OGC JSON Best Practice

Table of Contents

Open Geospatial Consortium

Submission Date: <yyyy-mm-dd>

Approval Date: <yyyy-mm-dd>

Publication Date: <yyyy-mm-dd>

External identifier of this OGC® document: http://www.opengis.net/doc/{doc-type}/{standard}/{m.n}

Internal reference number of this OGC® document: YY-nnnrx

Version: n.n

Category: OGC® Best Practice

Editor: Joan Masó

OGC JSON Best Practice

Copyright notice

Warning

This document defines an OGC Best Practice on a particular technology or approach related to an OGC standard. This document is not an OGC Standard and may not be referred to as an OGC Standard. It is subject to change without notice. However, this document is an official position of the OGC membership on this particular technology topic.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type:	OGC® Best Practice
Document subtype:	
Document stage:	Draft
Document language:	English

License Agreement

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the 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, 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.

i. Abstract

This OGC® document provides a set of best practices on how to use the JavaScript Object Notation (JSON) as an encoding for OGC standards. The document provides a set of rules on how to transform a Unified Modeling Language (UML) model describing the encoding of a format or a service description into a JSON and JSON for Linked Data (JSON-LD) encoding and their corresponding JSON schemas. The document is applicable to OGC data, metadata encodings and OGC services willing to provide a JSON encoding.

ii. Keywords

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, best practice, JSON, JSON-LD

iii. Preface

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.

iv. Submitting organizations

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

Organization name(s)

- UAB-CREAF

v. Submitters

All questions regarding this submission should be directed to the editor or the submitters:

Table 1. Contacts

Name	Affiliation
Joan Masó	UAB-CREAF
Gobe Hobona	OGC

Chapter 1. Scope

This OGC® document provides a set of best practices on how to use JSON as an encoding for OGC standards.

In addition, this OGC® document provides a set of rules on how to transform a UML model describing the encoding of a format or a service description into a JSON and JSON-LD encoding and their corresponding JSON schemas. The rules have been applied and validated in a service metadata document based on OWS Common.

This OGC® document is applicable to OGC data, metadata encodings and OGC services willing to provide a JSON encoding for fast adoption in web browsers.

This OGC® document is complemented by OGC 16-122 (Geo)JSON User Guide that includes guidelines and recommendations for the use of JSON and JSON-LD in OGC data encodings and in OGC services.

Chapter 2. Conformance

This Best Practice defines XXXX.

Requirements for N target types are considered: * AAAA * BBBB

Conformance with this Best Practice shall be checked using all the relevant tests specified in Annex A (normative) of this document.

In order to conform to this OGC® Best Practice, a software implementation shall choose to implement: * Any one of the conformance levels specified in Annex A (normative). * Any one of the Distributed Computing Platform profiles specified in Annexes TBD through TBD (normative).

All requirements-classes and conformance-classes described in this document are owned by the document(s) identified.

Chapter 3. References

The following normative documents contain provisions that, through reference in this text, constitute provisions of 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.

- ISO: ISO 19118:2010 Geographic information — Encoding.
- IETF: RFC7159, The JavaScript Object Notation (JSON) Data Interchange Format. https://www.ietf.org/rfc/rfc7159.txt
- IETF: RFC4627, The application/json Media Type for JavaScript Object Notation (JSON). https://www.ietf.org/rfc/rfc4627.txt
- W3C: JSON-LD 1.0, A JSON-based Serialization for Linked Data. http://www.w3.org/TR/json-ld/
- IETF: draft draft-zyp-json-schema-04, A JSON Media Type for Describing the Structure and Meaning of JSON. https://tools.ietf.org/html/draft-zyp-json-schema-04

NOTE: This ER demonstrates the usefulness of JSON Schema. Unfortunately, there has been no progress in the draft schema towards standardizing it since 2013 All work in this document about JSON schema is based on the last IETF draft (v4) expired "August 4, 2013". At the time to finalize this document we realized that a new version has been recently released (https://tools.ietf.org/html/draft-wright-json-schema-00) that is not considered a normative reference on this document but will be included in the bibliography.

- OGC: OGC 06-103r4, OpenGIS implementation Standard for Geographic Information — Simple feature access — Part 1: Common Architecture.
- OGC: OGC 06-121r9, OGC® Web Services Common Standard

Chapter 4. Terms and Definitions

NOTE: This OWS Common Standard contains a list of normative references that are also applicable to this document.

This document uses the terms defined in Sub-clause 5.3 of [OGC 06-121r8], which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this Best Practice.

For the purposes of this document, the following additional terms and definitions apply.

4.1. Linked Data

a set of best practices for publishing structured data on the Web (source: https://www.w3.org/wiki/LinkedData)

4.2. Abbreviated terms

CIS	Coverage Implementation Schema
GML	Geography Markup Language
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON-LD	JSON for Linked Data
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	RDF Schema
SPARQL	SPARQL Protocol and RDF Query Language
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Chapter 5. Conventions

This sections provides details and examples for any conventions used in the document. Examples of conventions are symbols, abbreviations, use of XML schema, or special notes regarding how to read the document.

5.1. Identifiers

The normative provisions in this document are denoted by the URI

http://www.opengis.net/spec/{standard}/{m.n}

All requirements and conformance tests that appear in this document are denoted by partial URIs which are relative to this base.

Chapter 6. Overview

6.1. Introduction to JSON

This section presents the JavaScript Object Notation (JSON). Afterwards, it discusses important questions such as what JSON offers, when a JSON encoding offers an advantage (e.g. in simplicity and flexibility) in comparison to an Extensible Markup Language (XML) encoding, and when XML encodings can provide a better solution (e.g. adding more expressivity and robustness). Finally it presents some suggestions of what OGC can add on top of the basic JSON definition to ensure better geospatial interoperability of applications using JSON.

6.2. The JSON format

JSON is a very simple data model that can represent four primitives (strings, numbers, booleans (*true* or *false*) and *null*) and includes two structured types (objects and arrays). Strings are enclosed in *quotation marks*, numbers do not have quotation marks, objects are enclosed in curly brackets "{}" and arrays are enclosed in square brackets "[]". An object is an unordered collection of zero or more parameters (name and value pairs, separated by a colon), where a name is a string and a value is a primitive or a structured type. Parameters are separated by commas and usually each of them is written on a different line. An array is an ordered sequence of zero or more values (primitives or structured types, separated by commas).

The fact that the names of properties are enclosed in *quotation marks* and the use of ":" are marks of identity that help to visually identify that text documents are actually instances written in JSON (instead of e.g. C++ or Java data structures).

Example of JSON document providing some metadata about a picture.

```
{
    "width":  800,
    "height": 600,
    "title":  "View from 15th Floor",
    "thumbnail": {
        "url":    "http://www.example.com/image/481989943",
        "height": 125,
        "width":  100
    },
    "animated" : false,
    "ids": [116, 943, 234, 38793]
}
```

In the example above, a *picture* is represented as an object with two numerical properties ("width" and "height"), a string property ("title"), an object property ("thumbnail"), a boolean property ("animated") and an array of numbers as a property ("ids"). Nothing in the document indicates that it is describing a *picture*. The reason is that we are supposed to assign the data structure to a variable the name of which will tell us what the variable is about. This is shown in the example below.

Assigning a JSON structure to a variable in JavaScript

```
picture={
    "width":  800,
    "height": 600,
    "title":  "View from 15th Floor",
    "thumbnail": {
        "url":    "http://www.example.com/image/481989943",
        "height": 125,
        "width":  100
    },
    "animated" : false,
    "ids": [116, 943, 234, 38793]
}
```

See the Section KVP GET client request for the way to incorporate a JSON document into JavaScript code on the fly.

6.3. If we had XML, why do we need JSON?

In the web, we find several comparisons between XML and JSON, some of them trying to do statistical analysis on some criteria, such as verbosity or performance. Some others (many, actually) are more based on opinions than in facts. This document will try to escape this debate and focus on practical facts.

XML was designed by a consensus process in several Internet groups and became a W3C recommendation on February 10th, 1998 as a document standard completely independent from any programming language. Since then, hundreds of document formats based on XML syntax have been proposed, including Really Simple Syndication (RSS), Atom, Simple Object Access Protocol (SOAP), and Extensible Hypertext Markup Language(XHTML), Office Open XML, OpenOffice.org, Microsoft .NET Framework and many others. OGC has adopted XML for many of its web service messages and for several data formats, including the Geography Markup Language(GML), WaterML, Sensor Model Language (SensorML), Geospatial User Feedback (GUF), etc. XML has some interesting additional components: XML Schema/RelaxNG/Schematron provide ways to restrict the classes and data types to a controlled set of them. Actually, all document formats cited before provide a some form of schema document that accompanies the standard document. By providing schema files, standards incorporate a very simple way to check if a document follows the standard data model: a process executed by automatic tools that is called "XML validation". Other components of XML family are XPath (a query language), Extensible Stylesheet Language Transformations (XSLT), etc. With time, these components have been implemented in many programming languages and tools.

JSON history is completely different. JSON was introduced in 1999 as a subset of the JavaScript language that extracted the essentials for defining data structures. This original idea is stated in the RFC7159: "JSON's design goals were to be minimal, portable, textual, and a subset of JavaScript". After 2005, JSON became popular and is used in the Application Programing Interfaces (APIs) of many web services, including those of well-known companies such as Yahoo! or Google. Currently, JSON is no longer restricted to web browsers, because JavaScript can now be used in web services

and in standalone applications (e.g. using node.js) and also because there are libraries that can read and write JSON for almost every programming language (see a long list of programming languages that have some sort of JSON libraries at http://json.org).

6.3.1. The secret of JSON success in the web

Even if this is a matter of opinion, there are some practical reasons that have helped to make JSON the favorite data encoding for many people.

- XML is not easy to learn. JSON format is so simple that it can be explained in a few lines (as has been done in The JSON format). XML requires some knowledge about namespaces and namespace combinations. It also requires some knowledge about classes, complex data types, class generalizations, etc. None of this is currently present in JSON in its basic form.
- XML is defined to be extensible, but XSD Schema validation is very restrictive in extensibility (at least in the way it is used in the OGC standards). In practice, extension points need to be "prepared" by the designer of the original standard to be extended. For example, adding an element to a class in a non-initially foreseen place results in an error during validation. In many occasions the only solution is changing the class name by generalization, giving up descend compatibility. In that sense, RelaxNG validation was designed with extensibility in mind but is not commonly used yet (it is now the recommended validation language for Atom feeds and OWS Context Atom encoding). JSON is used in a way that it can be extended (e.g. adding properties to objects is allowed without breaking compatibility). By default, JSON schema (see JSON Schema) respects this extensibility.
- JSON relies on the simplicity of JavaScript in three important ways:
  - JSON (and JavaScript) have a very limited set of data types. All numbers are "Number" (there is no distinction between float, integer, long, byte,...) and all strings are "String". Arrays and Objects are almost identical; actually Arrays are Objects with numerical consecutive property names.
  - In JavaScript, a JSON document can be converted in an JavaScript data structure (e.g. an object tree) with a single function, instead of a complicated chain of XML DOM data access function calls for each property needed to extract from an XML document. Unfortunately, this has nothing to do with XML or JSON, but in the way the XML DOM was initially implemented. Nevertheless, it has influenced the view that "JSON is simple; XML is complicated".
  - JSON objects do not rely on explicit classes and data types. Even the concept of "data constructor" that was present in early versions of JavaScript it is not recommended anymore. Objects are created on-the-fly and potentially all objects in JSON (and in JavaScript) have a different data structure. However, in practical implementations, many objects and object arrays will mimic the same common pattern.
  - JSON objects can be direct inputs of JavaScript API functions simplifying extensibility of APIs. All JavaScript functions can potentially have a very limited number of parameters, if some of them are JSON objects. New optional parameters can be introduced to these objects without changing the API.

As you will discover in the next sections of this document, a rigorous application of JSON in OGC services will require adoption of new additions to JSON, such as JSON validation and JSON-LD

resulting in a not-so-simple JSON utilization.

Chapter 7. From JSON to JSON-LD

7.1. What is JSON-LD

JSON-LD is a lightweight syntax to encode Linked Data in JSON. Its design allows existing JSON to be interpreted as Linked Data with minimal changes. JSON-LD is 100% compatible with JSON. JSON-LD introduces a universal identifier mechanism for JSON via the use of a Uniform Resource Identifier (URI), a way to associate data types with values.

JSON-LD is considered another Resource Description Framework (RDF) encoding, for use with other Linked Data technologies like SPARQL. Tools are available to transform JSON-LD into other RDF encodings like Turtle (such as the JSON-LD playground).

The authors of this document perceive the current documentation of JSON-LD as confusing. That is why another approach in explaining how to use JSON-LD is presented here.

7.2. Applying JSON-LD to JSON objects: minimum example

The main objective of JSON-LD is to define object identifiers and data types identifiers, on top of JSON objects. The identifiers used for objects and data types are unique URIs that provide extra semantics because they reuse definitions on the web (semantic web).

First of all, JSON-LD defines two specific properties for each object: @id and @type that can be populated with the identifier of the object and the complex data type identifier of the JSON object. Please note that even if JSON and JavaScript do **not** provide classes, in JSON-LD we assign a class name (a.k.a. data type) to an object. In this example we start by defining the Mississippi river.

Minimum example of a river object as a JSON-LD object

```
{
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "@type": "http://dbpedia.org/ontology/River"
}
```

Within the context of RDF, a triple is a construct built from a subject, predicate and an object. The conversion to RDF results in a single triple stating that the Mississippi river is of a river type.

Conversion to the minimum example of a river object to RDF triples

```
<http://dbpedia.org/page/Mississippi_River> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://dbpedia.org/ontology/River> .
```

NOTE	In this section, all conversions to RDF triples have been derived automatically for the JSON-LD precedent examples using the JSON-LD playground

7.3. JSON-LD encoding for JSON objects properties

To add a property to the Mississippi river we should define the semantics of the property by associating a URI to it. To do this we need to also add a @context property. Here we have two possibilities:

- reuse a preexisting vocabulary
- create our own vocabulary

In this example we are adding a *name* to a river resulting on a second triple associated to the object id. In this case we reuse the schema.org vocabulary to define the semantics of the word *name*. Note that http://schema.org/ is actually a URL to a JSON-LD @context document that is in the root of the web server defining the actual and complete schema.org vocabulary.

Adding a name property to a JSON-LD object using a pre-existing vocabulary

```
{
    "@context": "http://schema.org/",
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "@type": "http://dbpedia.org/ontology/River",
    "name": "Mississippi river"
}
```

Conversion to a named river object encoded as RDF triples

```
<http://dbpedia.org/page/Mississippi_River> <http://schema.org/name> "Mississippi
river" .
<http://dbpedia.org/page/Mississippi_River> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://dbpedia.org/ontology/River> .
```

As a second alternative, when the appropriate vocabulary in the JSON-LD format is not available, we can define the needed term on-the-fly as embedded content in the @context section of the JSON instance.

Adding a name property to JSON-LD object defined elsewhere

```
{
    "@context": {
        "name": "http://www.opengis.net/def/ows-common/name"
    },
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "@type": "http://dbpedia.org/ontology/River",
    "name": "Mississippi river"
}
```

It is also possible to combine two vocabularies, one pre-existing and another embedded. This could be particularly useful if, in the future, the OGC Web Services (OWS) Common standard releases a vocabulary for OGC thereby requiring other OGC web services to extend it. Note that, in this case,

@context is defined as an array of an external vocabulary and an internal enumeration of property definitions (in the following example an enumeration of one element).

Two properties defined combining the two alternatives described before

```
{
    "@context": ["http://schema.org/", {
        "bridges": "http://www.opengis.net/def/ows-common/river/bridge"
    }],
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "@type": "http://dbpedia.org/ontology/River",
    "name": "Mississippi river",
    "bridges": ["Eads Bridge", "Chain of Rocks Bridge"]
}
```

Conversion to a named river object encoded as RDF triples

```
<http://dbpedia.org/page/Mississippi_River> <http://schema.org/name> "Mississippi
river" .
<http://dbpedia.org/page/Mississippi_River> <http://www.opengis.net/def/ows-
common/river/bridge> "Chain of Rocks Bridge" .
<http://dbpedia.org/page/Mississippi_River> <http://www.opengis.net/def/ows-
common/river/bridge> "Eads Bridge" .
<http://dbpedia.org/page/Mississippi_River> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://dbpedia.org/ontology/River> .
```

7.4. Using namespaces in JSON-LD

Now we can refine the example and provide a more elegant encoding introducing the definition of abbreviated namespaces and their equivalent URI namespace.

