Testbed-12 JSON and GeoJSON User Guide

Table of Contents

Publication Date: 2017-06-21

Approval Date: 2017-06-19

Posted Date: 2016-12-29

Reference number of this document: OGC 16-122r1

Reference URL for this document: http://www.opengis.net/doc/PER/t12-A062

Category: User Guide

Editor: Joan Maso and Alaitz Zabala

Title: Testbed-12 JSON and GeoJSON User Guide

COPYRIGHT

IMPORTANT	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.
NOTE	This document is a user guide created as a deliverable in the OGC Innovation Program (formerly OGC Interoperability Program) as a user guide to the work of that initiative and is not an official position of the OGC membership. There may be additional valid approaches beyond what is described in this user guide.

POINTS OF CONTACT

Name	Organization

Joan Maso	UAB-CREAF
Alaitz Zabala	UAB-CREAF

Chapter 1. Introduction to JSON

This section presents JSON. Afterwords, it discusses about important questions such as what JSON offers, when a JSON encoding offers an advantage (e.g. in simplicity and flexibility) to an XML encoding, and when XML encodings can provide a better solution (e.g. adding more expressivity and robustness). Finally it presents some precisions that OGC can add on top of the basic JSON definition to ensure better interoperability of applications using JSON.

1.1. 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 does not have quotation marks, objects are enclosed in curly brackets "{}" and a 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 in 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 helps 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, 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 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.

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]
}
```

Check on KVP_GET_client_request for the way to incorporate a JSON document into a JavaScript code on the fly.

1.2. 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 had been proposed, including RSS, Atom, SOAP, and XHTML, Office Open XML, OpenOffice.org, Microsoft .NET Framework and many others. OGC has adopted XML for most of its web service messages and for several data formats, including GML, WaterML, SensorML, GUF, etc. XML has some interesting additional components: XML Schema/RelaxNG/Schematon 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), XSL transformations, 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 APIs of many web services offered by big companies on the web, 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 stand alone 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 on json.org).

1.2.1. The secret of JSON success in the web

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

- XML is not easy to learn. JSON format is so simple that 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 present in JSON.
- XML is defined 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 a error in 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 does 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 to adopt new additions to JSON, such as JSON validation and JSON-LD resulting in a no-so-simple JSON utilization.

1.3. 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.

1.3.1. 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.

1.3.2. 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"
}
```

1.3.3. 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"
}
```

The presence of this subtype is fundamental of JSON-LD that will be described in What_is_JSON_LD.

Chapter 2. JSON Schema

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 also be convenient.

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 also be used 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 losing 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 minimum 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 to 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".

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 also has 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
} ]
```

