST Interface Engineering Report	15-052r1	Approved Engineering Report
2015-12	OGC Observations and Measurements - JSON implementation	15-100r1	Discussion Paper
2016-01	OGC WMTS Simple Profile	13-082r2	Profile

Appendix B: Revision History

Table 19. Revision History

Date	Release	Editor(s)	Primary clauses modified	Descriptions
April, 2016	.1	S. Jirka, C. Stasch	all	initial version
September, 2016	.2	P. Vretanos, C. Stasch	8 + 9	description of WMS and WMTS
September, 2016	.3	B. Pross	11	description of WPS
September, 2016	.4	A. Dumitru	10	description of WCS
September, 2016	.5	J. Harrison	7	description of WFS
September, 2016	.6	C. Stasch, S. Jirka	6, 12, 13	content added
October, 2016	.7	P. Vretanos	8,9	update according to review feedbackC.
October, 2016	.8	A. Dumitru	10	update according to review feedback
October, 2016	.9	J. Harrison	7	update according to review feedback
October, 2016	1.0	C. Stasch, S. Jirka	all	final revision, section 6.5.3 added

Appendix C: Bibliography

Example Bibliography (Delete this note).

The TC has approved Springer LNCS as the official document citation type.

Springer LNCS is widely used in technical and computer science journals and other publications

NOTE

- For citations in the text please use square brackets and consecutive numbers: [1], [2], [3]

– Actual References:

[n] Journal: Author Surname, A.: Title. Publication Title. Volume number, Issue number, Pages Used (Year Published)

[1] Fielding, R.: Fielding Dissertation: Chapter 5: Representational State Transfer, http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (2000)

[2] IANA. Link Relations, http://www.iana.org/assignments/link-relations/link-relations.xhtml (2016).

[3] Richardson, L., Amundsen, M.: RESTful Web APIs, O'Reilly Media, ISBN 978-1-449-35806-8.

[4] OAI/Open API-Specification: The OpenAPI Specification Repository, https://github.com/OAI/OpenAPI-Specification (2016).

[5] Wikipedia Foundation: Representational State Transfer, https://en.wikipedia.org/wiki/Representational_state_transfer (2016)