Using abbreviated namespaces in JSON-LD

```
{
    "@context": ["http://schema.org/", {
        "owscommon":"http://www.opengis.net/def/ows-common/",
        "page": "http://dbpedia.org/page/",
        "dbpedia": "http://dbpedia.org/ontology/",
        "bridges": "owscommon:river/bridge"
    }],
    "@id": "page:Mississippi_River",
    "@type": "dbpedia:River",
    "name": "Mississippi river",
    "bridges": ["Eads Bridge", "Chain of Rocks Bridge"]
}
```

7.5. Defining data types for properties in JSON-LD

By default, JSON-LD considers properties as strings. JSON-LD also permits definition of data types not only for the objects but also for individual properties. It is common to define numeric data types.

Adding data types to properties

```
{
    "@context": ["http://schema.org/", {
        "owscommon":"http://www.opengis.net/def/ows-common/",
        "page": "http://dbpedia.org/page/",
        "dbpedia": "http://dbpedia.org/ontology/",
        "bridges": "owscommon:river/bridge",
        "length": {
            "@id": "http://schema.org/distance",
            "@type": "xsd:float"
        }
    }],
    "@id": "page:Mississippi_River",
    "@type": "dbpedia:River",
    "name": "Mississippi river",
    "bridges": ["Eads Bridge", "Chain of Rocks Bridge"],
    "length": 3734
}
```

Conversion of the length of a river object to RDF triples

```
<http://dbpedia.org/page/Mississippi_River> <http://schema.org/distance> "3734"
^^<http://www.w3.org/2001/XMLSchema#float> .
[...]
```

7.6. Ordered and unordered arrays in JSON-LD

An interesting aspect of JSON-LD is that it overwrites the behavior of JSON arrays. In JSON, arrays of values are sorted *lists* but in JSON-LD arrays are *sets* with no order. This way, in the previous examples, *bridges* is an array but the conversion to RDF is done in a way that "Eads Bridge" and "Chain of Rocks Bridge" are associated with the Mississippi river with no order. In general, this is not a problem because most arrays are only *sets* of values. Nevertheless, sometimes order is important for example in a list of coordinates representing a line or a polygon border (imagine what could happen if only one coordinate is out or order!!). Fortunately, there is a way to declare that the array values order is important: using "@container": "@list".

Example where the order of the list of bridges is important

```
{
    "@context": ["http://schema.org/", {
        "owscommon":"http://www.opengis.net/def/ows-common/",
        "page": "http://dbpedia.org/page/",
        "dbpedia": "http://dbpedia.org/ontology/",
        "bridges": {
            "@id": "owscommon:river/bridge",
            "@container": "@list"
        }
   }],
    "@id": "page:Mississippi_River",
    "@type": "dbpedia:River",
    "name": "Mississippi river",
    "bridges": ["Eads Bridge", "Chain of Rocks Bridge"]
}
```

Transformation, to RDF triples, of a list of bridges where order is important

```
<http://dbpedia.org/page/Mississippi_River> <http://schema.org/name> "Mississippi
river" .
<http://dbpedia.org/page/Mississippi_River> <http://www.opengis.net/def/ows-
common/river/bridge> _:b0 .
<http://dbpedia.org/page/Mississippi_River> <http://www.w3.org/1999/02/22-rdf-syntax
-ns#type> <http://dbpedia.org/ontology/River> .
_:b0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#first> "Eads Bridge" .
_:b0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#rest> _:b1 .
_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#first> "Chain of Rocks Bridge" .
_:b1 <http://www.w3.org/1999/02/22-rdf-syntax-ns#rest> <http://www.w3.org/1999/02/22
-rdf-syntax-ns#nil> .
```

Please note that lists of lists are not allowed in JSON-LD making it impossible to transform two-dimensional (2D) arrays of coordinates. This issue is being discussed in [Geospatial_dimension_in_JSON].

A special kind of data type is "@id". This indicates that a property points to another object *id* that can be in the same document or elsewhere in the linked data web. This is the way that JSON-LD is able to define links between objects as previously discussed in JSON-LD.

```
{
    "@context": ["http://schema.org/", {
        "owscommon":"http://www.opengis.net/def/ows-common/",
        "page": "http://dbpedia.org/page/",
        "dbpedia": "http://dbpedia.org/ontology/",
        "wiki": "http://en.wikipedia.org/wiki/Mississippi_River",
        "describedBy": {
            "@id": "http://www.iana.org/assignments/relation/describedby",
            "@type": "@id"
        }
    }],
    "@id": "page:Mississippi_River",
    "@type": "dbpedia:River",
    "name": "Mississippi river",
    "describedBy":  "wiki:Mississippi_River"
}
```

Conversion to a river object related to another object encoded as RDF triples

```
<http://dbpedia.org/page/Mississippi_River> <http://schema.org/name> "Mississippi
river" .
<http://dbpedia.org/page/Mississippi_River>
<http://www.iana.org/assignments/relation/describedby>
<http://en.wikipedia.org/wiki/Mississippi_RiverMississippi_River> .
<http://dbpedia.org/page/Mississippi_River> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> <http://dbpedia.org/ontology/River> .
```

7.7. The geometrical dimension in JSON

One of the main purposes of OGC is providing ways to represent the geospatial dimension of data; a representation for geometries. In the past, OGC has done this in several ways, some of the most recognized ones are:

- GML (Geographic Markup Language): a XML encoding for geospatial features exchange that mainly focus on providing geospatial primitives encoded in XML. Other XML encodings use it as a basis, such as CityGML, WaterML, O&M, IndoorML, etc.
- KML: a XML encoding for vector features, mainly focused on presentation in a virtual globe.
- WKT (Well Known Text): a textual encoding for vector features, to be used in geospatial SQL or SparQL queries and in OpenSearch Geo.
- GeoRSS: a XML encoding for inserting geospatial geometries in RSS and atom feeds.
- GeoSMS: a compact textual encoding for positions in SMS messages.

For the moment, there is no agreement for JSON encoding for geospatial features in OGC. This section discusses several alternatives.

7.7.1. Modeling features and geometries

The ISO 19109 *General Feature Model* discusses aspects of defining features. The ISO 19109 is generally accepted by the OGC community that includes many of its concepts in the OGC 08-126 *The OpenGIS® Abstract Specification Topic 5: Features*.

Next figure describes the most abstract level of defining and structuring geographic data. In the context of a geographic application, the real world phenomena are classified into feature types that share the same list of attribute types. This means that if, for example, the geographical application is capturing protected areas, a *protected area* feature type will define the attributes to capture it and all protected areas will share the same data structure.

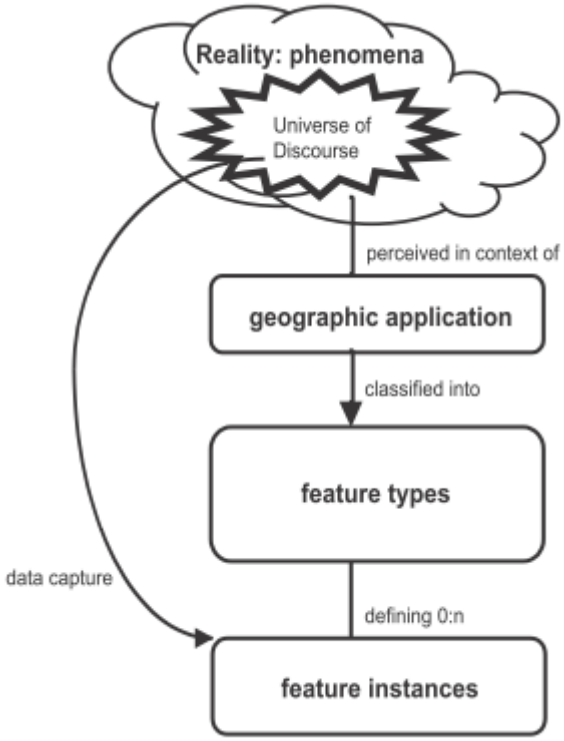

Figure 1. The process from universe of discourse to data

In practice, and following the same example, this means that there will be a *feature catalogue* where an abstract *protected area* is defined as having a multi-polygon, a list of ecosystem types, a list of ecosystem services, a elevation range, a year of definition and the figure of protection used, etc.

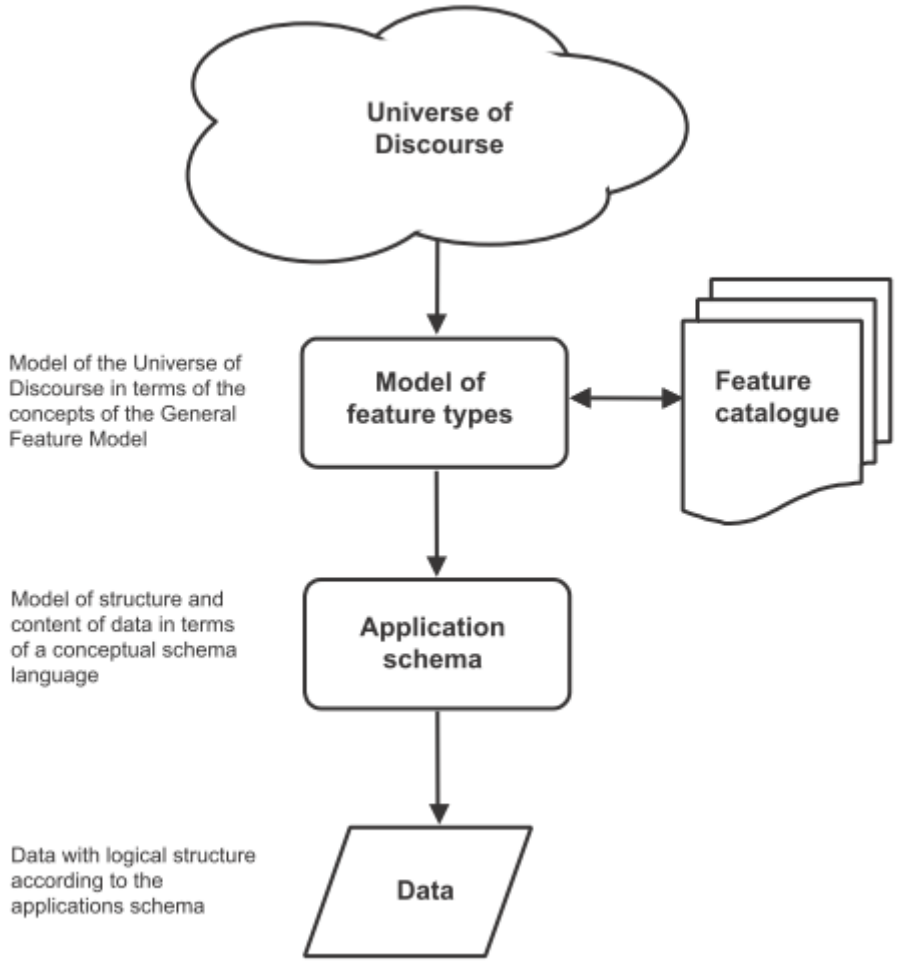

Figure 2. From reality to geographic data

This feature type will be formalized in an application schema. Here, we present a table as a formal way to define the attributes of the protected areas *feature type*.

Table 2. Protected area feature type attributes

Attribute	**Type**	**Multiplicity**
Official border	Multi-polygon	one
Influence area	Multi-polygon	one
Name	String	one or more
Ecosystem type	String	one or more
Ecosystem service	String	one or more
Elevation range	Float	two
Year of definition	Integer	zero or one
Figure of protection	String	zero or one

This way of defining features is basic for the OGC community. GML have included the concept of the application schema from its earlier versions (i.e. an XML Schema). Nevertheless, there are formats that does not follow explicitly the same approach. For example, GeoRSS uses a fixed structure for attributes (common for all features; whatever the feature type) and adds a geometry. KML did not included the capacity to group features in features types until version 2.2 (the first OGC adopted community standard), and this version 2 is the first one to allow more that one

property per feature. It includes a <Schema> tag to define feature types and its property names in a section of the document. Later, the feature type names can be used in PlaceMarks as part of the "ExtentedData/SchemaData" tag.

In the next subsections we will see how JSON can be used in different ways, some of them being compliant to the ISO General Feature Model.

7.7.2. GeoJSON

After years of discussion, in August 2016 the IETF RFC7946 was released, describing the GeoJSON format. GeoJSON is self-defined as "a geospatial data interchange format based on 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."

It defines the following object types "Feature", "FeatureCollection, "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", "MultiPolygon", and "GeometryCollection".

GeoJSON presents some contradictions about complex data types: JSON has no object type concept but GeoJSON includes a "type" property in each object it defines, to declare the type of the object. In contrast, GeoJSON does not include the concept of *feature type*, in the GFM sense, as will be discussed later.

GeoJSON presents a feature collection of individual features. Each Feature has, at least 3 "attributes": a fixed value "type" ("type":"Feature"), a "geometry" and a "properties". Geometry only have 2 "attributes": "type" and "coordinates":

- "type" can be: "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", "MultiPolygon", and "GeometryCollection".
- "coordinates" is based in the idea of position. A position is an array of 2 [long, lat] or 3 numbers [long, lat, h]. The data type of "coordinates" depends on the type of "geometry":
  - in Point, "coordinates" is a single position
  - in a LineString or MultiPoint, "coordinates" is an array of positions
  - in a Polygon or MultiLineString, "coordinates" is an array of LineString or linear ring
  - in a MultiPolygon, "coordinates" is an array of Polygon

There is no specification on what "properties" can contain so implementors are free to provide feature description composed by several attributes in it.

Example of GeoJSON file describing a protected area (coordinates are dummy)

```
{
    "type": "FeatureCollection",
    "features": [{
        "type": "Feature",
        "geometry": {
            "type": "MultiPolygon",
            "coordinates": [
              [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0,
2.0]]],
              [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
               [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
              ]
        },
        "id": "http://www.ecopotential.org/sierranevada",
        "bbox": [100.0, 0.0, 103.0, 3.0],
        "properties": {
            "name": "Sierra Nevada",
            "ecosystemType": "Mountain",
            "ecosystemService": ["turism", "biodiversity reserve"],
            "elevationRange": [860, 3482],
            "figureOfProtection": "National park"
        }
    }
}
```

GeoJSON particularities

A list of considerations extracted from the RFC 7946 require our attention:

- Features can have ids: "If a Feature has a commonly used identifier, that identifier SHOULD be included as a member of the Feature object with the name *id*"
- Features can have a "bbox": "a member named *bbox* to include information on the coordinate range. The value of the bbox member MUST be an array of numbers, with all axes of the most southwesterly point followed by all axes of the more northeasterly point."
- Coordinates are in CRS84 + optional *ellipsoidal* height. "The coordinate reference system for all GeoJSON coordinates is a geographic coordinate reference system, using the World Geodetic System 1984 (WGS 84) [WGS84] datum, with longitude and latitude units of decimal degrees. This is equivalent to the coordinate reference system identified by the Open Geospatial Consortium (OGC) URN urn:ogc:def:crs:OGC::CRS84. An OPTIONAL third-position element SHALL be the height in meters above or below the WGS 84 reference *ellipsoid*."
- "Members not described in RFC 7946 ("foreign members") MAY be used in a GeoJSON document."
- GeoJSON elements cannot be recycled in other places: "GeoJSON semantics do not apply to foreign members and their descendants, regardless of their names and values."
- The GeoJSON types cannot be extended: "Implementations MUST NOT extend the fixed set of

GeoJSON types: FeatureCollection, Feature, Point, LineString, MultiPoint, Polygon, MultiLineString, MultiPolygon, and GeometryCollection."

- "The media type for GeoJSON text is *application/geo+json*"

GeoJSON honors the simplicity of the JSON and JavaScript origins. GeoJSON defines *Feature collections* and *Features* but does not contemplate the possibility of defining Feature types or associating a Feature to a feature type. In our opinion this is consistent with JSON itself, that does not include the *data type* concept, but diverges from the General Feature Model (GFM). In practice, this means that the number and type of the properties of each feature can be different. With this level of flexibility, GeoJSON is not the right format for exchanging data between repositories based on the GFM. In the introduction, RFC7946 compares GeoJSON with WFS outputs. This comparison is an oversimplification; even if the response of a WFS return a feature collection, RFC7946 overlooks that WFS deeply uses the *Feature Type* concept that is missing in GeoJSON.

7.7.3. OGC needs that GeoJSON does not cover

In GeoJSON:

- There is no feature model. Sometimes there is the question about GeoJSON covering the OGC GML Simple Features. This is not the case: GML Simple Features uses the GFM in a simplified way but GeoJSON ignores the GFM.
- There is no support for CRSs other than CRS84.
- The geometries cannot be extended to other types.
- There is no support for the time component.
- There is no information on symbology.

