OGC Testbed-14

Application Schemas and JSON Technologies Engineering Report

Table of contents

1. Summary . . . 4
1.1. Requirements & Research Motivation . . . 4
1.2. Recommendations for Future Work . . . 4
1.2.1. Develop a new version of the ShapeChange JSON Schema target . . . 5
1.2.2. Develop JSON Schemas for ISO schemas . . . 5
1.3. Document contributor contact points . . . 5
1.4. Foreword . . . 6
2. References . . . 7
3. Terms and definitions . . . 8
3.1. Abbreviated terms . . . 8
4. Overview . . . 10
5. Enhancements for JSON Schema Conversion . . . 12
5.1. Overview . . . 12
5.2. Schema conversion with JSON Schema draft 07 . . . 13
5.2.1. Conversion of an application schema and its classes . . . 13
5.2.2. Documentation . . . 17
5.2.3. Conversion of UML <<union>> classes . . . 18
5.2.4. Conversion of generalization/inheritance . . . 19
5.2.5. Fixed / constant properties . . . 25
5.3. Enhancing the implementation of the ShapeChange JSON Schema encoding . . . 26
5.3.1. Leverage ShapeChange transformers . . . 26
5.3.2. Map entries for GeoJSON geometry types . . . 27
5.3.3. Defining conversion rules . . . 28
6. Defining the semantics of JSON data through the use of JSON-LD . . . 30
6.1. Overview . . . 30
6.2. Converting GeoJSON data to NEO RDF data . . . 38
6.2.1. Developing a JSON-LD @context . . . 38
6.2.2. Identified issues . . . 48
6.2.2.1. Mismatch between simple JSON structure and complex NEO structure . . . 48
6.2.2.2. Numeric code values . . . 50
6.2.2.3. NAS/NEO value or reason pattern . . . 53
6.2.2.4. NEO geometry representation does not use GeoSPARQL . . . 55
6.2.3. Potential solutions . . . 55
6.2.3.1. Semantically-enable JSON, without serializing as RDF . . . 55
6.2.3.2. Purpose built intermediate ontology . . . 61
6.3. Recommendations and best practices . . . 62
6.3.1. Context dependent mappings . . . 62
6.3.2. Handling geometry . . . 65

6.3.3. JSON-LD keywords . . . 66
6.4. Enhancing ShapeChange to derive JSON-LD @context documents . . . 68
7. Using both JSON Schema and JSON-LD . . . 69
Annex A: Revision History . . . 70
Annex B: Bibliography . . . 71

Publication Date: 2019-02-05

Approval Date: 2018-12-13

Submission Date: 2018-11-21

Reference number of this document: OGC 18-091r2

Reference URL for this document: http://www.opengis.net/doc/PER/t14-D022-2

Category: OGC Public Engineering Report

Editor: Johannes Echterhoff

Title: OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report

---

OGC Public Engineering Report

COPYRIGHT

Copyright (c) 2019 Open Geospatial Consortium. To obtain additional rights of use, visit http://www.opengeospatial.org/

WARNING

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Public Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

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

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

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

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

indicate compliance with any LICENSOR standards or specifications.

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

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Summary

This Engineering Report (ER) enhances the understanding of the relationships between data exchange based on Geography Markup Language (GML), JavaScript Object Notation (JSON), and Resource Description Framework (RDF) for future web services, e.g. Web Feature Service (WFS) 3.0. The work documented in this report:

- contributes to the ability to bridge between technology-dependent alternate representations of “features” (real-world objects), and to consistently employ alternate encoding technologies (Extensible Markup Language (XML), JSON, RDF) to exchange information about “features”; and
- determines principled techniques for the development of JSON-based schemas from ISO 19109-conformant application schemas.

1.1. Requirements & Research Motivation

The following requirements pertaining to increasing the understanding of state-of-the-art JSON technologies have been addressed by the work documented in this ER:

- Analyze the current version of the JSON Schema specification, as well as the schema transformations currently supported by ShapeChange, to ultimately enhance the ShapeChange capability of deriving a JSON Schema from an application schema in Unified Modeling Language (UML). The current ShapeChange target to derive JSON Schema is based on JSON Schema draft v3. The goals are to:
  - make use of new features in the latest version of JSON Schema, to improve the conversion from an application schema in UML to JSON Schema, and to
  - leverage existing transformation capabilities of ShapeChange, to facilitate re-use.
- Investigate how keys (i.e., property names) used in JSON instance data can be mapped to terms from one or more ontologies through JSON-LD. This work is intended to enable mapping of JSON instance data conforming to the NSG Application Schema (NAS) to RDF individual data conformant to the NSG Enterprise Ontology (NEO). The analysis shall also identify which documents ShapeChange can derive to support such a mapping.

1.2. Recommendations for Future Work

This ER increases the OGC community’s understanding of state-of-the-art JSON, JSON Schema, and JSON for Linked Data (JSON-LD) technologies as applied to the encoding of an ISO 19109-conformant Unified Modeling Language (UML) application schema. This ER builds upon, and extends, the analysis of such an encoding that was performed in Testbed-12, and is documented in the Testbed-12 ShapeChange ER [http://docs.opengeospatial.org/per/16-020.html#_json_json_schema_and_json_ld].

This ER analyzes draft-07 of the JSON Schema specification, with the goal of developing rules and tools that implement the process of encoding an ISO 19109-conformant UML application schema as a corresponding JSON Schema. With such assets, it would be possible to perform more rigorous validation of JSON data, similar to what is available for XML encoded data. This ER provides useful knowledge to any OGC member who has an application schema in UML, and needs to convert that

schema to a corresponding JSON Schema based upon well-defined rules using model-driven engineering tools.

This ER investigates how the semantics of JSON data can be defined through the use of JSON-LD. The analysis identifies a number of issues, potential solutions, as well as recommendations and best practices. This ER extends the JSON community's understanding of the limits of what can be achieved using JSON-LD regarding semantic annotation and mapping to RDF.

The following sections document work items that should be addressed next.

1.2.1. Develop a new version of the ShapeChange JSON Schema target

The current ShapeChange target for deriving JSON Schema from an application schema in UML is restricted to producing JSON Schemas that check GeoServices JSON feature data. The target produces schemas that comply with JSON Schema draft 03. Since the development of the ShapeChange JSON Schema target in OGC Testbed 9 (then referred to as OWS-9), both ShapeChange and the JSON Schema specification have evolved significantly.

The analysis of JSON Schema in OGC Testbed-14 has identified a number of improvements for producing JSON Schemas with ShapeChange. Those improvements are documented in Enhancing the implementation of the ShapeChange JSON Schema encoding. A new version of the ShapeChange JSON Schema target should be developed, that realizes these improvements. This would greatly benefit the geospatial community.

NOTE	As described in Enhancing the implementation of the ShapeChange JSON Schema encoding, having an up-to-date implementation of JSON Schema could be of particular interest for WFS 3.0 users.

An extension of the JSON Schema target could be to have it produce JSON-LD @context documents as well, which is described in more detail in section Enhancing ShapeChange to derive JSON-LD @context documents

1.2.2. Develop JSON Schemas for ISO schemas

Previous engineering reports (the OWS-9 SSI UGAS ER ([1]) and the OGC Testbed-12 ShapeChange ER ([2])) recommended that JSON Schemas be developed for ISO schemas - or profiles of those schemas (e.g. ISO 19107, ISO 19108, ISO 19115, and ISO 19157). Such JSON Schemas would facilitate interoperable, JSON-based implementations of application schemas that rely on these ISO schemas.

A future Testbed could develop drafts of JSON Schemas for a select set of ISO schemas. OGC working groups could take the results and work towards standardizing these JSON Schemas.

1.3. Document contributor contact points

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

Contacts

Name	Organization
Paul Birkel	Geosemantic Resources LLC
Johannes Echterhoff (editor)	interactive instruments GmbH

1.4. Foreword

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

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

Chapter 2. References

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

- ISO: ISO 19125-1, Geographic information — Simple feature access — Part 1: Common architecture, 2004
- IETF: RFC 8259, The JavaScript Object Notation (JSON) Data Interchange Format, available online at https://tools.ietf.org/html/rfc8259
- W3C: JSON-LD 1.1 - A JSON-based Serialization for Linked Data, W3C Final Community Group Report 07 June 2018, available online at https://www.w3.org/2018/jsonld-cg-reports/json-ld/
- OGC: OGC 11-052r4, OGC GeoSPARQL - A Geographic Query Language for RDF Data, available online at http://www.opengis.net/doc/IS/geosparql/1.0
- W3C: OWL 2 Web Ontology Language, Direct Semantics (Second Edition), W3C Recommendation 11 December 2012, available online at http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/
- W3C: OWL 2 Web Ontology Language, Structural Specification and Functional-Style Syntax (Second Edition), W3C Recommendation 11 December 2012, available online at http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/
- W3C: RDF 1.1 Primer, W3C Working Group Note 24 June 2014, available online at http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/
- W3C: SKOS Simple Knowledge Organization System Reference, W3C Recommendation 18 August 2009, available online at http://www.w3.org/TR/2009/REC-skos-reference-20090818/

Chapter 3. Terms and definitions

- **Linked Data**:

> Linked Data is the data format that supports the Semantic Web. The basic rules for Linked Data are defined as:
>
> - Use Uniform Resource Identifiers (URIs) to identify things.
> - Use HTTP URIs so that these things can be referred to and looked up ("dereferenced") by people and user agents.
> - Provide useful information about the thing when its URI is dereferenced, using standard formats such as RDF/XML
> - Include links to other, related URIs in the exposed data to improve discovery of other related information on the Web.
>
> — W3C Semantic Web Wiki [https://www.w3.org/2001/sw/wiki/Semantic_Web_terminology#linked_data]