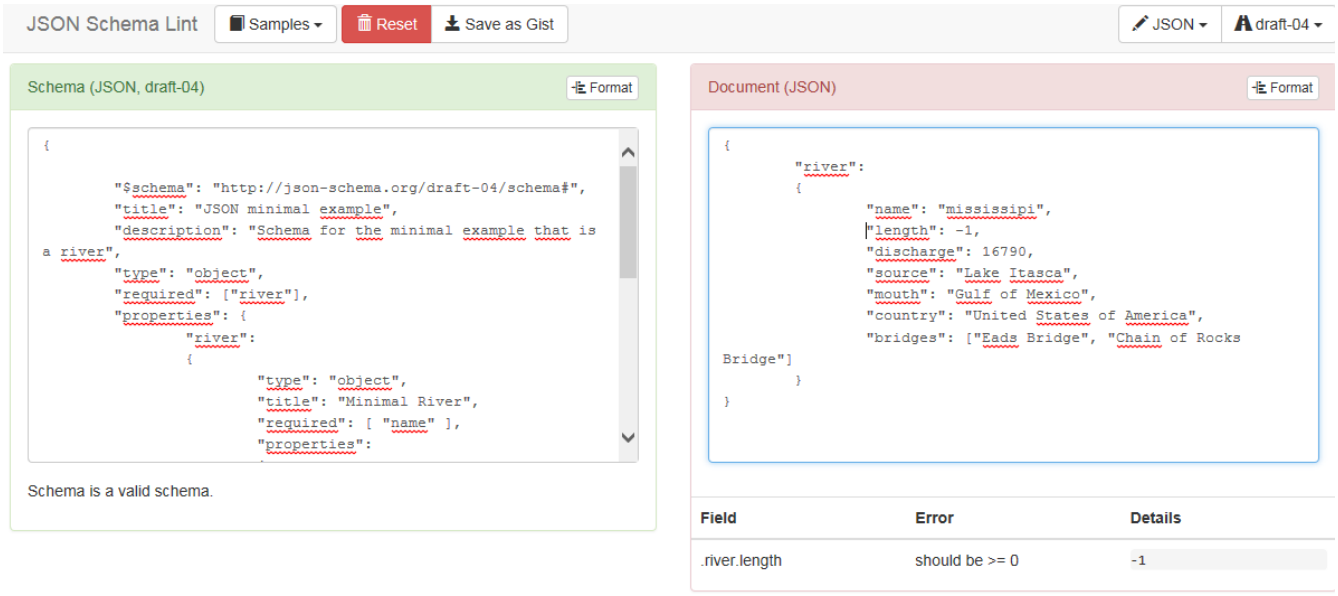

Figure 1. Response of the http://jsonschemalint.com/#/version/draft-04/markup/json

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" }
                    }
                }
            }
        }
    ]
}
```

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 collection 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 feature 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 no 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. Apart 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 Apply_JSONLD_to_JSON_objects_subsection).

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 3. Links in JSON

3.1. 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.

3.2. JSON-LD

In the next section we will explain JSON-LD in detail (please see Apply_JSONLD_to_JSON_objects_subsection) 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> .
```

3.3. 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, who 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" }
    }
}
```

3.4. 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.

3.5. 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 a standard body. It has also the advantage to connect with the RDF world.

Chapter 4. From JSON to JSON-LD

4.1. 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 URIs, a way to associate data types with values.

JSON-LD is considered another 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 exposed here.

4.2. 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 defined 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) does 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"
}
```

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

4.3. 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 add also 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 JSON-LD object using a preexisting vocabulary

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

Conversion to a river object with name to 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 preexisting and another embedded. This could be particularly useful if, in the future, OWS common decides to release a vocabulary for OGC and OGC concrete services need 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 river object with name and bridges to 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> .
```

4.4. 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"]
}
```

4.5. Defining data types for properties in JSON-LD

By default, JSON-LD considers properties as strings. JSON-LD also permits to define 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> .
[...]
```

4.6. 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 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 of a list of bridges where order is important 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> _: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 impossible to transform bi-dimensional arrays of coordinates. This issue is being discussed in Geospatial_dimension_in_JSON.

An 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_links_subsection.