In practice, this means that GeoJSON can only be used in similar circumstances where KML can be used (but without symbology). GeoJSON cannot be used in the following use cases:

- When there is a need to communicate features that are based on the GFM and that depend on the feature type concept.
- When there is a need to communicate features that need to be represented in other CRS that CRS84, such as the combination of UTM/ETRS89.
- When the time component needs to be considered as a coordinate.
- When Simple geometries are not enough and there is a need for circles, arcs of circle, 3D meshes, etc.
- When coverage based (e.g. imagery) or O&M based (e.g. WaterML) data need to be communicated.
- When there is a need to use JSON-LD and to connect to the *linked data*.

In these cases there are three possible options:

- Simplify our use case until it fits in the GeoJSON requirements (see Simplify our use case until it fits in the GeoJSON requirements)
- Extend GeoJSON. In the "feature" or in the "properties" element of each FeatureCollection,

include everything not supported by the GeoJSON (see Extend GeoJSON)

- Deviate completely from the GeoJSON and use another JSON model for geometries (see Another JSON model for geometries)

Lets explore these possibilities on one by one.

Simplify our use case until it fits in the GeoJSON requirements

In our opinion, GeoJSON is not an exchange format (as said by the RFC7946) but a visualization format ideal for representing data in web browsers. In that sense, the comparison in RFC7946 introduction with KML is appropriate. As said before, JSON lacks any visualization/portrayal instructions so symbolization will be applied in the client site or will be transmitted in an independent format.

In case where GeoJSON is a possible output of our information (complemented by other data formats), there is no problem on adapting our data model to the GeoJSON requirements (even if we are going to lose some characteristics) because we also offer other alternatives. In these scenarios, we will not recommend the GeoJSON format as a exchange format but as a visualization format. In OGC services, a WMS could server maps in GeoJSON and WFS can consider GeoJSON as one of the provided formats.

This is the way we can simplify our requirements to adapt them to JSON:

- Even if features are of the same feature type and share a common structure, we forget about this when transforming to JSON.
- If there is more than one geometric property in the features, select one geometric property for the geometries and remove the rest.
- Move all other feature properties inside the "properties" attribute. This will include, time, feature metadata, symbolization attributes, etc.
- Convert your position to CRS84.
- Convert any geometry that can not be directly represented in GeoJSON (e.g a circle) to a sequence of vertices and lines.

Extend GeoJSON

The GeoJSON extensibility is limited by the interpretation of the sentence in the IETF standard "Implementations MUST NOT extend the fixed set of GeoJSON types: FeatureCollection, Feature, Point, LineString, MultiPoint, Polygon, MultiLineString, MultiPolygon, and GeometryCollection.". The sentence is a bit ambiguous but, in general, you are allowed to include any content in the "properties" section, and there is no clear objection on adding attributes to "feature" (even most GeoJSON parsers will ignore them). It seems that you are neither allowed to invent new geometries nor to modify the current existing ones. With this limitations in mind, be can do several things, including the ones covered in the following subsections.

Adding visualization to GeoJSON

For some people, visualization is an important aspect that should be in GeoJSON and has provided some approach for including visualization styles.

- An style extension from MapBox includes terms in "properties" of the "Feature"s. https://github.com/mapbox/simplestyle-spec/tree/master/1.1.0

Mapbox simplestyle-spec to add some styles to GeoJSON

```
{
    "type": "FeatureCollection",
    "features": [{ "type": "Feature",
        "geometry": {
            "type": "Polygon",
            //...
        },
        "properties": {
            "stroke": "#555555",
            "stroke-opacity": 1.0,
            "stroke-width": 2,
            "fill": "#555555",
            "fill-opacity": 0.5
        }
    }]
}
```

- Leaflet.geojsonCSS is an extension for Leaflet to support rendering GeoJSON with css styles in a "style" object in "Feature". https://github.com/albburtsev/Leaflet.geojsonCSS

Leaflet.geojsonCSS to add some styles to GeoJSON

```
{
    "type": "FeatureCollection",
    "features": [{ "type": "Feature",
        "geometry": {
            "type": "Polygon",
        },
        "style": {
            "color": "#CC0000",
            "weight": 2,
            "fill-opacity": 0.6,
            "opacity": 1,
            "dashArray": "3, 5"
        },
        "properties": {
            //...
        }
    }]
}
```

Other CRS representation for the same geometry

Sometimes it could be necessary to distribute your data in other CRSs that are not CRS84. As long as you are not doing this in the "geometry" part of the GeoJSON, you are allowed to do this. You can

even reuse the *geometry* object in the *properties* section, knowing that they will be not considered by pure GeoJSON parsers.

Example of GeoJSON file describing a protected area also in EPSG:25831 (coordinates are dummy).

```
{
    "type": "FeatureCollection",
    "features": [{
        "type": "Feature",
        "geometry": {
            "type": "MultiPolygon",
            "coordinates": [
              [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0,
2.0]]],
              [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
               [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
              ]
        },
        "id": "http://www.ecopotential.org/sierranevada",
        "bbox": [100.0, 0.0, 103.0, 3.0],
        "bboxCRS": {
            "bbox": [500100.0, 4600000.0, 500103.0, 4600003.0],
            "crs": "http://www.opengis.net/def/crs/EPSG/0/25831",
        }
        "properties": {
            "geometryCRS": {
                "type": "MultiPolygon",
                "crs": "http://www.opengis.net/def/crs/EPSG/0/25831",
                "coordinates": [
                  [[[500102.0, 4600002.0], [500103.0, 4600002.0], [500103.0,
4600003.0], [500102.0, 4600003.0], [500102.0, 4600002.0]]],
                  [[[500100.0, 4600000.0], [500101.0, 4600000.0], [500101.0,
4600001.0], [500100.0, 4600001.0], [500000.0, 4600000.0]],
                   [[500100.2, 4600000.2], [500100.8, 4600000.2], [500100.8,
4600000.8], [500100.2, 4600000.8], [500100.2, 4600000.2]]]
                  ]
            },
            "name": "Sierra Nevada",
            "ecosystemType": "Mountain",
            "ecosystemService": ["turism", "biodiversity reserve"],
            "elevationRange": [860, 3482],
            "figureOfProtection": "National park"
        }
    }
}
```

Another JSON model for geometries

The last alternative is to completely forget about GeoJSON and define your own encoding strictly following the GFM.

Example of JSON file describing a protected area without using GeoJSON (coordinates are dummy).

```
{
    "id": "http://www.ecopotential.org/sierranevada",
    "featureType": "ProtectedArea",
    "officialBorder": {
        "type": "MultiPolygon",
        "crs": "http://www:opengis.net/def/crs/OGC/1/3/CRS84",
        "coordinates": "[
          [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
          [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
           [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
          ]"
    }
    "infuenceArea": {
        "type": "MultiPolygon",
        "crs": "http://www:opengis.net/def/crs/OGC/1/3/CRS84",
        "coordinates": "[
          [[[99.0, 1.0], [113.0, 1.0], [113.0, 5.0], [99.0, 5.0], [99.0, 1.0]]],
          [[[80.0, -10.0], [110.0, -10.0], [110.0, 11.0], [80.0, 11.0], [90.0,
-10.0]]],
           [[90.2, -0.2], [108.8, -0.2], [108.8, 1.8], [108.2, 1.8], [90.2, -0.2]]]
          ]"
    }
    "name": "Sierra Nevada",
    "ecosystemType": "Mountain",
    "ecosystemService": ["turism", "biodiversity reserve"],
    "elevationRange": [860, 3482],
    "figureOfProtection": "National park"
}
```

The previous example has been defined in a way that is compatible with JSON-LD and can be automatically converted to RDF if a @context is provided. Please, note that coordinates are expressed as strings to force a JSON-LD engine to ignore them and consider them string. This notation has been suggested in OGC 16-051 JavaScript JSON JSON-LD ER. We call it JSON double encoding as the string is written in a notation that is fully compatible with JSON and the content of "coordinates" can be parsed into a JSON object and converted into a multidimensonal array easily.

JSON for coverages

Since the first versions of the HTML and web browsers, it was possible to send a JPEG or a PNG to the browser and show it. With addition of HTML DIV tags, it was possible to overlay them in a layer stack and show them. WMS took advantage of it to create map browsers on the web. The main problem with this approach was that the "map" could not be manipulated in the client, so symbolization of the map had to be done in the server (and the interaction with the data became slow and limited. Modern web browsers implementing HTML5 allow for controlling pixel values on the screen representation in what is called the *canvas*. This capability allows sending an array of values from a coverage server to web browser that can be converted into a RGBA array and then represented in the canvas. This represents an evolution of what was possible in the past. By implementing this strategy it is possible to control the coloring of "maps" directly in the browser

and to make queries on the actual image values in the client. The map becomes a true coverage.

A good coverage needs to be described through a small set of metadata that defines the domain (the grid), the range values (the data) and the range meaning (the data semantics). This is exactly what the Coverage Implementation Schema (CIS) does (formerly known as GMLCov).

The idea of creating a JSON GMLCov associated to a JSON coverage appears for the first time in the section 9 of the OGC 15-053r1 Testbed-11 Implementing JSON/GeoJSON in an OGC Standard Engineering Report. This idea was taken by the MELODIES FP7 project (http://www.melodiesproject.eu/), and described as a full specification, as well as implemented as an extension of the popular map browser *Leaflet*. The description of the approach can be found here https://github.com/covjson/specification. A complete demostration on how it works can be found here: https://covjson.org/playground/ (tested with Chrome).

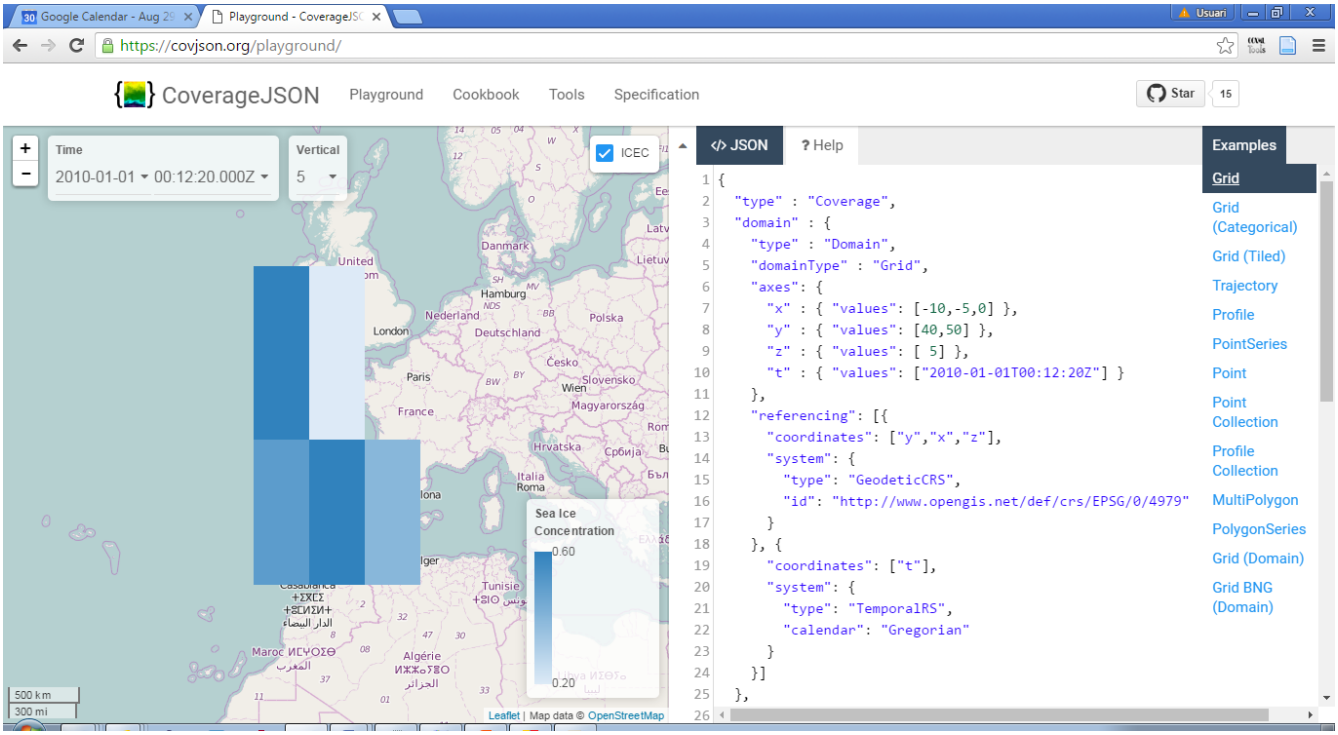

Figure 3. CoveragesJSON playground dummy example for continuous values in http://covjson.org

CoverageJSON is a demonstration of what can be done with coverages in the browsers. On our opinion, this approach will improve the user experience working with imagery and other types of coverages in web browsers. Unfortunately, the CoverageJSON defined by MELODIES deviates significantly from the OGC CIS. Actually CoverageJSON redesigns CIS to replicate most of the concepts in a different way and adds some interesting new concepts and functionalities of its own.

To better align with OGC coverages representation, a new JSON encoding is introduced in the OGC CIS 1.1. In this case, the JSON encoding strictly follows the new CIS 1.1 UML model. This encoding is presented in section 13 on CIS 1.1 and includes a set of JSON schemas. In addition, section 14 adds requirements for JSON-LD that are complemented by JSON-LD context files. Several examples are also informative material accompanying the CIS 1.1 document. More details can be found also in this ER: OGC 16-051 JavaScript JSON JSON-LD ER.

Example of a regular grid represented as a CIS JSON file

```
{
```

```
    "@context": ["http://localhost/json-ld/coverage-context.json", {"examples":
"http://www.opengis.net/cis/1.1/examples/"}],
    "type": "CoverageByDomainAndRangeType",
    "id": "examples:CIS_10_2D",
    "domainSet":{
        "@context": "http://localhost/json-ld/domainset-context.json",
        "type": "DomainSetType",
        "id": "examples:CIS_DS_10_2D",
        "generalGrid":{
            "type": "GeneralGridCoverageType",
            "id": "examples:CIS_DS_GG_10_2D",
            "srsName": "http://www.opengis.net/def/crs/EPSG/0/4326",
            "axisLabels": ["Lat", "Long"],
            "axis": [{
                "type": "RegularAxisType",
                "id": "examples:CIS_DS_GG_LAT_10_2D",
                "axisLabel": "Lat",
                "lowerBound": -80,
                "upperBound": -70,
                "uomLabel": "deg",
                "resolution": 5
            },{
                "type": "RegularAxisType",
                "id": "examples:CIS_DS_GG_LONG_10_2D",
                "axisLabel": "Long",
                "lowerBound": 0,
                "upperBound": 10,
                "uomLabel": "deg",
                "resolution": 5
            }],
            "gridLimits": {
                "type": "GridLimitsType",
                "id": "examples:CIS_DS_GG_GL_10_2D",
                "srsName": "http://www.opengis.net/def/crs/OGC/0/Index2D",
                "axisLabels": ["i", "j"],
                "axis": [{
                    "type": "IndexAxisType",
                    "id": "examples:CIS_DS_GG_GL_I_10_2D",
                    "axisLabel": "i",
                    "lowerBound": 0,
                    "upperBound": 2
                },{
                    "type": "IndexAxisType",
                    "id": "examples:CIS_DS_GG_GL_J_10_2D",
                    "axisLabel": "j",
                    "lowerBound": 0,
                    "upperBound": 2
                }]
            }
        }
    },
```

```
    "rangeSet": {
        "@context": "http://localhost/json-ld/rangeset-context.json",
        "type": "RangeSetType",
        "id": "examples:CIS_RS_10_2D",
        "dataBlock": {
            "id": "examples:CIS_RS_DB_10_2D",
            "type": "VDataBlockType",
            "values": [1,2,3,4,5,6,7,8,9]
        }
    },
    "rangeType": {
        "@context": "http://localhost/json-ld/rangetype-context.json",
        "type": "DataRecordType",
        "id": "examples:CIS_RT_10_2D",
        "field":[{
            "type": "QuantityType",
            "id": "examples:CIS_RT_F_10_2D",
            "definition": "ogcType:unsignedInt",
            "uom": {
                "type": "UnitReference",
                "id": "examples:CIS_RT_F_UOM_10_2D",
                "code": "10^0"
            }
        }]
    }
}
```

Chapter 8. JSON Best Practices

8.1. Basic JSON considerations

The basics of JSON encoding are defined initially in the RFC4627 (2006) and more recently in the RFC7159 (2013).