3.1. Abbreviated terms

CIS	Coverage Implementation Schema
DDL	Data Definition Language
DTD	Document Type Declaration
ER	Engineering Report
GEOINT	Geospatial Intelligence
GML	Geography Markup Language
IRI	Internationalized Resource Identifier
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON-LD	JSON for Linked Data
LOCN	Location Core Vocabulary
NAS	NSG Application Schema
NCV	NSG Core Vocabulary
NEO	NSG Enterprise Ontology
NSG	U.S. National System for Geospatial Intelligence
OCL	Object Constraint Language
OGC	Open Geospatial Consortium
OWL	Web Ontology Language
RDF	Resource Description Framework

RDFS	RDF Schema
SKOS	Simple Knowledge Organization System
SPARQL	SPARQL Protocol and RDF Query Language
TSC	Technical Steering Committee
UGAS	UML to GML Application Schema
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WFS	Web Feature Service
WKT	Well-Known Text
XML	Extensible Markup Language

Chapter 4. Overview

JSON is a popular data encoding, particularly because it can be easily processed by many categories of web applications. The structure of JSON data is fairly simple (for details, see the ECMA standard 404 [http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf]) and is sufficient for many applications. However, for a distributed system that requires applications to seamlessly interoperate, JSON itself lacks some key features:

- Ability to define the semantics of a JSON object and its properties, i.e. the set of the name/value pairs contained in a JSON object: An indication of the type of a JSON object returned by a general data provider would help an application to process the object correctly. Furthermore, the same property name may have completely different meaning for different applications. Namespaces as supported by XML would help, but JSON does not support namespaces. Therefore, a mechanism to encode the intended meaning of a given property is needed.
- Ability to reference other JSON objects: JSON supports 'inline' encoding of other objects, i.e. an object can encode another object as a key value. However, self-references as well as cross-references from any object to any other object are not supported.
- Ability to specify the allowed values of a key: It would be valuable for applications to know which types of value can be expected for a specific key of a JSON object.
- Ability to specify the structure of a JSON object, and to validate a given object against this specification: When information is exchanged between applications, they typically expect the information to be encoded with a specific structure. This may include, for example, the expected multiplicity of properties. Validation is used to ensure that data is structured as expected.

Fortunately, additional specifications exist to fill these gaps:

- JSON-LD [https://www.w3.org/TR/json-ld/]: Supports definition of the semantics of JSON objects and their properties, as well as references to other objects and value types of properties.
- JSON Schema [http://json-schema.org/]: Defines how a JSON document shall be structured. It can be used for validating JSON data.

NOTE

The situation is somewhat similar for XML encoded data. The W3C XML standard [https://www.w3.org/TR/xml/] defines the basics for encoding data in XML. It also defines a means to validate data through document type declarations (DTD). However, the most commonly used form of validating XML data at the present time is through XML Schema, which is defined in an additional set of standards [https://www.w3.org/XML/Schema#dev]. Assigning different semantics to XML elements with same (local) name can be achieved through XML namespaces [https://www.w3.org/TR/xml-names/], which is yet another W3C standard. Additional standards may define further rules for encoding information in XML. The ISO/OGC Geography Markup Language [http://www.opengeospatial.org/standards/gml], for example, specifies how to encode objects (that are defined by an application schema), as well as their properties and property values, such as geometries and references to other objects.

Testbed-14 analyzed the latest versions of JSON Schema and JSON-LD.

Chapter 5 documents an analysis of potential enhancements in the conversion from an application schema in UML to a JSON Schema.

Chapter 6 documents an analysis on mapping of keys (i.e., property names) used in JSON instance data to an ontology through JSON-LD, the use case being mapping NAS-conformant JSON instance data to NEO-conformant RDF individual data.

Chapter 7 documents some theoretical considerations for the combined use of JSON Schema and JSON-LD.

NOTE

When exchanging JSON data with other entities, one needs to be aware of potential interoperability and security issues. According to the article Parsing JSON is a Minefield [http://seriot.ch/parsing_json.php], libraries for parsing JSON data tend to behave differently. The article states that the reason for the different parsing behavior is mainly caused by the JSON libraries relying *"on specifications that have evolved over time and that left many details loosely specified or not specified at all"*.

Certain JSON data can apparently cause an application to crash, and would thus be a means to perform a Denial-of-Service (DoS) attack. The article comes to this conclusion based upon the results of an extensive set of tests that have been executed with different JSON libraries. Furthermore, while one JSON parser may succeed in reading JSON data, a different parser may fail to read the same data. Thus, while processing JSON data in one system may be successful, it is not guaranteed to work with another system, or if JSON libraries are being replaced (especially with a different library).

Chapter 5. Enhancements for JSON Schema Conversion

5.1. Overview

Many applications - in particular web applications - use JSON to encode and exchange information. Being able to ensure that JSON data is encoded correctly, i.e. with an expected structure, is important. As described in the introduction of this chapter, JSON itself does not provide a mechanism to validate the content of JSON data. Such a validation capability is defined by the JSON Schema specification.

In OGC Testbed 9, the encoding of an application schema as a JSON Schema was analyzed and implemented in ShapeChange. That work was based on JSON Schema draft 03 [https://tools.ietf.org/html/draft-zyp-json-schema-03], then current [3]. The specification has subsequently evolved to JSON Schema draft 07.

NOTE JSON Schema draft 07 consists of three documents, JSON Schema core [https://tools.ietf.org/html/draft-handrews-json-schema-01] ([4]), a schema for validation [https://tools.ietf.org/html/draft-handrews-json-schema-validation-01] ([5]), and a hyper schema [https://tools.ietf.org/html/draft-handrews-json-schema-hyperschema-01] ([6]). For the analysis in Testbed-14, the first two documents are of primary interest. Note also that the website http://www.json-schema.org provides a list of the current and older drafts of the specification, as well as the latest unreleased version [http://json-schema.org/specification-links.html]. The site also provides an overview of existing JSON Schema implementations [http://json-schema.org/implementations.html].

The following sections document the results of an analysis of the set of features supported by JSON Schema draft 07, as well as the current implementation of ShapeChange. The goal of this analysis was to identify:

- how the new features of JSON Schema draft 07 (when compared to draft 03) can improve the conversion of an application schema in UML to a JSON Schema, and
- how the implementation of the ShapeChange JSON Schema target can be enhanced, to realize these improvements and to leverage new transformation capabilities of ShapeChange.

Having a well-defined - and, ideally, automated - process for deriving JSON Schemas from application schemas will be an important means to increase the level of interoperability when using OGC web services to exchange JSON data. Without JSON Schema, services and clients have to either consume incoming JSON data and "hope for the best" regarding the correctness of the data structure, or apply (and potentially implement) a custom inspection algorithm to ensure that the data is as expected. If, however, a JSON Schema is available, services and clients can use it to validate the data, using a single, well-defined mechanism (and leverage existing implementations to perform the validation).

NOTE The Coverage Implementation Schema (CIS) version 1.1 [http://docs.opengeospatial.org/is/09-146r6/09-146r6.html] is an example of an OGC standard that defines a JSON Schema (the schema is available online [http://schemas.opengis.net/cis/1.1/json/]).

NOTE JSON Schema is also of interest for the OGC Web Feature Service (WFS) 3.0 standard, which is specified as a set of reusable OpenAPI components. WFS 3.0 will support multiple encodings of feature data, for example JSON.

At the time when this report was written, the OpenAPI Technical Steering Committee (TSC) considered an extension that would support specifying alternate schemas - in addition to the OpenAPI Schema Object [https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md#schema-object] (which is an extended subset of JSON Schema draft 05).

The extension would allow defining a JSON Schema which is not restricted to the expressiveness of the OpenAPI Schema Object for validating feature data that is published by a WFS in JSON encoding. The extension would also support validating XML encoded feature data with XML Schema and Schematron Schema. The WFS issue tracker has an entry for OpenAPI Alternate Schema Support [https://github.com/opengeospatial/WFS_FES/issues/129] that provides further details.

5.2. Schema conversion with JSON Schema draft 07

The following subsections document how new features of JSON Schema draft 07 (compared to 03) can be used to improve the conversion of an application schema into a JSON Schema.

NOTE This section does not provide a detailed comparison of JSON Schema drafts 03 and 07. The JSON Schema draft 07 documents each contain an appendix with a changelog, which describes relevant changes to prior versions.

NOTE This section is not intended to define a full set of rules for the conversion to JSON Schema. The development of such rules is part of a future work item.

NOTE https://www.jsonschemavalidator.net/ is a useful resource to test the JSON Schema examples contained in this section.

5.2.1. Conversion of an application schema and its classes

The *definitions* keyword of JSON Schema, in combination with keyword *$ref*, allows us to define a single JSON Schema file with schema definitions for all classes of an application schema.