```
{
    "@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 to 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> .
```

Chapter 5. 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.

5.1. 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*.

The 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.

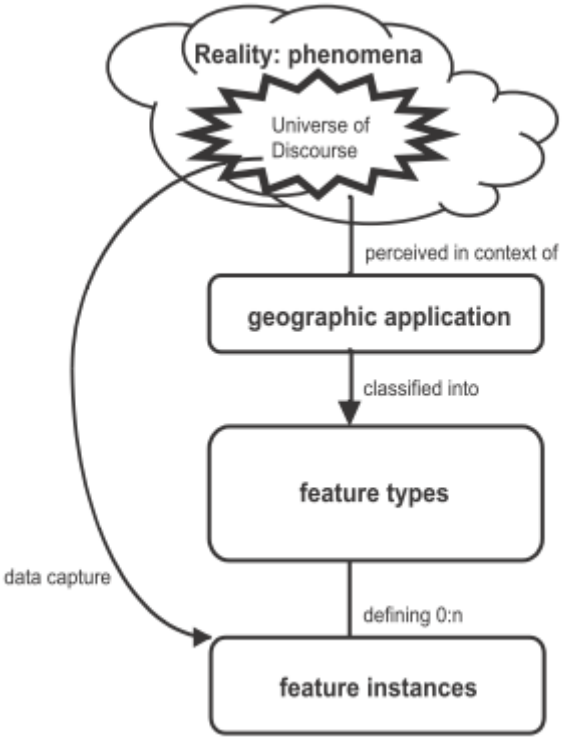

Figure 2. The process from universe of discourse to data

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.

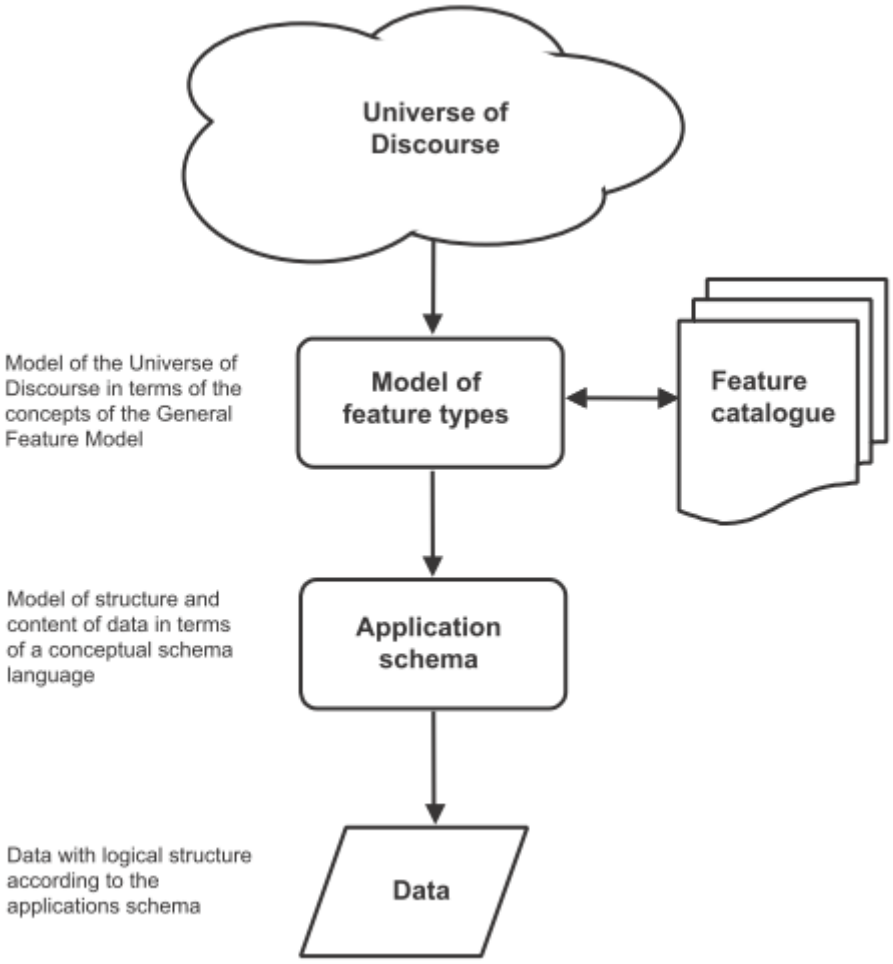

Figure 3. From reality to geographic data

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 1. 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 has included the concept of the application schema from its earlier versions (i.e. an XML Schema). Nevertheless, there are formats that do 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 "ExtendedData/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.

5.2. 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 has 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"
        }
    }
}
```

5.2.1. 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 returns a feature collection, RFC7946 overlooks that WFS deeply uses the *Feature Type* concept that is missing in GeoJSON.

5.3. 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 needs 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)
- 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 one by one.

5.3.1. 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 serve 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.

5.3.2. 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 these 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"
        }
    }
}
```

5.3.3. 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 multidimensional 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 defined by an 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) is doing (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 demonstration on how it works can be found here: https://covjson.org/playground/ (tested with Chrome).

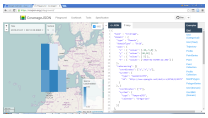

Figure 4. CoveragesJSON playground dummy example for continuous values in http://covjson.org

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 [source,json]

```
{
    "@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 6. The temporal dimension in JSON

We have talked before about the geometrical dimension in Geospatial_dimension_in_JSON. As the digitalization of the world progresses, features are captured several times, moving features are introduced and models generate forecasts of the future, the temporal dimension is becoming more relevant and also more interrelated with the geometrical dimension. The temporal dimension can be characterized by instants, time intervals, and a list of instants and intervals. This section discusses how to encode the temporal dimension in JSON.

6.1. Time and data instants

In JavaScript, dates are managed by objects created by with "new Date()".