The following aspects have been extracted from the RFC7159 and, in the opinion of the authors of this document, should be considered requirements to JSON in OGC implementations of JSON encodings. These aspects go a bit beyond the immediate description of a simple JSON format but are needed to improve JSON interoperability. In the following text, the use of *quotation marks* indicates that the text comes directly from RFC7159 (sometimes with minimal editorial changes).

- An object has "an *unordered* collection of zero or more name/value pairs". "Implementations whose behavior does not depend on member ordering will be interoperable". This means that the order of the properties can change and the object should be considered exactly the same. This is not what happens by default in the case of XML encodings, where any variation in the order of properties generates a validation error. (Note that in most of the classes in OGC UML models, the order of properties is not important, even if, after XML conversion, order becomes an extra imposed restriction).
- "The property names within an object SHOULD be unique". This means that if you need multiplicity *more than one* in a property, it should be defined as a single property of *array* type.
- "There is no requirement that values in an array have to be of the same type.". This means that, e.g. an array presenting an string (quotation marks), a number (no quotation marks), an object (curly brackets) and an array (square brackets) is a perfectly valid array (Note that you can limit this behavior in your application by applying a JSON Schema type restriction to a property).
- "An array is an *ordered* sequence of zero or more values". Note that this means that multiplicity *more than one* is ordered. This is not what happens by default in the case of XML, where multiplicity is unsorted by default (as, it is also the case in most of the OGC UML diagrams too).
- For numbers, "*Infinity*, *NaN*, etc are not permitted".
- For floating point numbers "since software that implements IEEE 754-2008 binary64 (double precision) numbers is generally available and widely used, good interoperability can be achieved by implementations that expect no more precision or range than these provide, in the sense that implementations will approximate JSON numbers within the expected precision. A JSON number such as 1E400 or 3.141592653589793238462643383279 may indicate potential interoperability problems, since it suggests that the software that created it expects receiving software to have greater capabilities for numeric magnitude and precision than is widely available."
- "JSON text SHALL be encoded in UTF-8, UTF-16, or UTF-32. The default encoding is UTF-8, and JSON texts that are encoded in UTF-8 are interoperable". In practice this affects both *names* of properties and *string values*. This is particularly important for non English languages that require to encode characters beyond the *common English* first 128 characters that are shared by most of the encodings.
- In property *names* and *string values* the characters "*quotation mark*, *reverse solidus*, and the

control characters (U+0000 through U+001F) " "must be escaped".

- Any other character in a property *names* and *string values* "may be escaped". There are two main escaping strategies. "It may be escaped by a six-character sequence: a *reverse solidus*, a lowercase letter *u*, and four hexadecimal digits that encodes the character's code point.". "For example, a *reverse solidus* character may be represented as *\u005C*". "Alternatively, there are two-character sequence escape representations of some popular characters" allowed: \" (*quotation mark*), \\ (*reverse solidus*), \/ (*solidus*), \b (*backspace*), \f (*form feed*), \n (*line feed*), \r (*carriage return*) and \t (*tab*). In addition there is the less used "UTF-16 surrogate pair" (e.g. "G clef" character may be represented as "\uD834\uDD1E".
- "Implementations MUST NOT add a byte order mark to the beginning of a JSON text. In the interests of interoperability, implementations that parse JSON texts MAY ignore the presence of a byte order mark rather than treating it as an error.". In UTF8, the *byte order mark* is the byte sequence: 0xEF,0xBB,0xBF.
- "The MIME media type for JSON text is application/json"
- "Security Consideration": Even if "JSON is a subset of JavaScript but excludes assignment and invocation", so that it can not contain code, "to use the "eval()" function to parse JSON texts constitutes an unacceptable security risk, since the text could contain executable code along with data declarations" (text that is not JSON but contains other elements of JavaScript). "The same consideration applies to the use of eval()-like functions in any other programming language".

In addition to these requirements, the authors consider that adding an extra requirement to normal behavior for JSON parsers is interesting: * A parser that finds an object property that it is not able to recognize, should ignore it rather than treating it as an error. (This is not what happens by default in the case of XML validation). In special circumstances JSON can limit this inherent behavior, if conveniently justified, but not in general.

To increase interoperability the authors of this document would like to add considerations for including more simple data types as restrictions of the "string" type. They were taken directly from the "format" parameter of the JSON Schema standard candidate:

- "date-time": Date representation, as defined by RFC 3339, section 5.6.
- "email": Internet email address, as defined by RFC 5322, section 3.4.1.
- "hostname": Internet host name, as defined by RFC 1034, section 3.1.
- "uri": A universal resource identifier (URI), according to RFC3986.

See more detail of these four simple data types below.

NOTE	*format* in JSON Schemas also defines *ipv4* and *ipv6* but these types are not commonly needed in OGC standards.

In addition to these simple types, complex geometrical types are fundamental for OGC and will be discussed in [Geospatial_dimension_in_JSON].

8.1.1. Date-Time format

It follows RFC 3339, section 5.6. This format follows the profile of ISO 8601 for dates used on the Internet. This is specified using the following syntax description, in Augmented Backus-Naur Form (ABNF, RFC2234) notation.

Description of date-time format (including range for each sub-element)

```
   date-fullyear   = 4DIGIT
   date-month      = 2DIGIT  ; 01-12
   date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on
                             ; month/year
   time-hour       = 2DIGIT  ; 00-23
   time-minute     = 2DIGIT  ; 00-59
   time-second     = 2DIGIT  ; 00-58, 00-59, 00-60 based on leap second
                             ; rules
   time-secfrac    = "." 1*DIGIT
   time-numoffset  = ("+" / "-") time-hour ":" time-minute
   time-offset     = "Z" / time-numoffset

   partial-time    = time-hour ":" time-minute ":" time-second
                     [time-secfrac]
   full-date       = date-fullyear "-" date-month "-" date-mday
   full-time       = partial-time time-offset

   date-time       = full-date "T" full-time
```

Example of date-time format

```
{
    "date": "1985-04-12T23:20:50.52Z"
}
```

This subtype is fundamental for OGC and the [Time_instants_as_strings] of this document is devoted to it, in the context of the time dimension described in the [The_time_dimension].

8.1.2. Email format

It follows RFC 5322, section 3.4.1. It specifies Internet identifier that contains a locally interpreted string followed by the at-sign character ("@", ASCII value 64) followed by an Internet domain. The locally interpreted string is either a quoted-string or a dot-atom. If the string can be represented as a dot-atom (that is, it contains no characters other than atext characters or "." surrounded by atext characters), then the dot-atom form SHOULD be used and the quoted-string form SHOULD NOT be used.

Example of email format

```
{
    "email": "mr.bob@opengeospatial.org"
}
```

8.1.3. URI format

It follows RFC3986 and supports both a URI and a URN. The following text represents the common structure of the two previously mentioned types.

URI component parts

```
     foo://example.com:8042/over/there?name=ferret#nose
     \_/   \______________/\_________/ \_________/ \__/
      |           |            |            |        |
   scheme     authority       path        query   fragment
      |   _____________________|__
     / \ /                        \
     urn:example:animal:ferret:nose
```

Example of URI format

```
{
    "url": "http://www.opengeospatial.org/standards"
}
```

8.2. JSON Schema

8.2.1. Why OGC needs JSON validation

OGC is transitioning from standards that were written in plain English to a robust way of written standards based on requirements classes that are linked to conformance test classes. Conformance tests are designed to determine if implementations follow the standard. When an XML encoding is involved, standards that provide XML Schema files defining each data type, provide a straightforward way to check if a document follows the standard: *validating* the XML document with XSD, RelaxNG or Schematron (or a combination of them).

If OGC is going to adopt JSON as an alternative encoding for data models, some automatic way of validating if objects in the JSON file follow the data models proposed by the corresponding standard could be also convenient.

8.2.2. JSON Schema standard candidate

JSON Schema is intended for validation and documentation of data models. It acts in a similar way to XSD for an XML file. Indeed, some applications (such us XML Validator Buddy) are able to combine a JSON file with its corresponding JSON schema to test and validate if the content of the

JSON file corresponds to the expected data model. Several implementations of JSON schema validation are available to be used also on-line. An example is the one available in this URL http://json-schema-validator.herokuapp.com/ and the corresponding opensource code available in github https://github.com/daveclayton/json-schema-validator.

The number of aspects that JSON Schema can validate is lower than the ones that XML Schema can control. Some factors contribute to that:

- JSON objects are considered extendable by default. This means that adding properties not specified in the schema does not give an error as result of validating, by default. This prevents detecting object or attribute names with typos (because they will be confused with *extended* elements) except if they are declared as mandatory (and the will be found *missing* in the validation process). Please note that JSON schema provides a keyword *additionalProperties* that if it is defined as *false*, then JSON object is declared as not extensible (and only the property names enumerated in *properties* are considered valid). Even if this will allow for a more strict validation, we are not recommending it because we will be loosing one of the *advantages* of JSON (this topic has been already discussed in the Basic JSON considerations).
- Objects have no associated data types (or classes). This forces the schema validation to be based in object *patterns* and not in class definitions.
- Another difference is that JSON properties are not supposed to have order so the order of the properties of an object cannot be validated. In many cases this is not a problem, since most of the data models used in OGC do not depend on the order of the properties, even if the XML “tradition _ has imposed this unnecessary restriction (this topic has been already discussed in the Basic JSON considerations).

Unfortunately, JSON schema is a IETF draft that expired in August 2013 and the future of the specification was uncertain. One of the authors blogged that he is forced to abandon the project due to lack of time. The project has been reactivated in September 2016 and a new version of the IETF documents has been released with minim changes. New releases with descend compatibility have been promised.

Note that the Internet media type is "application/schema+json". According to the last available draft of JSON Schema (v4), there is not a new file extension proposed for files storing JSON Schemas. The file extension ".json" is used profusely. To make the situation a bit more complex, there is no documented mechanism to associate a JSON instance to its schema (even if it seems that some applications use "$schema" to do this; as discussed in https://groups.google.com/forum/#!topic/json-schema/VBRZE3_GvbQ). In preparing these examples, we found the need to be able prepare json instances and json schemas with similar file names to make the relation between them more explicit and it was practical to name the schema files ending with "_schema.json".

8.2.3. JSON Schema simple example

Lets use a simple feature example encoded in JSON to later illustrate how JSON Schema is useful for documentation and validation.

Example of a river feature in JSON

```
{
    "river":
    {
        "name": "mississippi",
        "length": 3734,
        "discharge": 16790,
        "source": "Lake Itasca",
        "mouth": "Gulf of Mexico",
        "country": "United States of America",
        "bridges": ["Eads Bridge", "Chain of Rocks Bridge"]
    }
}
```

Now let's define a JSON Schema for validating it. The first thing we need is to start a JSON file with an indication telling everybody that this is a JSON Schema by adding a "$schema" property in the root object of the schema. The value used in this examples reflects the last draft version available some months ago (i.e. v4).

Indication that this file is a JSON Schema that follows the specification draft version 4.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#"
}
```

Title and description are useful properties to describe the schema purpose and the objects and properties it will validate.

Title and description to describe the schema (or the root element).

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "JSON minimal example",
    "description": "Schema for the minimal example of a river description"
}
```

The root element can be an object or an array. In this case we are validating an *object*.

The root object is an object.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "JSON minimal example",
    "description": "Schema for the minimal example that is a river",
    "type": "object"
}
```

Now it is time to enumerate the properties. The properties array allows to enumerate the property

names and to list their attributes. In the next example, there is only one property that is called "river". This property is an object and is declared as required.

The root object has a single property called "river"

```
{
    [...]
    "type": "object",
    "required": ["river"],
    "properties": {
        "river": {
            "type": "object"
        }
    }
}
```

Since *river* is an *object*, we can repeat the previous pattern for it. In particular, a river object has a *name* and this name is an "string".

The river object has also some properties

```
{
    [...]
        "river":
        {
            "type": "object",
            "title": "Minimal River",
            "required": [ "name" ],
            "properties":
            {
                "name": {"type": "string" },
                [...]
            }
        }
    [...]
}
```

A *river* has additional properties and some of them are numeric. Please note that in the case of numeric properties, the numeric allowed range can be indicated using *minimum* and *maximum*. In this case, we are forcing numbers to be non-negative since they represent characteristics that cannot be negative.

The river properties list

```
{
    [...]
            {
                "name": {"type": "string" },
                "length": { "type": "number", "minimum": 0 },
                "discharge": { "type": "number", "minimum": 0 },
                "source": { "type": "string" },
                "mouth": { "type": "string" },
                "country": { "type": "string" },
                [...]
            }
    [...]
}
```

Now we add a river property that is called *bridges* and that can contain a list of bridge names. It is encoded as an array of strings.

One river property is an array

```
{
    [...]
            {
                [...]
                "country": { "type": "string" },
                "bridges": {
                    "type": "array",
                    "items": { "type": "string" }
                }
            }
    [...]
}
```

Finally, we could use one of the JSON online schema validator tools to check the validity of the previous JSON file. There are many online validators and the initial JSON example has been validated with the proposed JSON Schema with the following validators:

- https://json-schema-validator.herokuapp.com/
- http://jsonschemalint.com/#/version/draft-04/markup/json
- http://www.jsonschemavalidator.net/

If we simply change the length of the river to a negative number (e.g. -1) we will get an error report that varies in the text from one implementation to the other but all give us an indication of the problem:

Response of the http://www.jsonschemavalidator.net/

```
Message: Integer -1 is less than minimum value of 0.
Schema path:#/properties/river/properties/length/minimum
```

Response of the https://json-schema-validator.herokuapp.com/

```
[ {
  "level" : "error",
  "schema" : {
    "loadingURI" : "#",
    "pointer" : "/properties/river/properties/length"
  },
  "instance" : {
    "pointer" : "/river/length"
  },
  "domain" : "validation",
  "keyword" : "minimum",
  "message" : "numeric instance is lower than the required minimum (minimum: 0, found:
-1)",
  "minimum" : 0,
  "found" : -1
} ]
```

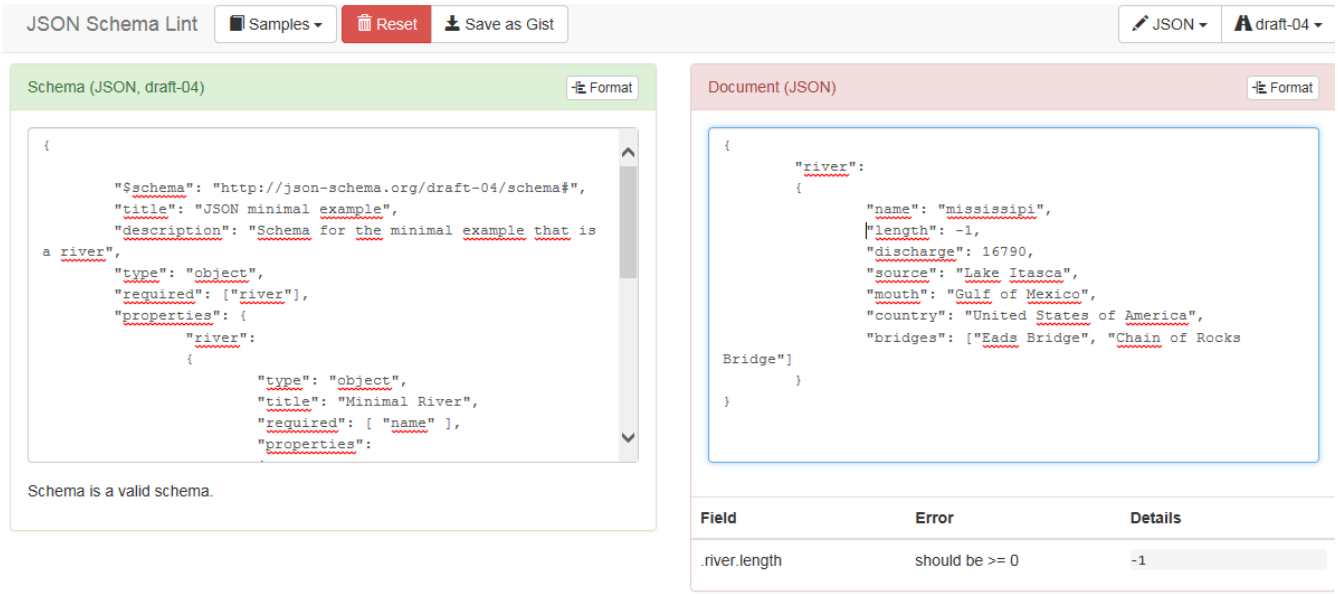

Figure 4. Response of the http://jsonschemalint.com/#/version/draft-04/markup/json

8.2.4. JSON Schema for an object that can represent two things

Lets consider now that I need to encode rivers and lakes. In this case, we will need an object that can present itself either as a river or as a lake. We have already seen an example for a river, and we now present an instance for a lake.

Example of a lake feature in JSON