Listing 1. Example of a JSON Schema with 'definitions'

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "$id": "http://shapechange.net/tmp/tb14/Example_definitions.schema.json",
 "definitions": {
  "Class1": {
   "type": "object",
   "properties": {
    "prop1": {"type": "string"}
   },
   "required": ["prop1"]
  },
  "Class2": {
   "type": "object",
   "properties": {
    "prop2": {"type": "number"}
   },
   "required": ["prop2"]
  }
 }
}
```

The example defines the schemas of two classes. The definitions schema can be accessed in one of two ways.

- If one of the schema definitions shall be accessed from within the same file (e.g. if Class1 had properties of type Class2), a simple JSON Pointer [https://tools.ietf.org/html/rfc6901] ([7]) suffices, for example: #/definitions/Class2
- Otherwise (a reference is made from outside of the definitions file), a URL that includes the JSON Pointer as fragment identifier can be used, for example http://shapechange.net/tmp/tb14/Example_definitions.schema.json#/definitions/Class2.

NOTE The JSON Schema that shall be used to validate a JSON document cannot be identified within that document itself. In other words, JSON Schema does not define a concept like an xsi:schemaLocation, which is typically used in an XML document to reference the applicable XML Schema(s). Instead, JSON Schema uses link headers and media type parameters to tie a JSON Schema to a JSON document (for further details, see JSON Schema core ([4]), sections 10.1 and 10.2). The relationship between a JSON document and the JSON Schema for validation can also be defined explicitly by an application.

The definitions schema in Listing 1 does not identify a particular schema to use for validating a JSON document. In order to use the definitions schema, another JSON Schema is needed that declares which of the definitions applies. The JSON Schema of Listing 2 simply references the schema definition of Class2 from the definitions file (using a URL with a JSON Pointer fragment).

Listing 2. Example of a JSON Schema that references the schema of 'Class2', defined by an external JSON Schema

```
{
  "$ref":
"http://shapechange.net/tmp/tb14/Example_definitions.schema.json#/definitions/Class2"
}
```

This JSON object is valid against the schema of Listing 2:

```
{"prop2": 42}
```

This JSON object is invalid (wrong type of "prop2") against the schema of Listing 2:

```
{"prop2": "Dent"}
```

A self-contained example of the definitions schema, to be used for testing, would be:

Listing 3. Self-contained example of a JSON Schema with 'definitions' - for testing

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "definitions": {
  "Class1": {
   "type": "object",
   "properties": {
    "prop1": {"type": "string"}
   },
   "required": ["prop1"]
  },
  "Class2": {
   "type": "object",
   "properties": {
    "prop2": {"type": "number"}
   },
   "required": ["prop2"]
  }
 },
 "oneOf": [
  {"$ref": "#/definitions/Class1"},
  {"$ref": "#/definitions/Class2"}
 ]
}
```

The example illustrates the uses of one of the JSON Schema keywords *allOf*, *anyOf*, *oneOf*, and *not*, which can be used to define logical combinations of JSON schemas (see Table 1).

Table 1. Boolean logic represented by JSON Schema keywords

JSON Schema keyword	Represented boolean logic
allOf	AND
anyOf	OR
oneOf	XOR
not	NOT

In this case, *oneOf* means that the JSON document must be valid for exactly one of the defined schemas (Class1 xor Class2).

This approach, however, should only be used in a definitions schema if performance is not critical. The reason is that a validator has to evaluate the JSON document against two or more of the schemas referenced by *oneOf*. In the worst case, the validator has to check the document against all of the schemas.

Having a single definitions schema would be beneficial to caching. An application could load the whole schema once, cache it, and then efficiently re-use it for validating JSON objects with different structures. However, an application may only need one particular portion of the definitions schema for data validation. Then having a large definitions schema, even locally cached, may result in significant overhead. The situation would get more complicated if individual schemas have cross-references to other schemas. It would also be possible to have both a definitions schema and individual class schemas (as illustrated in Figure 1), or to have multiple definitions schemas (e.g. one per package contained in the application schema).

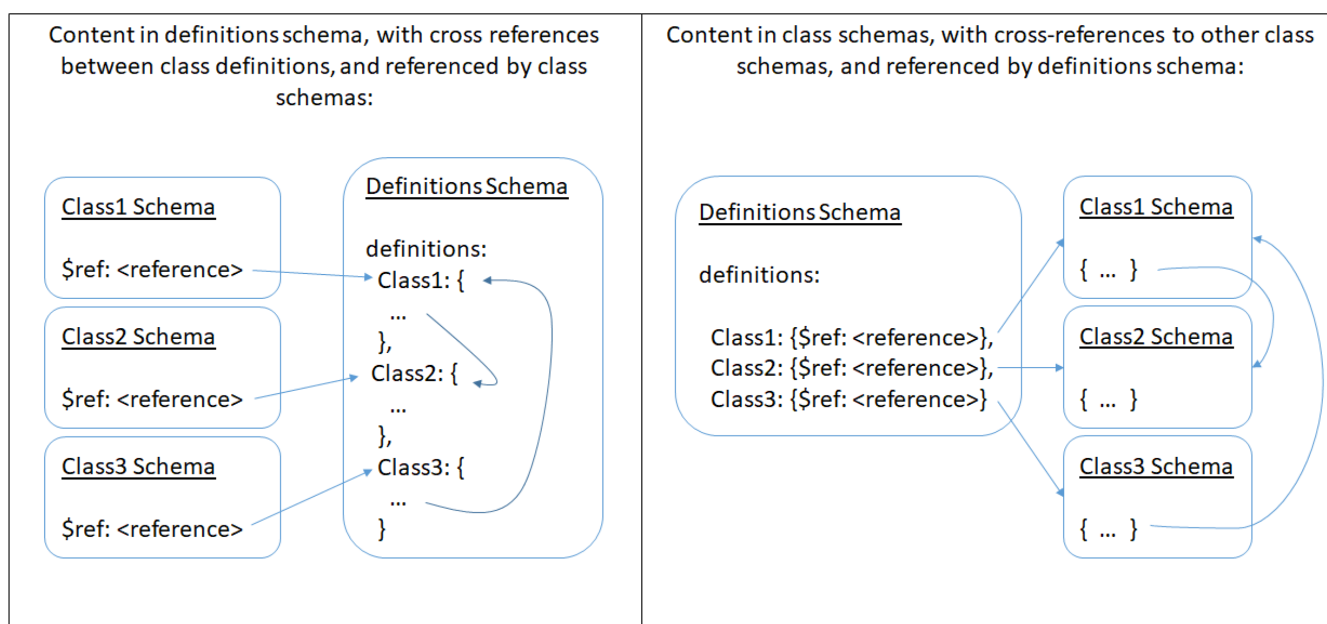

Figure 1. Examples of how JSON Schemas can be derived from an application schema

The *definitions* keyword opens up a range of possibilities for creating JSON Schemas for the classes defined by an application schema. It is difficult to say which partitioning and organization of JSON Schemas would be best suited for a given use case. Further testing and practical experience from real use cases is needed.

5.2.2. Documentation

JSON Schema supports a number of annotations, which can be used to provide commonly used information for documentation and user interface display purposes:

- *title* and *description* - Can be used to provide short and long descriptions (available since JSON Schema draft 03).
- *default* - Can be used to indicate a default value for a property (available since JSON Schema draft 03).
- *readOnly* and *writeOnly* - A read-only key can only be changed by the authority that owns a JSON instance, while a write-only key will never be returned when retrieving the resource (e.g. a password). These annotations appear to be geared towards use with JSON hyper schema ([6]). However, *readOnly* could be used when converting a UML property that is marked as fixed/constant and which does not have a defined initial value. If it had an initial value, the keyword *const* should be used (see section Fixed / constant properties).
- *examples* Can be used to provide sample values, for illustrating purposes.

The JSON Schema core ([4]) states that: *"A JSON Schema MAY contain properties which are not schema keywords. Unknown keywords SHOULD be ignored."* This can be useful for providing application (domain) specific metadata about JSON keys. For example, one could define that the unit of measure of a number-valued JSON key is meter.

NOTE	JSON Schema core also defines the *$comment* keyword: *"This keyword is reserved for comments from schema authors to readers or maintainers of the schema."* $comment can be used to describe a schema object, much like an XML comment.

The example in Listing 4 shows a JSON Schema that makes use of several of these documentation keywords and options.

Listing 4. Example of a JSON Schema with documentation

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Schema title",
  "description": "Description of the schema",
  "type": "object",
  "$comment": "This is a comment",
  "properties": {
    "propX": {
      "type": "number",
      "description": "Description of propX.",
      "examples": [
        2,
        4
      ],
      "readOnly": true,
      "uom": "m",
      "$comment": "This is a comment"
    }
  },
  "required": [
    "propX"
  ]
}
```

5.2.3. Conversion of UML <<union>> classes

The keyword *oneOf* can be used to represent a <<union>> class in JSON Schema.

Listing 5. Example of a JSON Schema for a <<union>> class

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "oneOf": [
    {
      "properties": {
        "option1": {
          "type": "string"
        }
      },
      "required": [
        "option1"
      ]
    },
    {
      "properties": {
        "option2": {
          "type": "number"
        }
      },
      "required": [
        "option2"
      ]
    }
  ]
}
```

The following two JSON objects are both valid against the schema:

```
{"option1": "Arthur"}

{"option2": 42}
```

The following two JSON objects, on the other hand, are invalid (wrong type, or not covered by the choices of *oneOf*):

```
{"option1": 2}

{"option3": 11}
```

5.2.4. Conversion of generalization/inheritance

JSON Schema was not designed to support concepts of object-oriented modeling, like inheritance. A JSON Schema simply defines a collection of constraints for a JSON document. However, it is possible to combine JSON Schemas using boolean operations to mimic inheritance, in particular the JSON

Schema keyword *allOf*.

NOTE	The topic of inheritance has been discussed in an issue of the json-schema-org/json-schema-org.github.io GitHub repository [https://github.com/json-schema-org/json-schema-org.github.io/issues/148]. One of the comments [https://github.com/json-schema-org/json-schema-org.github.io/issues/148#issuecomment-347334235] indicates that JSON Schema draft 08 might enhance support for modularity and re-usability, including ways to support inheritance.

The keyword *allOf* can be used to include constraints from a supertype in the JSON Schema of a subtype, with one specific design requirement: neither of the schemas must define *"additionalProperties":false*.