NOTE	In several places of this document we argue that there are no object classes in JavaScript and the use of object constructors are not encouraged anymore. *Dates* are still an exception in JavaScript. In JavaScript the objects created with new Date() provides many useful methods and *date* constructor is still widely used. Since methods are not allowed in JSON, objects created with new Date() cannot be adopted, creating a gap that needs to be solved.

The list of methods that an object created with new Date() can be used to deal with dates is described here (http://www.w3schools.com/jsref/jsref_obj_date.asp). In JSON we can represent object properties but methods are not allowed for security reasons, so the use of the Date() methods is excluded. JSON does not provide an alternative to dates this leaving a gap in the JSON data types. We have two alternatives: define a JSON date object pattern or use strings that represent dates.

6.1.1. Instants as complex objects

One possible alternative could be to use a complex data type with numbers separating the different parts of the date in numerical properties.

Example of JSON date as a complex type alternative

```
{
    "year": 2016,
    "month": 9,
    "day": 7,
    "hour": 8,
    "minute": 2,
    "second": 1
}
```

Then we will be able to transform the complex date into a JavaScript Date() using: .Example of conversion of a JSON date into a Date()

```
{
    var d = JSON.parse("{"year": 2016, "month": 9, "day": 7, "hour": 8, "minute": 2,
"second": 1}");
    var date = new Date(d.year, d.month, d.day, d.hour, d.minute, d.second);
}
```

6.1.2. Time instants as strings

Another alternative is to represent dates as JSON strings. Actually, JSON Schema provides a format parameter for the string time "format": "date-time". The date representation corresponding to this format, is defined following the RFC 3339, section 5.6 (http://tools.ietf.org/html/rfc3339) that is a profile of the ISO 8601. See DateTime_Format for more details. In brief, this standard defines a string format of the date that follows the pattern *YYYY-MM-DDTHH:mm:ss.sssZ* where *YYYY* is the year, *MM* is the month (always in two digits with a 0 in the left if necessary), *DD* is the day (always in two digits), *HH* is the hour (always in two digits) *mm* are the minutes (always in two digits), *ss* are the seconds (always in two digits) and *sss* are the milliseconds (always in three digits).

The JavaScript method Date().toJSON() converts a Date object into a string, formatted as an string date following the ISO-8601 standard format. Date.parse() does the opposite job.

In the different encodings we have examined, expressing dates as strings seems to be the preferred alternative.

6.2. Time intervals

Again we can describe two possible alternatives, complex date type intervals or two strings in an array.

6.2.1. Time intervals as complex objects

One possible alternative could be to use a complex data type with two separated time properties called *start* and *end*.

Example of JSON time interval as a complex type alternative.

```
{
    "start": "2016-09-07T07:50:21.552Z",
    "end": "2016-09-08T07:50:21.552Z",
}
```

In the different encodings we have examined, this seems to be the preferred alternative. For example, it is suggested by Sean Gillies in GitHub here: https://github.com/geojson/geojson-ld/blob/master/time.md

6.2.2. Time intervals as arrays

Another possibility for expressing time intervals is use an array of two strings with and string time

format

Example of JSON time interval as arrays of strings.

```
{
    "interval": ["2016-09-07T07:50:21.552Z", "2016-09-08T07:50:21.552Z"]
}
```

This approach can be combined also with the use of a time list that can include both instantaneous times and intervals:

Example of JSON time list combining time instants and time intervals

```
{
    "timeList": ["2016-08-07T07:50:21.552Z", "2016-08-17T07:50:21.552Z", ["2016-09-
07T07:50:21.552Z", "2016-09-08T07:50:21.552Z"], "2016-10-07T07:50:21.552Z"]
}
```

Chapter 7. JSON in web services

The previous sections of this document have concentrated on 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.

7.1. 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.

7.1.1. 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 sent 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 as 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 trusts. If the server declares that it 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);
}
```

7.1.2. 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.

7.1.3. 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 interpret 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 HTTP_Server_response.

7.1.4. 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 HTTP_Server_response, 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.

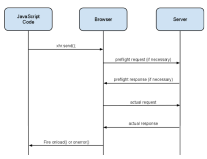

Figure 5. CORS flow in case of a POST request

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.