```
{
    "lake":
    {
        "name": "Tunica Lake",
        "area": 1000,
        "country": "United States of America"
    }
}
```

Obviously, rivers and lakes will have different properties. There is a *oneOf* property in JSON Schema that allows a thing to present more than one alternative definition. This way both, the previous JSON instance for the river and the one in this subsection, will be validated with the same JSON Schema.

Example of a JSON schema to validate a river or a lake

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "oneOf": [
        {
            "title": "JSON minimal river example",
            "description": "Schema for the minimal example that is a river",
            "type": "object",
            "required": ["river"],
            "properties": {
                "river":
                {
                    "type": "object",
                    "title": "Minimal river",
                    "required": [ "name", "length" ],
                    "properties":
                    {
                        "name": {"type": "string" },
                        "length": { "type": "number", "minimum": 0 },
                        "discharge": { "type": "number", "minimum": 0 },
                        "source": { "type": "string" },
                        "mouth": { "type": "string" },
                        "country": { "type": "string" },
                        "bridges": {
                            "type": "array",
                            "items": { "type": "string" }
                        }
                    }
                }
            }
        },{
            "title": "JSON minimal lake example",
            "description": "Schema for the minimal example that is a lake",
```

```
            "type": "object",
            "required": ["lake"],
            "properties": {
                "lake":
                {
                    "type": "object",
                    "title": "Minimal lake",
                    "required": [ "name", "area" ],
                    "properties":
                    {
                        "name": {"type": "string" },
                        "area": { "type": "number", "minimum": 0 },
                        "country": { "type": "string" }
                    }
                }
            }
        }
    ]
}
```

8.2.5. JSON Schema for an array of features

After showing how to do a single feature (i.e. rivers and lakes, each one in an independent JSON document that can be validated with the same JSON Schema) to show how to represent a feature collections as arrays can be useful. Following this approach, we are able to include rivers and lakes as array items in the same JSON file:

Example of a river and a lake feature in JSON. Variant A.

```
[
    {
        "river":
        {
            "name": "mississippi",
            "length": 3734,
            "discharge": 16790,
            "source": "Lake Itasca",
            "mouth": "Gulf of Mexico",
            "country": "United States of America",
            "bridges": ["Eads Bridge", "Chain of Rocks Bridge"]
        }
    },{
        "lake":
        {
            "name": "Tunica Lake",
            "area": 1000,
            "country": "United States of America"
        }
    }
]
```

This can be validated by the following JSON Schema, that is very similar to the last one, but defines the root element as an array of items.

Example of a JSON Schema to validate a river or a lake. Variant A.

```
{

    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "JSON feture array example",
    "description": "Schema for a feature array",
    "type": "array",
    "items": {
        "oneOf":[
        {
            "title": "JSON minimal river example",
            "description": "Schema for the minimal example that is a river",
            "type": "object",
            "required": ["river"],
            "properties": {
                "river":
                {
                    "type": "object",
                    "title": "Minimal river",
                    "required": [ "name", "length" ],
                    "properties":
                    {
```

```
                    "name": {"type": "string" },
                    "length": { "type": "number", "minimum": 0 },
                    "discharge": { "type": "number", "minimum": 0 },
                    "source": { "type": "string" },
                    "mouth": { "type": "string" },
                    "country": { "type": "string" },
                    "bridges": {
                        "type": "array",
                        "items": { "type": "string" }
                    }
                }
            }
        }
    },{
        "title": "JSON minimal lake example",
        "description": "Schema for the minimal example that is a lake",
        "type": "object",
        "required": ["lake"],
        "properties": {
            "lake":
            {
                "type": "object",
                "title": "Minimal lake",
                "required": [ "name", "area" ],
                "properties":
                {
                    "name": {"type": "string" },
                    "area": { "type": "number", "minimum": 0 },
                    "country": { "type": "string" }
                }
            }
        }
    }]
}
}
```

JSON is one of these cases where simplicity is highly appreciated. It could be useful to consider a second alternative, where there is not need to use an object name. Instead we will use a "type" property to differentiate among object types and this will result in a notation with less indentations. A part from being more elegant (what is a matter of opinion) it will result in a much more nice conversion to RDF when JSON-LD @context is introduced later (see Applying JSON-LD to JSON objects: minimum example).

Example of a river and a lake feature in JSON. Variant B.

```
[
    {
        "type": "river",
        "name": "mississippi",
        "length": 3734,
        "discharge": 16790,
        "source": "Lake Itasca",
        "mouth": "Gulf of Mexico",
        "country": "United States of America",
        "bridges": ["Eads Bridge", "Chain of Rocks Bridge"]
    },{
        "type": "lake",
        "name": "Tunica Lake",
        "area": 1000,
        "country": "United States of America"
    }
]
```

This is the corresponding JSON Schema that can be used to validate the array. Note that only "river" and "lake" values are allowed in the "type" key, and any other value will generate a validation error.

Example of a JSON Schema to validate a river or a lake. Variant B.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "JSON feture array example",
    "description": "Schema for a feature array",
    "type": "array",
    "items": {
        "oneOf":[
        {
            "title": "JSON minimal river example",
            "description": "Schema for the minimal example that is a river",
            "type": "object",
            "required": [ "type", "name", "length" ],
            "properties": {
                "type": {"enum": ["river"] },
                "name": {"type": "string" },
                "length": { "type": "number", "minimum": 0 },
                "discharge": { "type": "number", "minimum": 0 },
                "source": { "type": "string" },
                "mouth": { "type": "string" },
                "country": { "type": "string" },
                "bridges": {
                    "type": "array",
                    "items": { "type": "string" }
                }
            }
        },{
            "title": "JSON minimal lake example",
            "description": "Schema for the minimal example that is a lake",
            "type": "object",
            "required": [ "type", "name", "area" ],
            "properties":
            {
                "type": {"enum": ["lake"] },
                "name": {"type": "string" },
                "area": { "type": "number", "minimum": 0 },
                "country": { "type": "string" }
            }
        }]
    }
}
```

In JSON Schema, one can do much more than what has been explained here. Most of the needed characteristics of UML class diagram usually included in OGC and ISO standards, such as, generalization, association, composition, etc can be implemented by JSON Schemas as comprehensively discussed in the OGC 16-051 Testbed 12 A005-2 Javascript JSON JSON-LD ER.

Chapter 9. JSON Encoding

9.1. Links in JSON

9.1.1. Introduction

Following the RFC5988 description, a link is a typed connection between two resources that are identified by internationalized Resource Identifiers (IRIs) [RFC3987], and is comprised of:

- a context IRI,
- a link relation type (an initial list of types is in Section-6.2.2 of RFC5988. E.g.: describedBy),
- a target IRI, and
- optionally, target attributes.

This schema was adopted both by Xlink and Atom in a similar way. The question that is discussed in this section is what is the status of links in JSON and what is the recommendation for the OGC.

9.1.2. JSON-LD

In the next section we will explain JSON-LD in detail (please see Applying JSON-LD to JSON objects: minimum example) but, in this subsection, we can anticipate what is essential in JSON-LD that allow it to work with links. In JSON-LD, objects have an *id* that identifies the context IRI. To do that, JSON objects include a property called *@id*. As we know, a JSON object has properties, and JSON-LD defines how to use a key-value property in a particular way that allows expressing a link from this object to other objects.

Relation between a river and its description expressed in JSON

```
{
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "describedBy": "http://en.wikipedia.org/wiki/Mississippi_River",
    "name": "Mississippi river"
}
```

Adding JSON-LD to a JSON file introduces the possibility to define links as a property name that has a semantics defined by IANA (such as "describedBy") and the value of the property that has an @id type and shall contain a URI. This is done in a special @context property.

Context complex property added to a JSON file that defines describedBy as a relation type IANA describedBy.

```
{
    "@context": {
        "describedBy": {
            "@id": "http://www.iana.org/assignments/relation/describedby",
            "@type": "@id"
        }
    }
}
```

Both JSON fragments can be combined in a single JSON-LD document

Direct combination of the JSON example and the JSON-LD context semantic definition.

```
{
    "@context": {
        "name": "http://schema.org/name",
        "describedBy": {
            "@id": "http://www.iana.org/assignments/relation/describedby",
            "@type": "@id"
        }
    },
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "describedBy": "http://en.wikipedia.org/wiki/Mississippi_River",
    "name": "Mississippi river"
}
```

JSON-LD allows for a more elegant notation that defines abbreviated namespaces that can be reused later.

A more elegant version of the JSON example and the JSON-LD context semantic definition using namespaces.

```
{
    "@context": {
        "iana_rel": "http://www.iana.org/assignments/relation/",
        "dbpedia": "http://dbpedia.org/page/",
        "wiki": "http://en.wikipedia.org/wiki/",
        "schema": "http://schema.org/",

        "name": "schema:name",
        "describedBy": {
            "@id": "iana_rel:describedby",
            "@type": "@id"
        }
    },
    "@id": "dbpedia:Mississippi_River",
    "describedBy": "wiki:Mississippi_River",
    "name": "Mississippi river"
}
```

JSON-LD can be automatically converted to RDF triples. In the conversion, done using the JSON-LD playground, the second triple expresses that the *dbpedia* id (the context IRI) is *describedBy* (the link relation type) a *wikipedia* URL (the target IRI).

```
<http://dbpedia.org/page/Mississippi_River> <http://schema.org/name> "Mississippi
river" .
<http://dbpedia.org/page/Mississippi_River>
<http://www.iana.org/assignments/relation/describedby>
<http://en.wikipedia.org/wiki/Mississippi_River> .
```

9.1.3. Hypertext Application Language

Hypertext Application Language (HAL) is a simple format that tries to give a consistent and easy way to express hyperlinks between resources. It was defined by Mike Kelly, that is a software engineer from the UK that runs an API consultancy helping companies design and build APIs.

HAL notation for JSON links

```
{
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "name": "Mississippi river",
    "_links": {
        "describedBy": { "href": "http://en.wikipedia.org/wiki/Mississippi_River" }
    }
}
```

9.1.4. Atom link direct JSON encoding

The book "RESTful Web Services Cookbook; Solutions for Improving Scalability and Simplicity" By Subbu Allamaraju, in its chapter "How to Use Links in JSON Representations" proposes a direct translation of the Atom xlink notation in two forms:

Atom translation to JSON. Alternative 1.

```
{
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "name": "Mississippi river",
    "links": [{
        "rel": "describedBy",
        "href": "http://en.wikipedia.org/wiki/Mississippi_River"
    }]
}
```

This approach is consistent with what is proposed and generalized for applying it in JSON Schema: JSON Hyper-Schema: A Vocabulary for Hypermedia Annotation of JSON.

There is also a more compact format alternative.

Atom translation to JSON. Alternative 2.

```
{
    "@id": "http://dbpedia.org/page/Mississippi_River",
    "name": "Mississippi river",
    "links": [{
        "describedBy": { "href": "http://en.wikipedia.org/wiki/Mississippi_River" }
    }
    ]
}
```

The later alternative has the advantage that checking for a the presence of a "describedBy" linking is easier in JavaScript and at the same time looks almost identical to the HAL proposal.

Accessing a link in the alternative 2.

```
river=JSON.parse("...");
river.links.describedBy[0]
```

To do the same with the first alternative a JavaScript loop checking all links until finding one of the *describedBy* type will be needed.

If we remove the *grouping* property "links", then we almost converge to the JSON-LD alternative.

9.1.5. Recommendation

Even if it is difficult to formulate a recommendation, the authors of this guide consider that the

JSON-LD alternative has the advantage of simplicity and, at the same time, is the only alternative ratified and approved by an standard body. It has also the advantage to connect with the RDF world.

Chapter 10. Rules for encoding JSON Schema and JSON-LD context documents from UML class diagrams

The rules provided in this clause and in the following one Rules for encoding JSON-LD instances conformant to the UML model have been created in a very formal way using normative language (i.e. using SHALL as a indication of obligation). Even if this style could be considered not appropriate for an ER, it has been done with the objective of facilitating the migration of the rules into normative document or an standard in the near future. In this sense, they are numbered and contain only ONE normative paragraph after the rule number. The normative text has been elaborated with the intention of being self-sufficient to implement an *encoding service*. However, the consequences or the reasoning behind a rule are sometimes difficult to extract from the rule itself. In these cases, after the normative text, informative (non normative) justifications, clarifications and examples can follow.

Rules have been classified in subsections for clarification purposes only. All rules form a single corpus and need to be applied together.

Also note that we target *JSON Schemas* and have @context documents to guide the JSON-LD to RDF conversions and give semantics of objects and attribute names. This JSON Schemas and @context documents are generic and will be applied to several JSON instances. Actually, next clause Rules for encoding JSON-LD instances conformant to the UML model targets the JSON instances.

In practice, these rules inform UML application schema conversion implementors. In other words, it can be applied in a future automatic UML application schema conversion tool (e.g. ShapeChange).

10.1. Root element

Rule 1.1: The JSON schema SHALL declare the first type as "object"

Example of the start of a JSON Schema file that defines the root property as an object.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Coverage objects",
    "description": "Schema for Coverage objects",
    "type": "object"
}
```

Even if a JSON-LD could also be of the array type (or a simple type) in encoding a root class coming from UML, this UML class is represented by a single the root element.

10.2. Namespaces

Rule 2.1: The *root @context document* SHALL contain a list of properties with the names of the abbreviated namespaces and value of the full namespace URI for the properties of the root element

of the JSON document.

Example of a fragment of the root "@context" document (coverage-context.json) containing definition of the abbreviated and full namespace.

```
{
    "@context":
    {
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "cis": "http://www.opengis.net/cis/1.1/",
        [...]
    }
}
```

Rule 2.2: The *root "@context" document* SHALL define a property "id" as "@id" and a property "type" as "@type".

Example of a fragment of the root "@context" document where id and type are defined.

```
{
    "@context":
    {
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "cis": "http://www.opengis.net/cis/1.1/",

        "id": "@id",
        "type": "@type"
        [...]
    }
}
```

The properties that starts with "@" are a bit more difficult to access in JavaScript that the others because they can NOT be called using "coverage.@id" but using the array notation that is less *natural* "coverage[@id]". The existance of this rule allows for using "context.id" and "context.type" instead for "@id" and "@type" in JavaScript.

Rule 2.3: For all abbreviated namespaces, the correspondence to full namespace URI SHALL be listed in a @context document. If the document has more than one "@context", and the abbreviated namespace will be used only inside a particular JSON object that has a "@context" property, the definition SHALL be done in this "@context" (instead of using the root "@context").

10.3. Objects

Rule 3.1: In the JSON schema, each JSON object (including the root) SHALL define "id" property with *type* "string" and *format* "uri". This property SHALL be defined as mandatory (*required*).

Example of the addition of the necessary id's

```
{
    [...]
    "type": "object",
    "required": [ "id", ... ],
    "properties": {
        "id": { "type": "string", "format": "uri"},
        [...]
    },
}
```

Rule 3.2: In a JSON schema, each JSON object (including the root) SHALL define a "type" property with *type enum*. The *enum* will enumerated the UML class type name. The value of the *type* property of each JSON object in a JSON instance SHALL not contain an abbreviated namespace prefix. UML class type names are usually in UpperCammelCase and the values of the *enum* will be too.

Example of the addition of the necessary type's

```
{
    "seviceIdentification": {
        "title": "Sevice identification",
        "type": "object",
        "required": [ "id", "type", ... ],
        "properties": {
            "type": { "enum": ["SeviceIdentification"] },

}
```

Rule 3.3: The possible values of the *type* property SHALL be listed as a property name of a @context document with a value of the abbreviated namespace, a ":" character and the name again.

Example of the definition of types in JSON-LD.

```
{
    "@context":
    {
        "md": "http://www.opengis.net/md",
        "examples": "http://www.opengis.net/cis/1.1/examples/"

        "id": "@id",
        "type": "@type",
        "MD_LegalConstraints": "md:MD_LegalConstraints"
    },
    "type": "MD_LegalConstraints",
    "id": "examples:CIS_05_2D",
    ...
}
```

Using the types values without namespace in the JSON object definitions allows for creating a JSON schema that is able to correctly enumerate the possible values of a type property without the abbreviated namespace. This way the abbreviated namespace may vary from one instance without affecting the common JSON schema. In contrast, "id" values are expected to be different in each instance (and their values are not verified by the JSON schema) so they should contain the abbreviated namespace.