Listing 6. Example of a JSON Schema that mimics inheritance

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "definitions": {
  "Supertype1": {
   "type": "object",
   "properties": {
    "prop1": {"type": "string"}
   },
   "required": ["prop1"]
  },
  "Supertype2": {
   "type": "object",
   "allOf": [
    {"$ref": "#/definitions/Supertype1"}
   ],
   "properties": {
    "prop2": {"type": "number"}
   }
  }
 },
 "type": "object",
 "allOf": [
  {"$ref": "#/definitions/Supertype2"}
 ],
 "properties": {
  "prop2": {
   "type": "number",
   "minimum": 1
  },
  "prop3": {"type": "boolean"}
 },
 "required": ["prop3"]
}
```

In Listing 6, the main schema defines constraints for properties "prop2" and "prop3". In addition, it

includes all constraints defined by the schemas for "Supertype2", which, by the same mechanism, includes all constraints from the schema of "Supertype1".

NOTE The schemas for the supertypes are defined in the same JSON Schema only to get a self-contained example. As discussed in Conversion of an application schema and its classes, there are a number of ways to derive JSON Schema(s) from an application schema.

The schema for Supertype1 defines the required property "prop1". The schema for Supertype2 defines the optional property "prop2" with value type number. The main schema defines an additional constraint on "prop2": it restricts the value range to positive integers. This is an example for how restricting a property in UML through override or an Object Constraint Language (OCL) constraint can be represented in JSON Schema. Finally, the main schema also defines the required property "prop3".

The following two JSON objects are both valid against the schema:

```
{
  "prop1": "Arthur",
  "prop3": true
}

{
  "prop1": "Mary",
  "prop2": 5,
  "prop3": false
}
```

The following JSON object, on the other hand, is invalid (value of "prop2" is a negative integer):

```
{
  "prop1": "Arthur",
  "prop2": -1,
  "prop3": true
}
```

NOTE If one of the schemas from the example would define *"additionalProperties":false*, then no JSON document could be valid against the main schema.

Another aspect to consider regarding validation of JSON objects whose types are part of an inheritance hierarchy, is what this means for the validation of a JSON key value. If, for example, the UML property "propX" had type "Supertype", then in the JSON Schema that defines "propX", the definition of the "Supertype" schema can be referenced to validate the value of "propX".

Listing 7. Example of a JSON Schema with property type referencing the schema of a supertype

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "definitions": {
  "Supertype": {
   "type": "object",
   "properties": {
    "propS1": {"type": "string"}
   },
   "required": ["propS1"]
  },
  "Subtype": {
   "type": "object",
   "allOf": [
    {"$ref": "#/definitions/Supertype"}
   ],
   "properties": {
    "propS2": {"type": "number"}
   },
   "required": ["propS2"]
  }
 },
 "type": "object",
 "properties": {
  "propX": {"$ref": "#/definitions/Supertype"}
 },
 "required": ["propX"]
}
```

The following two JSON objects are both valid against the schema of Listing 7:

```
{
  "propX": {
    "propS1": "C"
  }
}

{
  "propX": {
    "propS1": "C",
    "propS2": "D"
  }
}
```

The second JSON object shows that validation of "propX" only checks the constraints defined by the "Supertype" schema. The schema of its "Subtype" is ignored (otherwise, "propS2" would have been flagged as invalid, since its value is a string, not a number).

An application can validate a property value with different JSON Schemas if the value has a unique characteristic, for example a property that identifies the type of the given JSON object. In the example of Listing 8, "t" is such a property.

Listing 8. Example of a JSON Schema with value-dependent validation

```
{
 "$schema": "http://json-schema.org/draft-07/schema#",
 "definitions": {
  "Supertype": {
   "type": "object",
   "properties": {
    "t": {"type": "string"},
    "propS1": {"type": "string"}
   },
   "required": [
    "t",
    "propS1"
   ]
  },
  "Subtype": {
   "type": "object",
   "allOf": [
    {"$ref": "#/definitions/Supertype"}
   ],
   "properties": {
    "propS2": {"type": "number"}
   },
   "required": ["propS2"]
  }
 },
 "type": "object",
 "properties": {
  "propX": {
   "oneOf": [
    {
     "if": {
      "properties": {
       "t": {"const": "Supertype"}
      }
     },
     "then": {"$ref": "#/definitions/Supertype"},
     "else": false
    },
    {
     "if": {
      "properties": {
       "t": {"const": "Subtype"}
      }
     },
     "then": {"$ref": "#/definitions/Subtype"},
```

```
      "else": false
    }
   ]
  }
 },
 "required": ["propX"]
}
```

This JSON Schema contains two definitions, those of a supertype and its subtype. The main schema defines required property "propX", which must be valid against exactly one of the cases defined using an if-then-else construct. The "if" defines a schema that checks the value of "t". If the value is "Supertype", then the value of "propX" must be valid against the supertype schema. If, on the other hand, the value of "t" is "Subtype", then "propX" must be valid against the subtype schema. In both if-then-else constructs, the else-case is explicitly set to false, to ensure that the condition does not evaluate to true if the if-case is not fulfilled.

The first of the following two JSON objects is valid against the schema, while the second is not (since propS2 is not a number):

```
{
  "propX": {
    "t": "Subtype",
    "propS1": "C",
    "propS2": 3
  }
}

{
  "propX": {
    "t": "Subtype",
    "propS1": "C",
    "propS2": "D"
  }
}
```

NOTE

Due to the rather complex constraints defined by the schema, it can be difficult to identify the exact cause of validation errors. For example, when validating the second JSON object against the schema on https://www.jsonschemavalidator.net/, the following messages were reported:

```
Found 4 error(s)

Message: JSON is valid against no schemas from 'oneOf'.
Schema path: #/properties/propX/oneOf

   Message: JSON does not match schema from 'then'.
   Schema path: #/definitions/Subtype/then

   Message: JSON does not match schema from 'else'.
   Schema path: #/properties/propX/oneOf/0/else/else

      Message: Schema always fails validation.
      Schema path: #/properties/propX/oneOf/0/else/valid
```

The approach to value-dependent validation illustrated in Listing 8 can be used to ensure full validation of a property value for a defined set of cases, for example all subtypes contained in the application schema. This approach relies on the type of a JSON object being correctly encoded in the data. Due to the restrictions of the representation of inheritance (using *allOf* to include the constraints of the supertype), it is not possible to have constraints in supertype and subtype schemas to validate the type (in Listing 8: the value of property "t").

NOTE This approach could also be used to validate the contents of a feature collection encoded in JSON.

If a JSON document contains JSON objects representing subtypes that are not covered by these cases, then in this approach validation will fail. In order to support such a scenario, the approach of Listing 7, a validation based solely on the supertype (which is defined as value type of the UML property) will need to be applied.

5.2.5. Fixed / constant properties

The JSON Schema keywords *readOnly* and *const* can be used to define properties in more detail.

When a UML attribute is defined as fixed/constant and has an initial value, the JSON Schema can define the property with *"const": <the initial value>*. The keyword can also be used to enable content validation of a property value in case that the value type is a supertype (see Listing 8).

If a fixed/constant UML property does not have an initial value, then the keyword *readOnly* can still be added to the JSON Schema definition of the property as additional documentation (also see section Documentation).

5.3. Enhancing the implementation of the ShapeChange JSON Schema encoding

ShapeChange has a target [https://shapechange.net/targets/json/] to encode an application schema as a JSON Schema. The target was implemented during OGC Testbed 9. It produces a JSON Schema based on draft 03. The target supports two encoding rules: *geoservices* and *geoservices_extended*. For further details, see the OGC Testbed-9 SSI UGAS Engineering Report ([1]).

Based upon a significant number of enhancements, both to ShapeChange and the JSON Schema specification, several opportunities for creating a revision of the ShapeChange JSON Schema target, and thus improving the JSON Schema encoding capabilities, have been identified. The following subsections document them in more detail.

5.3.1. Leverage ShapeChange transformers

The current ShapeChange JSON Schema target contains code to flatten an application schema. Inheritance, multiplicity, and complex types can thus be transformed, resulting in a simplified structure of the JSON Schema.

Since the development of the JSON Schema target, ShapeChange has been enhanced in a number of ways. A major enhancement was the introduction of model transformers. They essentially represent algorithms to modify (the application schemas of) a UML model in certain ways. Several model transformations have been implemented. One of them is the Flattener [https://shapechange.net/transformations/flattener/]. The Flattener realizes a number of transformation rules, including improved versions of the model transformations that the JSON Schema target currently implements. The code to perform these transformations should be removed from the JSON Schema target. Instead, whenever a flattened model is required for encoding as JSON Schema, the ShapeChange workflow should include the Flattener transformation. This would facilitate re-use, and allows leveraging existing (and future) transformation capabilities that the Flattener - as well as other ShapeChange transformers - provides. In addition, it will enable focus on the development of generic JSON Schema conversion rules.

The flattening functionality currently implemented by the JSON Schema target can be replaced by the following Flattener transformation rules:

- inheritance: *rule-trf-cls-flatten-inheritance* [https://shapechange.net/transformations/flattener/#rule-trf-cls-flatten-inheritance]
- multiplicity: *rule-trf-prop-flatten-multiplicity* [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-multiplicity]
- complex types: *rule-trf-prop-flatten-types* [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-types]

In addition, *rule-trf-prop-flatten-ONINAs* [https://shapechange.net/transformations/flattener/#rule-trf-prop-flatten-ONINAs] - or an updated version of the rule - could be useful for handling the encoding of null reasons in a flattened version of the NSG Application Schema (NAS). [1: https://nsgreg.nga.mil/nas/] A common approach for encoding NAS-based information in simple encodings (e.g. GeoJSON - see the example from the JSON-LD chapter) is to use specific string and numeric values to represent

null reasons. For example, depending on the value type of a property, the reason *noInformation* (which is one of the enums in the NAS void value reason enumerations) can be encoded as property value "noInformation" and -999999. For example, if no information was available for the value of feature property 'width', then in a GeoJSON representation of the feature, the property value would be -999999. *rule-trf-prop-flatten-ONINAs* can transform a model so that codes for void value reasons are copied into the actual value type (applicable for enumerations and boolean). NAS XxxReason classes can then be removed by the transformation.

Other transformations, such as the ConstraintConverter [https://shapechange.net/transformations/constraintconverter/], could also be of interest. For example, if an OCL constraint defined a regular expression that applied to a property, the transformation could extract the expression from the constraint and add it to the model in the form of tagged values. A new conversion rule of the JSON Schema target could then use the expression to define a pattern for the JSON Schema definition of the property.

5.3.2. Map entries for GeoJSON geometry types

When OGC Testbed 9 was conducted, JSON Schemas for GeoJSON were not available. Such schemas are now available [https://github.com/geojson/schema]. ShapeChange map entries can be defined for types of ISO 19107, to map them to corresponding GeoJSON schemas (see Listing9). The conversion of an application schema to a JSON Schema could use these mappings, resulting in a JSON Schema where geometries are encoded as GeoJSON geometries.

Listing 9. Map entries using GeoJSON geometry schemas

```
<?xml version="1.0" encoding="UTF-8"?>
<mapEntries xmlns="http://www.interactive-
instruments.de/ShapeChange/Configuration/1.1">
    <MapEntry type="GM_Point" rule="*" targetType=
"ref:http://geojson.org/schema/Point.json" param="geometry"/>
    <MapEntry type="GM_MultiPoint" rule="*" targetType=
"ref:http://geojson.org/schema/MultiPoint.json" param="geometry"/>
    <MapEntry type="GM_Curve" rule="*" targetType=
"ref:http://geojson.org/schema/LineString.json" param="geometry"/>
    <MapEntry type="GM_LineString" rule="*" targetType=
"ref:http://geojson.org/schema/LineString.json" param="geometry"/>
    <MapEntry type="GM_MultiCurve" rule="*" targetType=
"ref:http://geojson.org/schema/MultiLineString.json" param="geometry"/>
    <MapEntry type="GM_Surface" rule="*" targetType=
"ref:http://geojson.org/schema/Polygon.json" param="geometry"/>
    <MapEntry type="GM_Polygon" rule="*" targetType=
"ref:http://geojson.org/schema/Polygon.json" param="geometry"/>
    <MapEntry type="GM_MultiSurface" rule="*" targetType=
"ref:http://geojson.org/schema/MultiPolygon.json" param="geometry"/>
    <MapEntry type="GM_Object" rule="*" targetType=
"ref:http://geojson.org/schema/Geometry.json" param="geometry"/>
</mapEntries>
```

5.3.3. Defining conversion rules

The JSON Schema target of ShapeChange currently supports two fixed encoding rules: *geoservices* and *geoservices_extended*. Both encoding rules create a JSON Schema that is consistent with the GeoServices JSON feature model. However, the *geoservices* encoding rule creates a schema with a flattened and thus simplified structure, while the *geoservices_extended* encoding rule creates a non-flattened schema.

These two encoding rules are built into the JSON Schema target. Only a single conversion rule exists to modify the encoding behavior: *rule-json-all-notEncoded*. The conversion rule can be used to suppress encoding of a model element (application schema, class, property). The encoding behavior is more or less fixed. By adding new conversion rules, the JSON Schema target would be able to support more domain- and technology-specific encodings. In addition to the GeoServices JSON feature model, the target could also support the GeoJSON feature model. The JSON Schema produced by the target could also be structured in a completely different way, depending on the requirements of a given community.

NOTE	OGC WFS 3.0 users are a good example of a community that may need a specific set of JSON Schema encoding rules. WFS 3.0 is based on OpenAPI. OpenAPI defines validation rules for input and output datatypes through the OpenAPI Schema Object [https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md#schema-object], which is an extended subset of JSON Schema draft 05. Having JSON Schema conversion rules that support this "extended subset of JSON Schema" could lead to encoding rules for deriving OpenAPI Schema Objects from an application schema. This would be a useful capability for setting up a WFS 3.0. In addition, as discussed in an earlier note regarding WFS 3.0, the draft OpenAPI alternate schemas extension would allow specifying additional schemas for validating JSON data published by a WFS. These alternate schemas could be JSON Schemas derived using any kind of community specific JSON Schema encoding rule.

In order to support the creation of a wide range of encoding rules, the JSON Schema conversion rules need to focus on how the different parts of an application schema can be encoded in a JSON Schema. This may include having alternative rules for converting a specific aspect in different ways.

NOTE	Section 8.2 of the Testbed-12 ShapeChange Engineering Report [http://docs.opengeospatial.org/per/16-020.html#rdf_cr] can be used as an example, where rules for the conversion from the contents of an application schema in UML to RDF / Simple Knowledge Organization System (SKOS) / Web Ontology Language (OWL) elements are defined.

Ideas for new JSON Schema conversion rules include, but are not limited to:

- Rules to control the structure of JSON Schemas produced by the JSON Schema target, including the creation of definition schemas. For further details, see the section on Conversion of an application schema and its classes.
- Rules to convert inheritance (for further details, see the section on Conversion of generalization/inheritance).

- Rules to define the basic structures of a JSON object relevant for a GeoServices and a GeoJSON feature, such as:
  - the GeoJSON "type": "Feature" key-value pair
  - the feature "geometry"
  - the "attributes"/ "properties" key (under which all feature properties would be encoded, except geometry properties)
- A rule to add an "entityType" key, much like it is done in the current *geoservices* encoding rules. This key could be used to support value-dependent validation. It could also be used in a JSON-LD @context definition as source for the "@type" (for further details, see the section on JSON-LD).
- A rule to create a property to store the feature identifier. This would be relevant if application schema types do not define such a property themselves.
- A rule to define the conversion of a <<union>> class. An example of how this can be done using the keyword *oneOf* is given in Listing 5.
- A rule to control if *nilReason* properties shall be generated (for further details, see the OGC Testbed 9 SSI UGAS Engineering Report ([1]), section 6.2.3.3).

Chapter 6. Defining the semantics of JSON data through the use of JSON-LD

> **NOTE** The Enhancements for JSON Schema Conversion chapter focuses on enhancing the conversion from an application schema in UML to a JSON Schema. The primary concern in that chapter is how class-like entities and their characteristics (e.g. their properties) can be represented in JSON Schema. The focus of this chapter, however, shifts to instance/individual data.

6.1. Overview

Web applications commonly use JSON for exchanging information. As outlined in the introduction of this chapter, JSON is a simple encoding that itself lacks a number of key features that are often needed for interoperable information exchange. For applications that were purpose-built to communicate with each other, this is typically not an issue. However, that also means that the JSON data that is published by these applications typically cannot be used by other applications, simply because these other applications cannot clearly identify the meaning of the data.

Consider the example of Listing 10:

Listing 10. Tree example - information about two trees encoded in JSON

```
[
 {
  "art": "Eiche",
  "hoehe": 16,
  "eid": "08218adf-7947-4f28-bcaf-e069ef43e012",
  "alter": 242,
  "ort": {"wkt": "POINT(8.191035,51.899666)"}
 },
 {
  "art": "Walnuss",
  "hoehe": 10,
  "eid": "54e610a3-d317-4e20-85ea-31a56db8afe3",
  "alter": 33,
  "ort": {"wkt": "POINT(8.195380,51.903862)"}
 }
]
```

This example contains an array with two JSON objects. While a human reader can make assumptions regarding what this data is about - particularly if the reader knows the German language - an application will not be able to make sense of the data, unless it was built to consume this particular kind of JSON data.

NOTE In order to keep the complexity of examples in this section on a reasonable level, the JSON objects from the examples do not contain values in multiple languages. However, JSON-LD has some features for working with language-specific values of JSON keys. Please refer to the JSON-LD specification for further details (see string internatialization [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#string-internationalization], language indexing [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#language-indexing], and language maps [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#language-maps]).

A JSON-LD @context document like that in Listing 11 can be used to identify the meaning of each JSON object:

Listing 11. Tree example - JSON-LD @context defining the semantics of the JSON objects

```
{"@context": {
 "@base": "http://example.org/baumregister/",
 "@version": 1.1,
 "xsd": "http://www.w3.org/2001/XMLSchema#",
 "geosparql": "http://www.opengis.net/ont/geosparql#",
 "ex": "http://example.org/ontology/flora/",
 "ort": "geosparql:hasDefaultGeometry",
 "wkt": {
  "@id": "geosparql:asWKT",
  "@type": "geosparql:wktLiteral"
 },
 "eid": "@id",
 "art": "@type",
 "Eiche": "ex:oak",
 "Walnuss": "ex:walnut",
 "hoehe": {
  "@id": "ex:height",
  "@type": "xsd:double"
 },
 "alter": {
  "@id": "ex:age",
  "@type": "xsd:integer"
 }
}}
```

NOTE The JSON-LD @context example has @version: 1.1. That indicates that the example is based on a newer version of JSON-LD than the current W3C JSON-LD standard [http://www.w3.org/TR/2014/REC-json-ld-20140116/], which has version 1.0. This standard is currently being revised by W3C. The final community group report defining JSON-LD version 1.1 [https://www.w3.org/2018/jsonld-cg-reports/json-ld/] is available, and will be taken forward by the JSON-LD Working Group (the publication status and milestones are documented online [https://www.w3.org/2018/json-ld-wg/PublStatus]). The analysis of JSON-LD for defining the semantics of JSON data was performed based upon JSON-LD 1.1 (final community group report).