If an object can present more that one type (e.g. there is a one or more generalized classes of a more generic class) the all the alternative types are listed in the appropriate @context section.

Example of more that one type definition for a class.

```
{
    "@context":
    {
        "MD_Constraints": "md:MD_Constraints",
        "MD_LegalConstraints": "md:MD_LegalConstraints",
        "MD_SecurityConstraints": "md:MD_SecurityConstraints"
    }
}
```

Again, there is no need to list the Abstract types due to they cannot be instanciated in JSON-LD instances.

In a JSON-LD instance, when UML generalization is used to derive more specific classes from a generic class, the value of the property "type" will be the specialized class type name, instead of the generic class name. This behavior will favor both old and the new implementations. Old implementations will be able to ignore the "type" value, identify the object name and read the common properties from the generic class. New implementation will recognize the "type" value and be prepared to find the specialized properties.

Rule 3.4: A UML *Union* SHALL be expressed as a JSON object defined as an *anyOf* array. Each anyOf item SHALL be defined as one property (plus the *type* and *id* properties) that are defined as required. All *type* properties SHALL be identically defined.

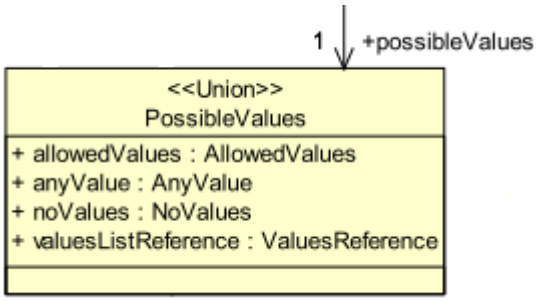

Figure 5. UML model for a generalized class ServiceIdentification.

JSON Schema fragment example of a UML Union encoded in JSON.

```
"possibleValues": {
    "title": "Possible Values list",
    "type": "object",
    "anyOf": [
        {
            "required": [ "type", "allowedValues"],
            "properties": {
                "type": { "enum": ["PossibleValues"] },
                "allowedValues": { "$ref": "#/definitions/AllowedValues" }
            }
        },{
            "required": [ "type", "anyValue"],
            "properties": {
                "type": { "enum": ["PossibleValues"] },
                "anyValue": { "$ref": "#/definitions/AnyValue" }
            }
        },{
            "required": [ "type", "noValues"],
            "properties": {
                "type": { "enum": ["PossibleValues"] },
                "noValues": { "$ref": "#/definitions/NoValues" }
            }
        },{
            "required": [ "type", "valuesListReference"],
            "properties": {
                "type": { "enum": ["PossibleValues"] },
                "valuesListReference": { "$ref": "#/definitions/ValuesReference" }
            }
        }
    ]
}
```

Rule 3.5: A UML class that is a *aggregation* or a *composition* of a parent class, SHALL be considered equivalent to a JSON complex property of the parent object (considering it as the same as a UML complex attribute). The name of the source extreme of the relation (the tip of the arrow) will be considered the name of the complex *attribute* and the *class name* will be considered the *type* of the property.

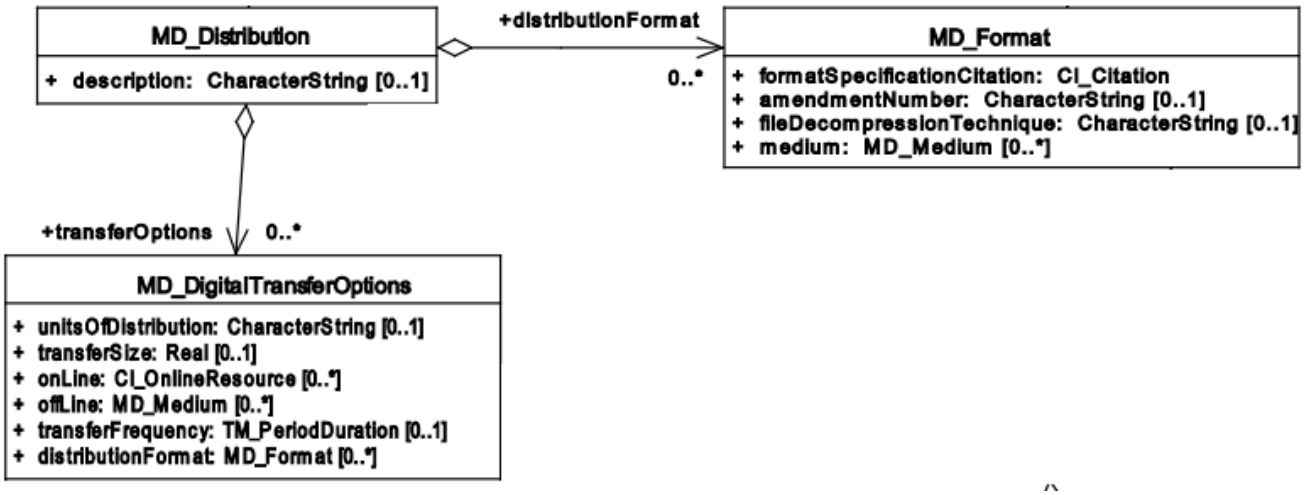

Figure 6. UML model showing a class that is defined as an aggregation of MD_Distribution and also as a complex data type MD_DigitalTransferOptinos

JSON Schema fragment example of a class used as an aggregation and as a data type (the example ignores the tranferOption multiplicity for simplicity)

```
{
    ...
    "distributionInfo": {
        "description": {"type": "string"},
        "distributionFormat": {"$ref": "#/definitions/MD_format"},
        "transferOptions": {
            ...
            "distributionFormat": {"$ref": "#/definitions/MD_format"}
        }
    }
}
```

Please consider the rules for complex attributes and properties below.

10.4. Object attributes; names

Rule 4.1: All UML class attributes SHALL be listed in a @context object. If the document has more than one "@context", the property SHALL be defined in the "@context" that is closer to it (with the exception of "id" and "type" that are defined in the root "@context" document and always have global scope). The value of the listed properties has to be the abbreviated namespace, the ":" character and the name of the property again.

Defining the properties in the "@context" that is closer to it makes the definition local to the object where the "@context" belongs.

Example of the addition of the abbreviate namespace to properties

```
{
    "domainSet":{
        "@context":
        {
            "generalGrid": "cis:generalGrid",
            "axis": "cis:axis",
            "axisLabel": "cis:axisLabel",
            "lowerBound": "cis:lowerBound",
            "upperBound": "cis:upperBound"
        },
        "generalGrid":{
            "axis": [{
                "type": "IndexAxisType",
                "axisLabel": "i",
                "lowerBound": 0,
                "upperBound": 2
            },{
                "type": "IndexAxisType",
                "axisLabel": "j",
                "lowerBound": 0,
                "upperBound": 2
            }]
        }
    }
}
```

Rule 4.2: All UML class attributes (as well as composition and aggregations) SHALL be listed as *properties* of the object in the JSON Schema.

Example of axisLabel, lowerBound and upperBound properties definition.

```
{
    "type": "object",
    "properties": {
        "type": { "enum": [ "IndexAxisType"] },
        "axisLabel":  { "type": "string" },
        "lowerBound":  { "type": "number" },
        "upperBound": { "type": "number" }
    },
```

10.5. Object attributes; multiplicity

Rule 5.1: Attributes, aggregations and compositions of an object with multiplicity 0 or 1 in the UML will be listed as JSON properties in the JSON schema.

Example of axisLabel, lowerBound and upperBound properties definition.

```
    "axis": {
        "type": "object",
        "properties": {
            "type": { "enum": [ "IndexAxisType"] },
            "axisLabel":  { "type": "string" },
            "lowerBound":  { "type": "number" },
            "upperBound": { "type": "number" }
        }
    },
```

Rule 5.2: Attributes, aggregations and compositions of an object with multiplicity more than 0 in the UML will have their names listed in the array of "required" properties in the JSON schema

Example of the "required" list of property.

```
    "axis": {
        "type": "object",
        "required": [ "type", "axisLabel", "lowerBound", "upperBound" ],
        "properties": {
            "type": { "enum": [ "IndexAxisType"] },
            "axisLabel":  { "type": "string" },
            "lowerBound":  { "type": "number" },
            "upperBound": { "type": "number" }
        }
    },
```

Rule 5.3: Attributes, aggregations and compositions with multiplicity more that 1 in the UML will be encoded as JSON properties of the *type* "array" in the JSON schema

In JSON instances the arrays of JSON objects will defined the type for each member of the array. The "type" can be different from the other members of the array (JavaScript allows arrays that are heterogenous in types) but all "type" values will be an generalization of the same generic UML class.

Example of JSON schema for numeric properties with multiplicity more than 1.

```
{
    "coordinates": {
        "type": "array",
        "items": { "type": "number" }
    },
}
```

Rule 5.4: A property defined in the UML as *ordered* and with multiplicity more than 1, SHALL be defined as "@container": "@list" in a @context document.

There is an important singularity in JSON-LD about this. When a JSON document is converted into JavaScript all JSON arrays becomes automatically with order. However in JSON-LD the situation is

the opposite, and by default arrays are NOT considered as ordered when converting JSON-LD arrays to RDF. There is technical reason behind this: ordered arrays require much more RDF code and the conversion does *not* result in a *nice* RDF code. For that reason if the a property is maked as *ordered* in the UML, we have to explicitly indicate this in a @context. Please limit the use this parameter only when order is really important.

10.6. Object attributes; data types

Rule 6.1: Numeric attributes in the UML SHALL have "type": "number" in the JSON Schema.

Example of JSON schema for anyURL properties with multiplicity 0 or 1.

```
{
    "lowerBound":  { "type": "number" },
    "upperBound": { "type": "number" }
}
```

In JSON there is no able to distinction between diferent numeric data types: E.g. Integer, Float, Double etc and they all became *number*.

Numeric attributes with multiplicity more than 1 will have "type": "array" and "items": {"type": "number"} in the JSON Schema.

Example of JSON schema for numeric properties with multiplicity more than 1.

```
{
    "coordinates": {
        "type": "array",
        "items": { "type": "number" }
    },
}
```

Rule 6.2: Boolean attributes SHALL have "type": "boolean" in the JSON Schema.

Boolean attributes with multiplicity more than 1 will have "type": "array" and "items": {"type": "boolean"} in the JSON Schema.

Rule 6.3: String attributes SHALL have "type": "string" in the JSON Schema.

Example of JSON schema for anyURL properties with multiplicity 0 or 1.

```
{
    "axisLabel":  { "type": "string" }
}
```

If there is a reason to believe that the attribute has been defined as UML string to allow both numbers or strings (depending of the case), define the type as an array of "number" and "string" is recommended.

Example of JSON schema for an string that can be also instanciated as a number.

```
{
    "value":  { "type": ["number", "string"] },
}
```

String attributes with multiplicity more than 1 will have "type": "array" and "items": {"type": "string"} in the JSON Schema.

Example of JSON schema for string properties with multiplicity more than 1.

```
{
    "axisLabels":
    {
        "type": "array",
        "items": { "type": "string" }
    }
}
```

Rule 6.4: anyURL attributes with multiplicity 0 or 1 SHALL have "type": "string" and "format": "uri" in the JSON Schema.

Example of JSON schema for anyURL properties with multiplicity 0 or 1.

```
{
    "srsName": { "type": "string", "format": "uri"}
}
```

anyURL attributes with multiplicity more that 1 will have "type": "array" and "items": {"type": "string", "format": "uri"} in the JSON Schema.

Example of JSON schema for anyURL properties with multiplicity more than 1.

```
{
    "srsNames":
    {
        "type": "array",
        "items": { "type": "string", "format": "uri"}
    }
}
```

Rule 6.5: A property that has the value anyURI SHALL be described as "@type": "@id" in a JSON @context document

Example of the definition of a value anyURI in JSON-LD

```
{
    "@context":
    {
        "srsName": {"@id":"swe:srsName", "@type": "@id"}
    }
}
```

The reasoning behind this is that property values defined without "@type": "@id" are considered literals when converted to RDF. Property values defined with "@type": "@id" considered as URIs when converted to RDF.

Rule 6.6: Complex type attributes SHALL have "type": "object" in the JSON Schema.

Example of JSON schema for complex properties with multiplicity 0 or 1.

```
{
    "generalGrid":{
        "title": "General Grid",
        "description": "General Grid",
        "type": "object"
    }
}
```

Complex type attributes with multiplicity more than 1 will have "type": "array" and "items": {"type": "object"} in the JSON Schema

Example of JSON schema for complex properties with multiplicity more than 1.

```
{
    "axis": {
        "type": "array",
        "items": {
            "type": "object"
        }
    }
}
```

Rule 6.7: A UML complex type attribute SHALL be encoded as JSON Object properties or as an Array of JSON Object type properties in a JSON-LD instance.

10.7. Object attributes; null values

Rule 7.1: For a attribute that can have null values, an array of types combining the variable data type and "null" SHALL be used.

Example of a nullable value.

```
{
    "lowerBound":  { "type": ["number", "null"] },
}
```

10.8. Object attributes; enumerations and code-lists

Rule 8.1: A UML "enumeration" SHALL be encoded as an "enum" in a JSON schema. Enumerations SHALL be listed in the "definitions" section of the JSON schema to be able to reuse them as needed.

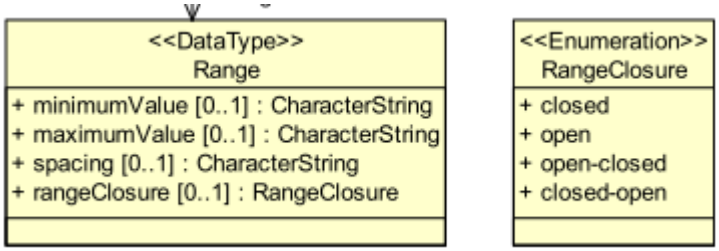

Figure 7. UML model for a enumeration.

Example of enumerations

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "rangeClosure": {"$ref": "#/definitions/RangeClosure"},
        ...
    }
    "definitions": {
        "RangeClosure": {
            "title": "Values of RangeClosure enumeration",
            "enum": ["closed", "open", "open-closed", "closed-open"]
        }
    }
}
```

Rule 8.2: A UML "codelist" SHALL be encoded as an oneOf "enum" or "string" in a JSON schema. Codelists SHALL be listed in the *definitions* section of the JSON schema to be able to reuse them as needed.

Example of the addition of the necessary id's

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "required": [ "codelist_a"],
    "properties": {
        "codelist_a": { "$ref": "#/definitions/codelist1" },
        ...
    },
    "definitions": {
        "codelist": {
            "oneOf": [
                {
                    "enum": [ "a2", "b2", "c2" ]
                },{
                    "type": "string"
                }
            ]
        }
    }
}
```

The reason for this is that codelist are considered extendable and in practice they should support any value. See a good discussion on how to encode enumeration an codelist in JSON Schema here: http://grokbase.com/t/gg/json-schema/14b79eqgqq/code-list-enum-extension

10.9. Objects: Data types and inheritance

Rule 9.1: If a UML class is defined as *DataType* (and potentially used in more that one place in the UML model) it SHALL be defined in the *definitions* section of the JSON schema and referenced by each attribute that is declared of this *DataType*

Actually, it is highly recommended that all UML classes are defined in the *definitions* sections. Only the objects defined in the *definitions* and the root object can be referenced from another JSON schema.

JSON Schema fragment example of the use of definitions section.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "seviceIdentification": { "$ref": "#/definitions/SeviceIdentification" },
        ...
    },
    "definitions": {
        "ServiceIdentification": {
            "title": "Service identification",
            "type": "object",
            "required": [ "type", "serviceType", "serviceTypeVersion" ],
            "properties": {
                "type": { "enum": ["ServiceIdentification"] },
                "serviceType": { "$ref": "#/definitions/Code"},
                "serviceTypeVersion":  {
                    "type": "array",
                    "items": {"type": "string" }
                },
                "profile":  {
                    "type": "array",
                    "items": {"type": "string"}
                },
                "fees": {"type": "string"},
                "accessConstraints": {"type": "string"}
            }
        }
    }
}
```

Example of using "definitions" section to define an object that can be used in more that one place emulating the UML data type behaviour.