The JSON-LD @context allows defining the semantics of the JSON objects as well as their key-value pairs - which is the primary goal of this analysis. When the JSON-LD @context (see Listing 11) is applied to each of the two JSON objects (see Listing 10), the JSON data can also be transformed into Linked Data, formatted as RDF (for further details, see the JSON-LD chapter on Serializing/Deserializing RDF [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#serializing-deserializing-rdf]). The result of that transformation is documented in Table 2. Notice that some of the predicates in the result are from the OGC GeoSPARQL standard (OGC 11-052r4).

NOTE A JSON-LD @context can be provided with JSON data by adding a link header to the HTTP response that contains the data (for further details, see the JSON-LD specification [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#interpreting-json-as-json-ld]). This appears to be a suitable mechanism for semantically enabling existing JSON-based data services, while requiring only minimal changes to such existing services.

Table 2. Tree example - result of applying the JSON-LD @context and serializing as RDF statements

Subject	Predicate	Object	Datatype
_:b0	http://www.opengis.net/ont/geosparql#asWKT	POINT(8.191035,51.899666)	http://www.opengis.net/ont/geosparql#wktLiteral
_:b1	http://www.opengis.net/ont/geosparql#asWKT	POINT(8.195380,51.903862)	http://www.opengis.net/ont/geosparql#wktLiteral
http://example.org/baumregister/08218adf-7947-4f28-bcaf-e069ef43e012	http://example.org/ontology/flora/age	242	http://www.w3.org/2001/XMLSchema#integer
http://example.org/baumregister/08218adf-7947-4f28-bcaf-e069ef43e012	http://example.org/ontology/flora/height	1.6E1	http://www.w3.org/2001/XMLSchema#double
http://example.org/baumregister/08218adf-7947-4f28-bcaf-e069ef43e012	http://www.opengis.net/ont/geosparql#hasDefaultGeometry	_:b0	
http://example.org/baumregister/08218adf-7947-4f28-bcaf-e069ef43e012	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://example.org/ontology/flora/oak	

Subject	Predicate	Object	Datatype
http://example.org/baumregister/54e610a3-d317-4e20-85ea-31a56db8afe3	http://example.org/ontology/flora/age	33	http://www.w3.org/2001/XMLSchema#integer
http://example.org/baumregister/54e610a3-d317-4e20-85ea-31a56db8afe3	http://example.org/ontology/flora/height	1.0E1	http://www.w3.org/2001/XMLSchema#double
http://example.org/baumregister/54e610a3-d317-4e20-85ea-31a56db8afe3	http://www.opengis.net/ont/geosparql#hasDefaultGeometry	_:b1	
http://example.org/baumregister/54e610a3-d317-4e20-85ea-31a56db8afe3	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://example.org/ontology/flora/walnut	

Each row of Table 2 defines an RDF statement.

> *RDF allows us to make statements about resources. The format of these statements is simple. A statement always has the following structure: <subject> <predicate> <object>. […] An RDF statement expresses a relationship between two resources. The subject and the object represent the two resources being related; the predicate represents the nature of their relationship. The relationship is phrased in a directional way (from subject to object) and is called in RDF a property. Because RDF statements consist of three elements they are called triples. […] three types of RDF data […] occur in triples: IRIs, literals and blank nodes. […] The abbreviation IRI is short for "International Resource Identifier". An IRI identifies a resource. […] RDF is agnostic about what the IRI represents. However, IRIs may be given meaning by particular vocabularies or conventions. […] Literals are basic values that are not IRIs. Examples of literals include strings such as "La Joconde", dates such as "the 4th of July, 1990" and numbers such as "3.14159". Literals are associated with a datatype enabling such values to be parsed and interpreted correctly. […] IRIs and literals together provide the basic material for writing down RDF statements. In addition, it is sometimes handy to be able to talk about resources without bothering to use a global identifier. A resource without a global identifier […] can be represented in RDF by a blank node. Blank nodes are like simple variables in algebra; they represent some thing without saying what their value is. Blank nodes can appear in the subject and object position of a triple. They can be used to denote resources without explicitly naming them with an IRI.*
>
> — RDF 1.1 Primer [https://www.w3.org/TR/rdf11-primer/]

The HTTP URLs for subjects, predicates, and objects in the table are IRIs. The strings "_:b0" and "_:b1" are blank nodes.

The result of this set of RDF statements contains two RDF resources, one of type http://example.org/ontology/flora/oak, the other of type http://example.org/ontology/flora/walnut. For each resource, we have information about its age, height, and location (as a point geometry).

NOTE	IRIs like in the example typically represent concepts of, or expressed with, semantic web languages, such as RDFS vocabularies and OWL ontologies. These languages are used to define classes, their properties, relationships between them, and much more. This report cannot provide a full introduction to semantic web languages. However, the W3C has published useful documentation which helps the interested reader getting a better understanding of these languages: the RDF 1.1 Primer [https://www.w3.org/TR/rdf11-primer/] as well as the OWL 2 Primer [https://www.w3.org/TR/owl2-primer/].

The following table describes how each value within the JSON-LD @context from the tree example

(see Listing 11) is used. For a full description of JSON-LD keywords and structure, please refer to the official specification(s).

Table 3. Tree example - description of the JSON-LD @context

@context value	Description
"@base": "http://example.org/baumregister/"	Used to construct absolute IRIs from relative IRIs contained in the object identifier key (in the example: "eid"). If not set explicitly in the @context, the location of the JSON document would be used as base, which may not be desirable, particularly with respect to a possible change of the document URL in the future. If the JSON data already had an absolute IRI as identifier, @base would not be used/needed to augment the IRI.
"@version": 1.1	The JSON-LD version for which the @context document was created.
"xsd": "http://www.w3.org/2001/XMLSchema#", "geosparql": "http://www.opengis.net/ont/geosparql#", "ex": "http://example.org/ontology/flora/"	Define mappings of prefixes to IRIs. The prefixes are used by other terms, resulting in compact IRIs.
"ort": "geosparql:hasDefaultGeometry"	Maps the key "ort" to the compact IRI "geosparql:hasDefaultGeometry" (which expands to http://www.opengis.net/ont/geosparql#hasDefaultGeometry).
"eid": "@id"	Identifies the key "eid" as identifier for the object. Since in the example the "eid" does not contain an absolute IRI, the document base is used to construct an absolute IRI. That base is explicitly set in the @context using "@base". As result, we get the IRI http://example.org/baumregister/08218adf-7947-4f28-bcaf-e069ef43e012 as identifier of the first JSON object from the example.
"art": "@type"	Defines the key "art" to provide the type of the object.
"Eiche": "ex:oak", "Walnuss": "ex:walnut"	Maps the term "Eiche" to the compact IRI "ex:oak", and provides a similar mapping for the term "Walnuss". In the example, these mappings are used to explicitly map type identifiers ("Eiche", "Walnuss") to RDFS classes. The first object from the example thus has rdf:type (expressed via the predicate IRI http://www.w3.org/1999/02/22-rdf-syntax-ns#type) http://example.org/ontology/flora/oak.

@context value	Description
"wkt": { "@id": "geosparql:asWKT", "@type": "geosparql:wktLiteral" }, "hoehe": { "@id": "ex:height", "@type": "xsd:double" }, "alter": { "@id": "ex:age", "@type": "xsd:integer" }	Maps a key (e.g. "wkt") to a compact IRI (e.g. "geosparql:asWKT"), and identifies its value type (e.g. "geosparql:wktLiteral"). For keys that are mapped to properties whose value is a literal, the value type should be declared explicitly in the JSON-LD @context. That will allow a JSON-LD application to parse the value as this specific type, instead of as one of the set of general types that JSON supports (string, number, boolean).

A JSON-LD @context defines how to interpret JSON objects as Linked Data. In this example, the JSON data was mapped to a particular example vocabulary. Other JSON-LD @context documents could be used to apply a mapping to different vocabularies. In any case, the resulting data can now be used by semantic and Linked Data applications. The information that was previously exclusively used by the applications that exchanged the original JSON data can now be shared in a wider context, allowing combination with other data to acquire new, useful information, and opening up the possibility to unlock new applications and services.

The following figures illustrate this concept. Figure 2 shows the current situation, where NAS instance data is available in different JSON formats that only specific applications understand and consume.

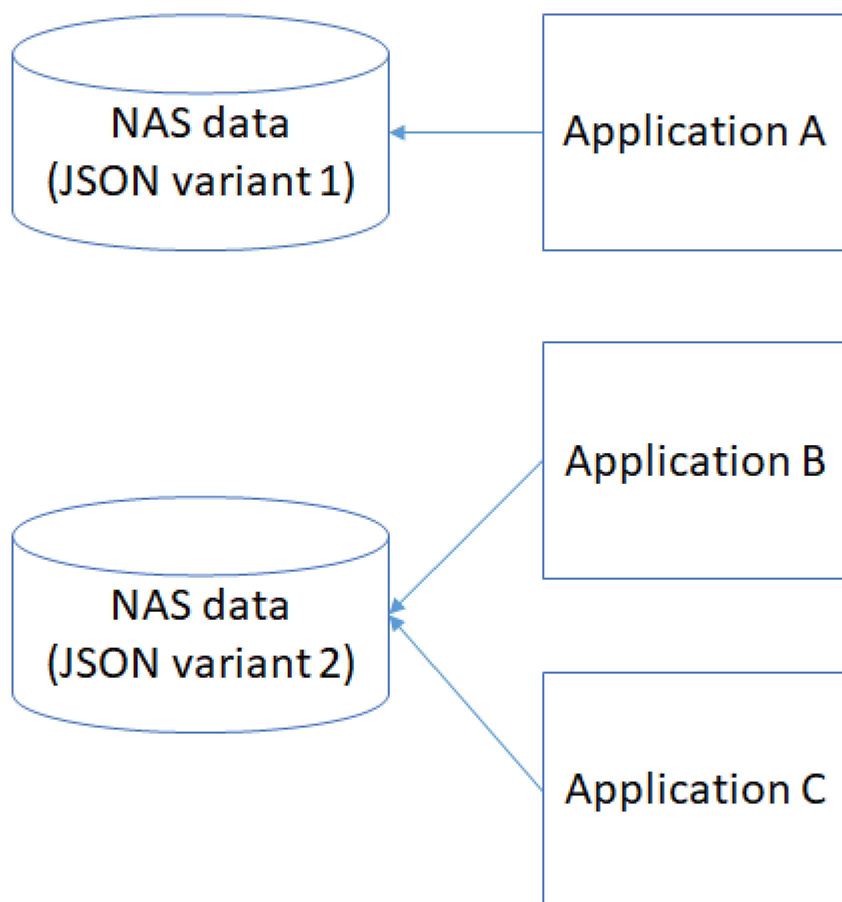

Figure 2. JSON-LD use cases - current situation: application specific JSON formats

When JSON-LD @context documents are defined for these JSON formats, to define the semantics, then new applications are enabled (see Figure 3).

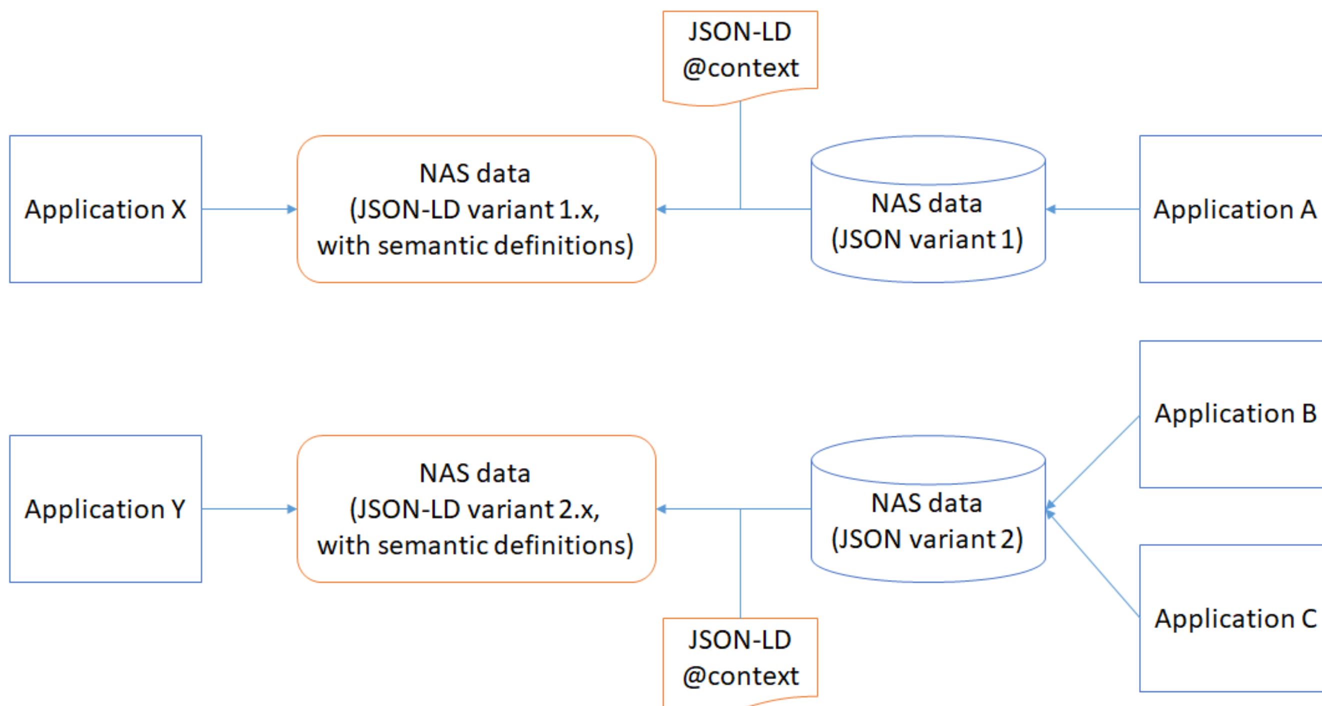

Figure 3. JSON-LD use cases - JSON data converted to JSON-LD data using JSON-LD @context documents

In an ideal situation, the @context documents support mapping of the JSON data in different formats to a common, NEO compliant RDF representation (see Figure 4). This approach would establish a solution for the "babylonian confusion" caused by the different JSON formats (the number of which may grow over time, as more projects add their specific "flavor" of JSON to the mix). The RDF data could then be added to a single RDF data store. Multiple applications could use this data store as their main data source.

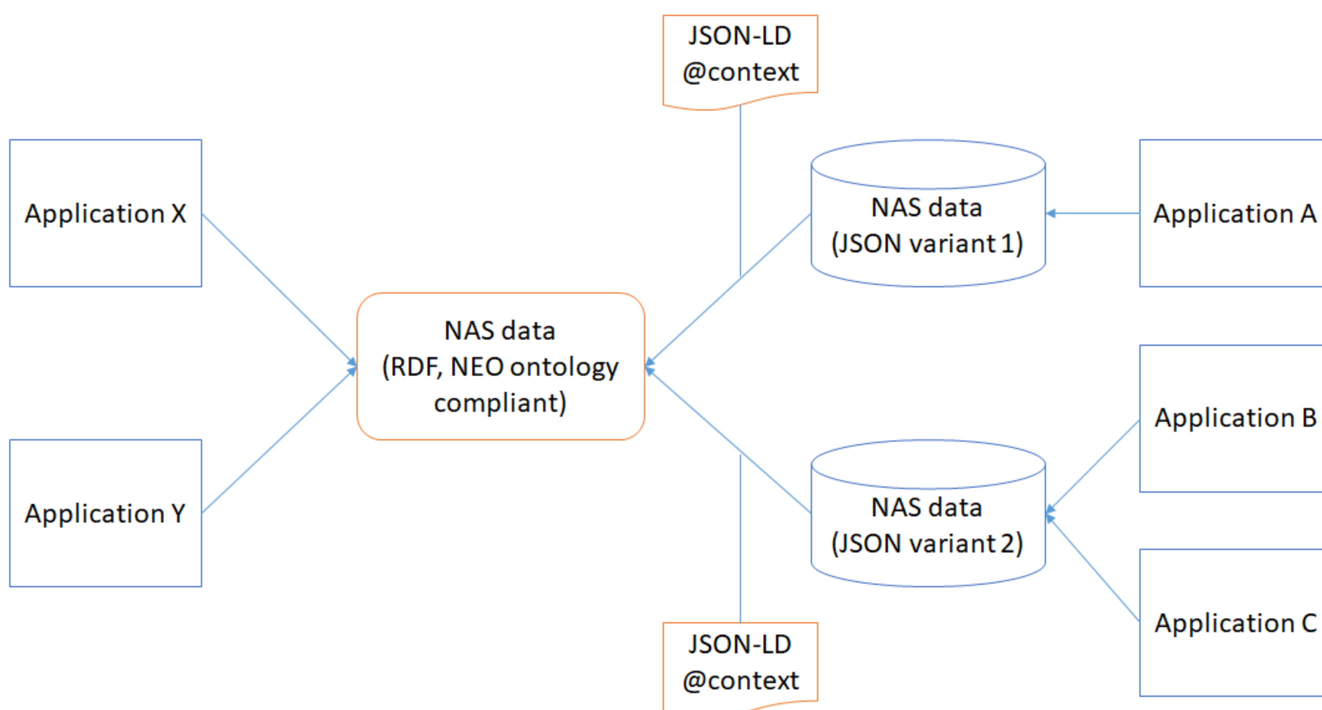

Figure 4. JSON-LD use cases - JSON data converted to NEO RDF data using JSON-LD @context documents

The following sections document the results of the JSON-LD analysis performed in Testbed-14. They show how the aforementioned use cases can be realized using JSON-LD, but they also identify issues as well as restrictions of this approach.

6.2. Converting GeoJSON data to NEO RDF data

6.2.1. Developing a JSON-LD @context

The previous section illustrates the conversion of JSON data to RDF using JSON-LD with a simple example (see Listing 10 and Listing 11). To discover potential problems and issues when performing such a conversion, a more extensive example is necessary. The GeoJSON feature collection specified in Listing 12 provides such an example.

NOTE The purpose of this example is to analyze how the semantics of (Geo)JSON can be defined, as well as if and how the data can be converted to RDF - using JSON-LD. Without the restriction to JSON-LD, any kind of purpose-built transformation could be created to achieve the conversion to RDF in each (unique) case in which such a conversion is desired. However, the intent is to identify potential issues of the conversion to RDF when it is performed with a standardized approach based on the use of JSON-LD.