```
{
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "profile" : { "$ref": "#/definitions/links" },
        "links" : {
            "type": "array",
            "items": { "$ref": "#/definitions/links" }
        }
    }
    "definitions": {
        "links": {
            "title": "links",
            "description": "Properties that all types of links have. It mimics the
Atom link",
            "required": [ "href" ],
            "properties": {
            "href": {"type": "string", "format": "uri" },
            "type" : { "type": "string" },
            "title" : { "type": "string" },
            "lang" : { "type": "string" }
            }
        }
    }
```

Rule 9.2: If an class is generalized into other classes in the UML, the JSON schema SHALL define the main class properties in the definitions section with the UML class name and the word "Properties" (not including the "type"). If the main class is not abstract, it SHALL then be defined with the UML call name by combining the previous "properties" and the "type" property with *allOf*. The generalized class will do the same and combine the main properties, the "type" and its own properties with *allOf*

If the main class is empty (it has no attributes) this rule does not apply.

Unnecessary duplication of the definition do the common elements coming from the abstract class is avoided by using *$ref* and pointing to a *definitions* element in the JSON Schema. There is no mechanism to inhered properties from a previous object in JSON schema but the suggested mechanism achieves an equivalent results.

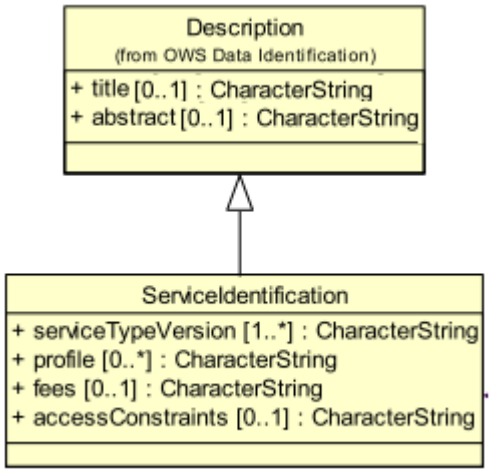

Figure 8. UML model for a generalized class ServiceIdentification.

JSON Schema fragment example of generalization of ServiceIdentification inheriting a group of properties DescriptionProperties. Note that "Description" is defined for completeness but it is not used.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "serviceIdentification": { "$ref": "#/definitions/ServiceIdentification" }
    },
    "definitions": {
        "ServiceIdentification": {
            "required": ["type"],
            "allOf": [
                { "$ref": "#/definitions/DescriptionProperties" },
                {
                    "properties":{
                        "type": {"enum": ["ServiceIdentification"]},
                        "serviceTypeVersion": {"type": "string"},
                        "profile": {"type": "string"},
                        "fees": {"type": "string"},
                        "accessConstraints": {"type": "string"}
                    }
                }
            ]
        },
        "Description": {
            "required": ["type"],
            "allOf": [
                { "$ref": "#/definitions/DescriptionProperties" },
                {
                    "properties":{
                        "type": {"enum": ["Description"]}
                    }
                }
            ]
        },
        "DescriptionProperties": {
            "properties": {
                "id": {"type": "string", "format": "uri"},
                "title": {"type": "string"},
                "abstract": {"type": "string"}
            }
        }
    }
}
```

JSON fragment that validates with he previous example

```
{
    "serviceIdentification": {
        "type": "ServiceIdentification",
        "title": "My WMS server",
        "abstract": "This WMS server is mine",
        "serviceTypeVersion": "1.1.1",
        "profile": "http://www.opengis.net/profiles/nga"
    }
}
```

Rule 9.3: If a class is generalized into more than one classes in the UML (which forces the instance to choose one among of them), the JSON Schema SHALL define an object that offers the different options using the *oneOf* property.

When the main class is *abstract*, there is no need to define it as a data type due to it cannot be instanciated in JSON-LD instances.

In the following example, the class MD_Constraints is generalized into 2 different classes: MD_LegalConstraints and MD_SecurityConstraints. The pattern used in the example is a bit diffent for the one in the example of he Rule 3.4. Testing the us of *oneOf* to select between object definitions that contains *allOf* including the same $ref fails to work. To solve this, we define *MD_LegalConstraintsAdditions* and *MD_SecurityConstraintsAdditions* that only defines the non common properties. Then A new object *Alternatives_MD_Constraints* offers all the alternatives available with *oneOf* and add the common properties *MD_ConstraintsProperties* at the end with *allOf*.

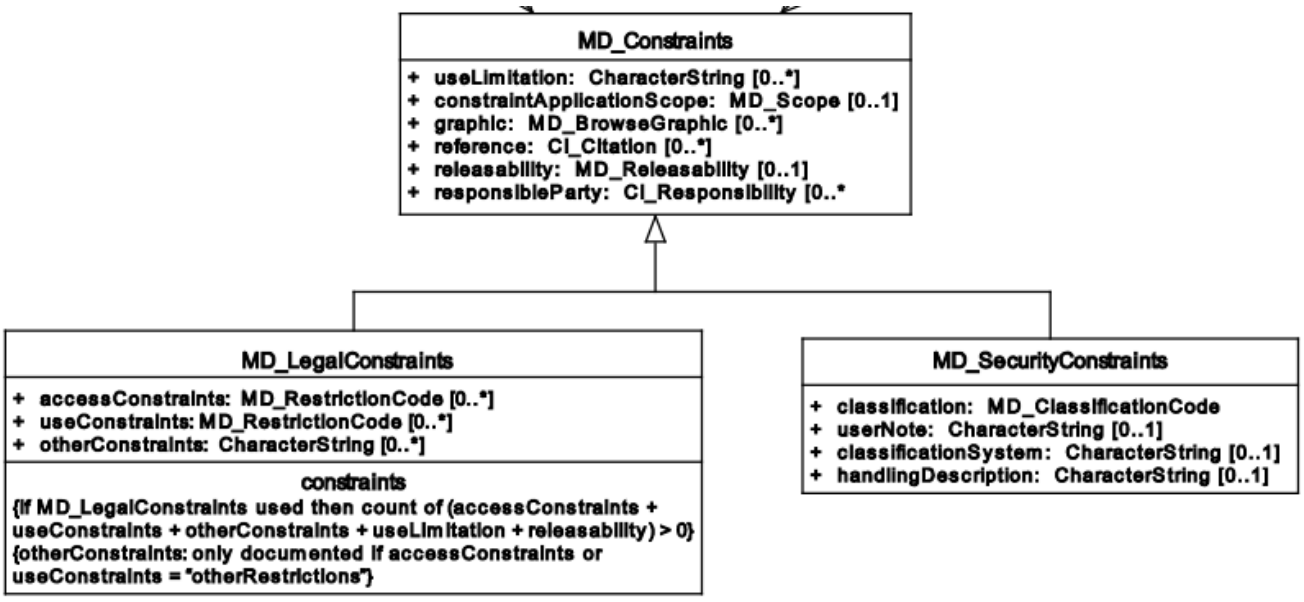

Figure 9. UML model for a generalized class from MD_Constraints.

Example of multiple generalization

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
```

```
    "properties":
    {
        "resourceConstraints": {
            "$ref": "#/definitions/Alterantives_MD_Constraints"
        }
    },
    "definitions": {
        "Alterantives_MD_Constraints": {
            "type": "object",
            "allOf": [
                { "$ref": "#/definitions/MD_ConstraintsProperties" },
                { "oneOf": [
                        {"$ref": "#/definitions/MD_SecurityConstraintsAdditions"},
                        {"$ref": "#/definitions/MD_LegalConstraintsAdditions"},
                        {"$ref": "#/definitions/MD_ConstraintsAdditions"}
                    ]
                }
            ]
        },
        "MD_LegalConstraintsAdditions": {
            "required": ["type"],
            "properties":{
                "type": {"enum": ["MD_LegalConstraints"]},
                "accessConstraints": {"type": "object"},
                "otherConstraints": {"type": "string"}
            }
        },
        "MD_SecurityConstraintsAdditions": {
            "required": ["type"],
            "properties":{
                "type": {"enum": ["MD_SecurityConstraints"]},
                "useNote": {"type": "string"},
                "classificationSystem": {"type": "string"},
                "handlingDescription": {"type": "string"}
            }
        },
        "MD_ConstraintsAdditions": {
            "required": ["type"],
            "properties":{
                "type": {"enum": ["MD_Constraints"]}
            }
        },
        "MD_ConstraintsProperties": {
            "properties": {
                "id": {"type": "string", "format": "uri"},
                "useLimitation": {"type": "string"},
                "constraintApplicationScope": {"type": "object"}
            }
        }
    }
}
```

10.10. Object libraries and multiple schemas

UML classes can be structured in a modular way in packages. In this case there is a need to use more than one schema file. The core schema offers a set of *datatypes* in the *definitions* section. This core schemas can be reused by other schemas pointing the definition of the right objects in the core schemas using a full path to them.

Rule 10.1: Each UML class packages SHALL be described in the *definitions* in a JSON schema. Schemas that reuse other UML classes in other packages SHALL point to them using a full path.

Example of a core JSON schema (called ServiceMetadata_schema.json) for a common class in OWS common package

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "ServiceIdentification": {
            "title": "Service identification",
            "description": "Metadata about this specific server. The contents and
organization of this section should be the same for all OWSs. ",
            "type": "object",
            "properties": {
                "type": { "enum": ["ServiceIdentification"] },
                "serviceType": { "$ref": "#/definitions/Code"},
                "serviceTypeVersion":  {
                    "type": "array",
                    "items": {"type": "string" }
                },
                "profile":  {
                    "type": "array",
                    "items": {"type": "string"}
                },
                "fees": {"type": "string"},
                "accessConstraints": {"type": "string"}
            }
        }
    }
}
```

Example of a another JSON schema (called WMSServiceMetadata_schema.json) for a common class in OWS common package

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "WMS Service Metadata root object",
    "required": [ "type", "version"],
    "properties": {
        "type": { "enum": [ "WMSServiceMetadata"] },
        "id":  { "type": "string" },
        "version": { "type": "string"},
        "serviceIdentification": { "$ref":
"ServiceMetadata_schema.json/#/definitions/ServiceIdentification" },
    }
}
```

Example of a JSON instance (called WMSServiceMetadata.json) validated with the WMSServiceMetadata_schema.json file.

```
{
    "type": "WMSServiceMetadata",
    "version": "1.4",
    "serviceIdentification": {
        "type": "ServiceIdentification",
        "serviceType": {
            "type": "Code",
            "code": "WMS"
        },
        "serviceTypeVersion": ["1.4"],
        ...
    }
}
```

10.11. Other considerations

10.11.1. Large arrays

A property containing a large array of values (or coordinates), where the order is important, should not be defined in @context objects if an alternative representation of the same information can be provided by other means. Not defining the long ordered arrays properties in @context avoids them to be converted into RDF. Ordered arrays an nasty and very verbose, double ordered arrays cannot even be converted. *Alternative representations* can be "string literals" (encoding the arrays in a text that is more compact; e.g. coordinates encoded as WKT) or links to URLs in a non RDF format containing this information (such as NetCDF or GeoTIFF files). This is further discussed in the section [Clause_Data_Formats_Encoding] with specific focus on coordinate arrays but not only.

Chapter 11. Rules for encoding JSON-LD instances conformant to the UML model

The rules provided in this section and in the previous one Rules for encoding JSON Schema and JSON-LD context documents from UML class diagrams have been created in a very formal way using normative language (i.e. using SHALL as a indication of obligation). Even if this style could be considered not appropriate for an ER, it has been done with the objective of facilitating the migration of the rules into normative document or an standard in the near future. In this sense, they are numbered and contain only ONE normative paragraph after the rule number. The normative text has been elaborated with the intention of being self-sufficient to implement an *encoding service*. However, the consequences or the reasoning behind a rule are sometimes difficult to extract from the rule itself. In these cases, after the normative text, informative (non normative) justifications, clarifications and examples can follow.

Rules have been classified in subsections for clarification purposes only. All rules form a single corpus and need to be applied together.

Also note that we target *JSON-LD instances* that are validated using JSON Schema and have @context documents to guide the JSON-LD to RDF conversions and give semantics of objects and attribute names. Actually, the previous clause Rules for encoding JSON Schema and JSON-LD context documents from UML class diagrams targets the JSON schemas and @context documents

In practice, these rules inform JSON instance writers

11.1. General

Rule 1.1: A JSON-LD instance SHALL follow the JSON syntax specified in the IETF RFC7159 The JavaScript Object Notation (JSON) Data Interchange Format.

This *RFC7159 conformance rule* actually implies several other rules stated in RFC7159 and that will not be repeated here but need also considered (e.g. JSON Strings values will be encoded in UTF8 in a JSON-LD instance)

Rule 1.2: A JSON-LD instance SHALL validate with the JSON schema derived from the corresponding UML class diagram model.

This rule imposes automatically all the JSON schema specific rules for property values encoding (e.g. Numeric attributes will be encoded as JSON Numbers or Boolean attributes will have the *true* or *false* values or Arrays of them in a JSON-LD instance). Most of this rules will not be included here because they are imposed by the JSON schema validation process. Some others are included here are rules for clarity.

Several tools exist to validate JSON instances with a JSON schema automatically (such as XML Validator Buddy).

11.2. Root element

Rule 2.1: The root element in the JSON-LD instance shall have no name.

The JSON file is supposed to be loaded in a JavaScript object that will have a name identifying the object.

Example of code that load the JSON objects in the coverage variable

```
var coverage=JSON.parse(json_serialized_text);
```

Rule 2.2: The first property of the root element of a JSON-LD instance SHALL have the name "@context". The root "@context" property SHAL have a value consisting in an array formed two parts: a reference to a @context document (called *root @context document*) and a embedded context (called *root embedded context*).

Rule 2.3: A JSON-LD instance *root embeded context* SHALL contain at least a property with the name of the abbreviated namespace and value of the full namespace URI of the identifiers present on this JSON document.

Example of a fragment of the first part of the root object of a JSON document. If references the @context document in the last example and also embeds the definition of the id's namespace.

```
{
    "@context": [
        "http://localhost/json-ld/coverage-context.json",
        {
            "examples": "http://www.opengis.net/cis/1.1/examples/"
        }]
}
```

It is important to note that it can be other "@context" objects in the interior of any object (as the first property). This is particularly useful to define properties with local scope.

11.3. Objects

Rule 3.1: In a JSON-LD instance, each JSON object (including the root) SHALL contain a "id" property that uniquely identifies it (this is enforced by the JSON Schema validation). The value of this property shall be composed by a abbreviated namespace of the identifiers, a ":" character and an identifier name.

Example of the addition of the necessary id's

```
{
    "@context": [
        "http://localhost/json-ld/coverage-context.json",
        {
            "examples": "http://www.opengis.net/cis/1.1/examples/"
        }]  "id": "examples:CIS_05_2D",
    [...]
    "domainSet":{
        "id": "examples:CIS_DS_05_2D",
        "generalGrid":{
            "id": "examples:CIS_DS_GG_05_2D",
            [...]
        }
    }
}
```

Rule 3.2: In a JSON-LD instance, each JSON object (including the root) SHALL include a "type" property (this is enforced by the JSON Schema validation)

Parsers of the JSON encoding are not required to read and understand the "type" property but they can use it to redirect the code to the right parser of a code fragment or to identify a specialized type among all possible ones of a generic one (among the subclassified types).

Rule 3.3: The value of the *type* property of each JSON object in a JSON instance shall not contain an abbreviated namespace (actually, valid nomes are enforced by the JSON Schema validation).

Using the types values without namespace in the JSON object definitions allows for using a JSON schema that is able to correctly enumerate the possible values of a type property without the abbreviated namespace. This way tha abbreviated namespace may vary without affecting the common JSON schema. In contrast, "id" values are expected to be different in each instance (and their values are not verified by the JSON schema) so they should contain the abbreviated namespace.

Appendix A: Conformance Class Abstract Test Suite (Normative)

Appendix B: Annex B: JSON Schema Documents (informative)

In addition to this document, this standard includes several normative JSON Schema Documents.

Appendix C: Example JSON documents (informative)

Chapter 12. Introduction

This annex provides more example JSON documents than given in the body of this document. == $$$ example This is a complete example of JSON for…

Annex D: JSON in web services (informative)

The previous sections of this document have concentrated in JSON general principles and how them can be applied to data models and data encodings. This section focuses on how to use the same techniques in OGC web services. Traditionally, the OGC web services have used XML, the *lingua franca* for services. Currently, OGC web services are able to expose their capabilities and are able to send requests that cannot be easily encoded in KVP as XML documents that are posted to servers. Actually, in OWS Common, all versions up to 2.0 (the last currently available at the moment of writing this section), the only described encoding is XML. In principle, nothing prevents OGC web services to use other encodings that can be more convenient for modern clients and servers. This section describes how OGC web services can adopt JSON encoding.

HTML5 integrated web clients are the ideal platform to work with JSON encoded documents. They are able to automatically interpret a JSON serialized string and convert it to a JavaScript object only invoking JSON.parse(). The server side technologies were apparently not related with JSON and JavaScript, but this has changed with the introduction of Node.js. Node.js allows for the creation of web servers and networking tools using JavaScript and a collection of "modules" that handle various core functionality. Modules are provided for file system I/O, networking (DNS, HTTP, TCP, TLS/SSL, or UDP), binary data (buffers), cryptography functions, data streams, etc. In Node.js, developers can create highly scalable servers without using threading, by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task. Since the Node.js can be defined as JavaScript for services it can easily work with JSON. But this is not the only language supporting JSON; actually there are libraries to use JSON in many modern programming languages, including C. The authors of this document use cjson to include JSON support to CGI server applications developed in C code.

D.1. Sequence of steps to use JSON in services

This subsection describes the sequence of steps that client and server have to execute to use JSON in both a KVP HTTP GET and HTTP POST with JSON in the body of the request.

D.1.1. A KVP HTTP GET request

This subsection discusses how to use a KVP HTTP GET request to invoke a simple operation to an OGC service that can be transmitted in a KVP encoded URL. The example retrieves a WMS GetCapabilities document in JSON format.

NOTE	Currently, OWS Common 2.0 does not provide any official JSON encoding. In addition, there is no draft version of WMS 1.4 that describes the support to JSON yet. You can find a complete tentative encoding for OWS Common 2.0 and how to apply it to WMS 1.4 GetCapabilities in the "OGC 16-051 Testbed 12 JavaScript JSON JSON-LD Engineering Report". The following example uses materials produced and validated in OGC 16-051.

KVP GET client request

One of the things that surprises about JSON is that there is no *easy* way to *include* a JSON file to an

HTML page on-the-fly. To include a JSON file in a JavaScript based client you should use a method designed for asynchronous XML retrieval but can be used to transfer other file formats too.

The following code was obtained and adapted from Stack Overflow and has been incorporated in MiraMon WMS-WMTS client.

JavaScript function to get a JSON file and incorporate its content in the variables available to JavaScript

```
function loadJSON(path, success, error)
{
var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function()
    {
        if (xhr.readyState === XMLHttpRequest.DONE) {
            if (xhr.status === 200) {
                if (success)
                {
                    var data;
                    try {
                        data = JSON.parse(xhr.responseText);
                    } catch (e) {
                        if (error)
                            return error("JSON file: \""+ path + "\". " + e);
                    }
                    success(data);
                }
            } else {
                if (error)
                    error("JSON file: \""+ path + "\". " + xhr.statusText);
            }
        }
    };
    xhr.open("GET", path, true);
    xhr.send();
}
```

In the following example, a JavaScript client retrieves a GetCapabilities request in JSON format using loadJSON(). The first parameter is the URL of the GetCapabilities request. The second and third parameter are functions that will be executed in a "callback" style. Indeed, loadJSON() function will return as soon as the GetCapabilities request has been send without waiting for the communication to be completed. In other words, the client program flow continues without waiting for the server to respond. When the response is received correctly, the JSON text of the body of the response message will be converted to a JavaScript data structure in a process that is called *parse*. If the conversion is correct, the the success function will be then invoked and execute in another thread as described in the next subsection.

JavaScript call to the function to get a JSON file

```
loadJSON("http://www.opengis.uab.cat/miramon.cgi?request=GetCapabilities&service=WMS&a
cceptFormats=application/json",
        ShowCapabilities,
        function(xhr) { alert(xhr); });
```

Please, note that loadJSON() is sending a request to the following server application: http://www.opengis.uab.cat/miramon.cgi. The request contains a KVP encoding for a WMS GetCapabilities that includes a format key with the value application/json. The server expects a response in JSON.

KVP GET server response in JSON

While the client is performing other tasks, the server receives the request and extracts the information it needs to recognize the right task to perform from the KVP URL string. It recognizes the request for a GetCapabilities document in the JSON format. To generate it, it has two options: use a library to build a JSON object or to write a text string in memory. Generally, the server will opt for the second option since is easier to do. This approach has its risks, because the server will not validate that the string is actually well formed and contains a valid JSON. Sending the HTTP response back to the client with the 200 status, in a CGI server style is as simple as writing it in the stdout port preceded by the mandatory HTTP header that at least has to include to the MIME type. Nevertheless, it could be useful to structure the header at least with this three informations:

- Content-Type: indicates the MIME type of the response and it is needed by the web server (e.g. Internet Information Server, Apache Web Server, ...) to know how to handle the response.
- Content-Length: indicates the length of the body of the message in bytes (in our case the length of the JSON string). It is not mandatory but it is useful for the client to show the percentage of download to the user before the transmission has been complete.
- Access-Control-Allow-Origin: This header has been introduced in HTML5 to better handle the Cross Origin Resource Sharing (CORS). For security reasons, by default, the client will only accept content that comes from the same server than the original JavaScript code. This prevents to create an JavaScript integrated clients that access and visualize different servers and shows the content together (preventing any interoperability demonstrations). Now, the server can list in this entry the URLs of the servers it trust. If the server declares that trusts the JavaScript client, then the JSON file will be accessible for the client to read. In practice, it is difficult for the server to anticipate the clients we will trust, and, in practice, many clients declare that they trust anybody by using '*'.

Server response of a JSON file containing the description of the capabilities document (fragment)

```
Content-Type: application/json
Content-Length: 1456
Access-Control-Allow-Origin: *

{
    "type": "WMSServiceMetadata",
    "version": "1.4",
    "updateSequence": "a",
    "serviceIdentification": {
        "type": "ServiceIdentification",
        "serviceType": {
            "type": "Code",
            "code": "WMS"
        },
        "serviceTypeVersion": ["1.4"],
        "title": [{"type": "LanguageString", "value": "WMS service", "lang": "en-
en"}],
        "keywords": [{"type": "Keywords", "keyword": [{"type": "LanguageString",
"value": "service", "lang": "en-en"}]}]
    },
    "serviceProvider": {
        "type": "ServiceProvider",
        "providerName": "CREAF",
    [...]
}
```

When the response is received by the client, either the function in the second parameter or the function in the third parameter will be executed depending on the success or failure of the request.

In the following example we demonstrate how the *capabilities* variable already has the same structure as the JSON document received.

JavaScript callback function that will process a successfully received and parsed JSON file

```
function ShowCapabilities(capabilities)
{
    if (capabilities.version!="1.4" ||
        capabilities.serviceIdentification.serviceType.code!="WMS")
        alert("This is not a compatible WMS JSON server");
    alert("The provider name is: " +
        capabilities.serviceProvider.providerName);
}
```

D.1.2. KVP GET server exception in JSON

OWS Common defines the exception messages and HTTP status codes for a response to a request that cannot be processed by a server. The content of the message exception is also defined in XML

but it can be easily translated to an equivalent JSON encoding. In the following example, the server will return a HTTP status 400 (Bad request) and in the body will include a more precise description of the reason for not succeeding in providing a response (actually, there are two reasons in the example).

Example of an exception report encoded in JSON (equivalent to the one in section 8.5 of OWS Common 2.0)

```
{
    "type": "ExceptionReport",
    "version": "1.0.0",
    "lang": "en",
    "exception": [{
        "type": "Exception",
        "exceptionCode": "MissingParameterValue",
        "exceptionText": "Service parameter missing",
        "locator": "service"
    },{
        "type": "Exception",
        "exceptionCode": "InvalidParameterValue",
        "exceptionText": "Version number not supported",
        "locator": "version"
    }]
}
```

NOTE Modifications on the error handling part of the function loadJSON() could be required to better inform the user with the content of the exception report.

D.1.3. A JSON HTTP POST request

This subsection discusses how to use a HTTP POST request to invoke an operation to an OGC service. This is particularly useful when the content to be sent to the server is too long to embed it in a KVP URL. The example sends a WMS GetFeatureInfo request as a JSON file an expects also a JSON document as a response.

NOTE GetFeatureInfo is normally sent to the server as KVP URL. In this example we use the POST version for illustration purposes.

HTTP POST client request

The following code was obtained and adapted from Stack Overflow but have not been tested in the MiraMon WMS-WMTS client yet.

JavaScript callback function that will send a JSON document in a POST operation

```
function POSTandLoadJSON(path, body, success, error)
{
var xhr = new XMLHttpRequest();
var body_string;
    xhr.onreadystatechange = function()
    {
        if (xhr.readyState === XMLHttpRequest.DONE) {
            if (xhr.status === 200) {
                if (success)
        {
            var data;
            try {
                data = JSON.parse(xhr.responseText);
            } catch (e) {
                if (error)
                    return error("JSON file: \""+ path + "\". " + e);
            }
            success(data);
        }
            } else {
                if (error)
                    error("JSON file: \""+ path + "\". " + xhr.statusText);
            }
        }
    };
    xhr.open("POST", path, true);
    xhr.setRequestHeader("Content-type", "application/json");
    body_string=JSON.stringify(body);
    xhr.send(body_string);
}
```

The first thing that is needed is to create a JavaScript data structure that can be converted to a JSON string (a process called *stringify*). We are going exemplify this by proposing a data structure for a WMS GetFeatureInfo request.

NOTE The data structure in the example shows how a GetFeatureInfo could look like in JSON and POST. The proposed syntax is not based on any data model resulting from a standardization discussion but from a reasonable guess on how it could look like.

GetFeatureInfo request data structure in JSON

```
getFeatureInfoRequest={
    "StyledLayerList": [{
        "NamedLayer": {
            "Identifier": "Rivers"
        }
    }],
    "Output": {
        "Size": {
            "Width": 1024,
            "Height": 512
        },
        "Format": "image/jpeg",
        "Transparent": false
    },
    "BoundingBox": {
        "crs": "http://www.opengis.net/gml/srs/epsg.xml#4326",
        "LowerCorner": [-180.0, -90.0],
        "UpperCorner": [180.0, 90.0]
    },
    "QueryLayerList": [{
        "QueryLayer": {
            "Identifier": "Rivers"
        }
    }],
    "InfoFormat": "text/html",
    "PointInMap": {
        "I": 30,
        "J": 20
    },
    "Exceptions": "text/xml"
};
```

Having both the server URL and the JavaScript data structure we can now send the POST request to the server using the POSTandLoadJSON() function presented before.

GetFeatureInfo request data structure in JSON

```
POSTandLoadJSON("www.opengis.uab.cat/miramon.cgi",
        getFeatureInfoRequest,
        ShowGetFeatureInfo,
        function(xhr) { alert(xhr); });

function ShowGetFeatureInfo(getFeatureInfo)
{
    //Put here the code to show the data in the
    //same way as the ShowCapabilities does.
    //Normally you will interprete the getFeatureInfo
    //data structure and create a string that will be send to
    //a division with innerHTML
}
```

The server receives the JSON file and extracts the information it needs and continues with the sequence explained in KVP GET server response in JSON.

D.1.4. Cross Origin Resource Sharing security issue

The Cross Origin Resource Sharing (CORS) is a security issue that appears when a JavaScript code coming from a server requests information to another service that is in another domain. In this case, the default behavior is to deny access, except if the requested server (the server that is going to respond) specifically authorizes reading the data to the server that generated the code for the client that is making the request.

In KVP GET server response in JSON, we already have described the issue of CORS in HTTP GET requests and the need for the server that is responding with a JSON string to include "Access-Control-Allow-Origin" in the headers, allowing the origin server to merge data with the requested server. In practice the server is granting the client the right to access the JSON responded data.

In implementing POST requests and responses that require CORS, we have discovered that the situation is not so simple. The a HTMP5Rocks CORS tutorial (Handling a not-so-simple request) page describes the issue quite well.

To prevent the client to send unnecessary or sensible information to a server that will not grant access to the JSON data to the client, a "preflight" request is going to be formulated. This is invisible to the JavaScript client code but the server side (the OGC web server) needs to know it and needs to deal with it.

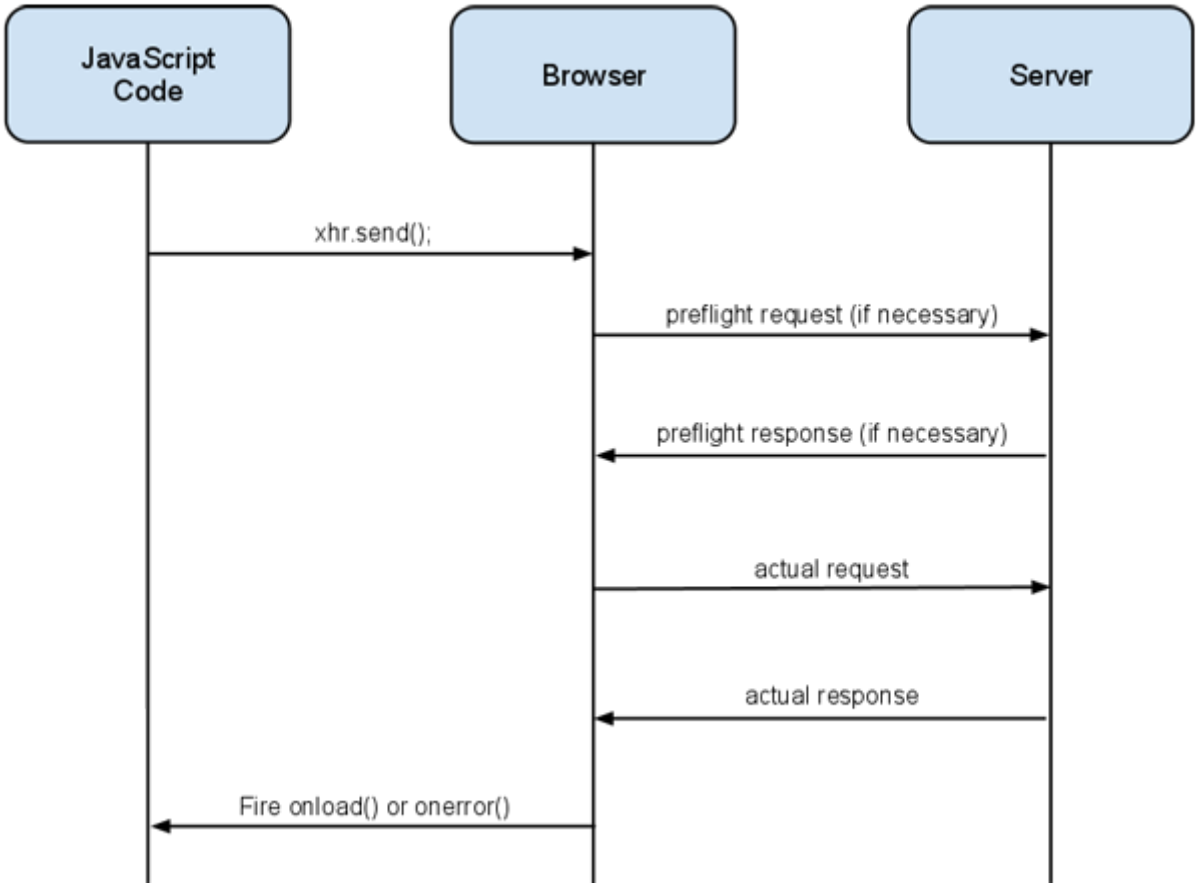

Figure 10. CORS flow in case of a POST request

The browser (not the JavaScript code) will issue a *preflight* request, that is normally an OPTIONS request. The server needs to be prepared for a request like this:

```
OPTIONS HTTP/1.1
Origin: http://client.bob.com
Access-Control-Request-Method: PUT
...
```

Then, the server need to respond a message that will contain only headers (no body) saying that it will support the requested method (and some others) to the requested server origin (and my be some others).

```
Access-Control-Allow-Origin: http://client.bob.com
Access-Control-Allow-Methods: GET, POST, PUT
...
```

Now that the web browser is convinced that the POST request will be accepted, it will issue it. Note that if the server does not respond correctly the OPTIONS request, the POST request will not be formulated and the POSTandLoadJSON() will receive and error and will trigger the error function.

Annex E: Revision History

Date	Release	Editor	Primary clauses modified	Description
2018-03-19	0.1	Joan Masó	all	initial version

Annex F: Bibliography

NOTE

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

- 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)

[n] Web: Author Surname, A.: Title, http://Website-Url

[1] OGC: OGC Testbed 12 Annex B: Architecture. (2015).