Listing 12. GeoJSON example - feature collection containing aeronautical feature data

```
{
 "type": "FeatureCollection",
 "name": "Aeronautic",
 "crs": {
  "type": "name",
  "properties": {"name": "urn:ogc:def:crs:OGC:1.3:CRS84"}
 },
 "features": [
  {
   "type": "Feature",
   "properties": {
    "FCSUBTYPE": 100441,
    "F_CODE": "GB030",
    "ADR": "noInformation",
    "APT": 1,
    "APT2": -999999,
    "APT3": -999999,
    "ARA": 150,
    "CAA": 16,
    "LZN": 50,
    "UFI": "588fb71e-5965-11e4-863e-7845c4f8683b",
    "ZI005_FNA": "Metropolitan Hospital",
    "ZI020_GE4": "ge:GENC:3:1-2:USA",
    "ZI026_CTUC": 5,
    "ZI026_CTUL": 1,
    "ZI026_CTUU": 200000,
    "ZSAX_RS0": "U",
    "GLOBALID": "{64599E4D-7D68-4CBC-8DA1-E57EC0BC6DA0}"
   },
   "geometry": {
```

```
      "type": "Point",
      "coordinates": [
       -74.0059731,
       40.7143528,
       57.293799999999
      ]
     }
    },
    {
     "type": "Feature",
     "properties": {
      "FCSUBTYPE": 100454,
      "F_CODE": "GB075",
      "ARA": -999999,
      "AXS": 2,
      "LZN": 165,
      "UFI": "94f2deb1-a88c-11e4-b9c7-7845c4f86835",
      "WID": 12,
      "ZI005_FNA": "No Information",
      "ZI020_GE4": "ge:GENC:3:1-2:USA",
      "ZI026_CTUC": 5,
      "ZI026_CTUL": 1,
      "ZI026_CTUU": 25000,
      "ZSAX_RS0": "U",
      "GLOBALID": "{1CC4F529-8463-469F-A222-4EC4449A1348}",
      "SHAPE_Length": 0.0018287501057487
     },
     "geometry": {
      "type": "MultiLineString",
          "coordinates": [[[-74.532791552,40.791092272,-50000],[-74.532613521
,40.791014823,-50000],[-74.532477231,40.790955144,-50000],[-74.53220331,40.79083545,
-50000],[-74.531973982,40.79073537,-50000],[-74.531327737,40.790453403,-50000],[
-74.531115507,40.790360699,-50000]]]
     }
    }
  ]
}
```

The feature collection contains two aeronautical features, with their properties and geometries. A task in OGC Testbed-14 was to develop a JSON-LD @context document with which this data can be transformed into NEO compliant RDF data. The first step was to identify which information can or cannot be mapped:

- The example contains a feature collection. The primary interest is in the actual features. Information about the collection itself, like its name, is irrelevant.
- A GeoJSON feature has a "type", "properties", and a "geometry".
  - The GeoJSON "type" is irrelevant. It contains generic information that results from the GeoJSON format.
  - The "properties" key is used for nesting the actual feature properties. As such, the

"properties" key is irrelevant, but the key/value pairs contained in its value are of interest.

- The "geometry" is of interest. However, JSON-LD cannot convert a multi-dimensional array (NOTE: this restriction has led to a feature request for JSON-LD, which is documented online [https://github.com/w3c/json-ld-syntax/issues/7]). This issue has been investigated in OGC Testbed 11. The engineering report Implementing JSON/GeoJSON in an OGC Standard [https://portal.opengeospatial.org/files/?artifact_id=64595] ([8]), section 7.2., describes the issue in more detail. A reasonable solution for the issue is presented in chapter 7.5 of the ER: handle GeoJSON geometries by converting them to Well Known Text (WKT) as defined by ISO 19125-1, which is one of the two serializations supported by OGC GeoSPARQL. Listing 13 shows what the feature collection would look like after such a transformation. When transforming a GeoJSON geometry to a JSON object that contains WKT, some details need to be considered:
    - The transformation of the geometry to a WKT literal as defined by GeoSPARQL can include the CRS (for details, see GeoSPARQL, section 8.5.1). GeoJSON as specified by IETF 7946 [https://tools.ietf.org/html/rfc7946] requires the CRS to be "urn:ogc:def:crs:OGC:1.3:CRS84" (that the example in Listing 12 still declares the "crs" shows that it has very likely been developed based upon an older version of GeoJSON). This CRS is defined by GeoSPARQL as its default CRS. So when transforming the geometries from the example, we can ignore the CRS.
    - The transformation of the GeoJSON geometry to WKT should result in a JSON object structure that is suitable for conversion to a GeoSPARQL geometry property, with the geometry value having a WKT serialization. The key on the first level of the JSON object should identify the geometry property. This can be geosparql:hasGeometry, or a subproperty thereof. Since the rdfs:range of that property is defined as geosparql:Geometry, a reasoner - a piece of software able to infer logical consequences from a set of asserted facts or axioms - can infer that the JSON object on the second level is a (subtype of) geosparql:Geometry, even if the JSON-LD @context does not declare a specific type for that object. However, the transformation of the GeoJSON geometry could declare the node type, allowing a specific conversion of the JSON object on the second level to, for example, http://www.opengis.net/ont/sf#Point or http://www.opengis.net/ont/sf#MultiLineString. The second level object should contain a key that does not occur in the GeoJSON feature properties. That key will be mapped to http://www.opengis.net/ont/geosparql#asWKT, and the value of the key will be the WKT literal. The result of the transformation will be a valid representation of a GeoSPARQL geometry property (and its value), which can be used for spatial computations in SPARQL queries (of RDF triple stores).

- Some of the feature properties in the example have been identified as being an ESRI database artifact, or as no longer being used by NAS and NEO (at least not in the form shown in the example). That applies to the keys "FCSUBTYPE", "GLOBALID", and "SHAPE_Length" under all circumstances, and "ADR" when used with "F_CODE" = "GB030". These keys are therefore also irrelevant.

Listing 13 shows the result of transforming the geometry of each GeoJSON feature, as described before:

Listing 13. GeoJSON example - result of transforming the GeoJSON geometry of the aeronautical features

```
{
```

```
"type": "FeatureCollection",
"name": "Aeronautic",
"crs": {
 "type": "name",
 "properties": {"name": "urn:ogc:def:crs:OGC:1.3:CRS84"}
},
"features": [
 {
  "type": "Feature",
  "properties": {
   "FCSUBTYPE": 100441,
   "F_CODE": "GB030",
   "ADR": "noInformation",
   "APT": 1,
   "APT2": -999999,
   "APT3": -999999,
   "ARA": 150,
   "CAA": 16,
   "LZN": 50,
   "UFI": "588fb71e-5965-11e4-863e-7845c4f8683b",
   "ZI005_FNA": "Metropolitan Hospital",
   "ZI020_GE4": "ge:GENC:3:1-2:USA",
   "ZI026_CTUC": 5,
   "ZI026_CTUL": 1,
   "ZI026_CTUU": 200000,
   "ZSAX_RS0": "U",
   "GLOBALID": "{64599E4D-7D68-4CBC-8DA1-E57EC0BC6DA0}"
  },
  "geometry": {
   "@type": "sf:Point",
   "asWKT": "POINT(-74.0059731, 40.7143528, 57.293799999999)"
  }
 },
 {
  "type": "Feature",
  "properties": {
   "FCSUBTYPE": 100454,
   "F_CODE": "GB075",
   "ARA": -999999,
   "AXS": 2,
   "LZN": 165,
   "UFI": "94f2deb1-a88c-11e4-b9c7-7845c4f86835",
   "WID": 12,
   "ZI005_FNA": "No Information",
   "ZI020_GE4": "ge:GENC:3:1-2:USA",
   "ZI026_CTUC": 5,
   "ZI026_CTUL": 1,
   "ZI026_CTUU": 25000,
   "ZSAX_RS0": "U",
   "GLOBALID": "{1CC4F529-8463-469F-A222-4EC4449A1348}",
   "SHAPE_Length": 0.0018287501057487
```

```
    },
    "geometry": {
     "@type": "sf:MultiLineString",
     "asWKT": "MULTILINESTRING((-74.532791552 40.791092272 -50000, -74.532613521
40.791014823 -50000, -74.532477231 40.790955144 -50000, -74.53220331 40.79083545
-50000, -74.531973982 40.79073537 -50000, -74.531327737 40.790453403 -50000,
-74.531115507 40.790360699 -50000))"
    }
   }
  ]
}
```

The JSON-LD @context document specified in Listing 14 could be used to convert the transformed JSON data from Listing 13 to RDF data.

Listing 14. GeoJSON example - JSON-LD @context defining the semantics of the aeronautical features based on the NEO

```
{
 "@context": {
  "@version": 1.1,
  "ic": "http://api.nsgreg.nga.mil/ontology/ic/ism/V13/ISM-Public#",
  "geojson": "https://purl.org/geojson/vocab#",
  "geosparql": "http://www.opengis.net/ont/geosparql#",
  "neo": "http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#",
  "sf": "http://www.opengis.net/ont/sf#",
  "xsd": "http://www.w3.org/2001/XMLSchema#",
  "@base": "http://example.org/base/neo/",
  "UFI": "@id",
  "features": "geojson:features",
  "properties": "@nest",
  "F_CODE": "@type",
  "GB030": "neo:Helipad",
  "GB075": "neo:Taxiway",
  "APT": {
   "@id": "neo:LandAerodrome.airfieldUse",
   "@type": "xsd:integer"
  },
  "APT2": {
   "@id": "neo:LandAerodrome.airfieldUse",
   "@type": "xsd:integer"
  },
  "APT3": {
   "@id": "neo:LandAerodrome.airfieldUse",
   "@type": "xsd:integer"
  },
  "ARA": {
   "@id": "neo:AerodromeMoveArea.featureArea",
   "@type": "xsd:integer"
  },
  "AXS": {
```

```
    "@id": "neo:AerodromeMoveArea.aerodromeSurfaceStatus",
    "@type": "xsd:integer"
   },
   "CAA": {
    "@id": "neo:Helipad.controllingAuthority",
    "@type": "xsd:integer"
   },
   "LZN": {
    "@id": "neo:AerodromeMoveArea.featureLength",
    "@type": "xsd:integer"
   },
   "WID": {
    "@id": "neo:AerodromeMoveArea.featureWidth",
    "@type": "xsd:integer"
   },
   "ZI005_FNA": {
    "@id": "neo:GeoNameInfo.fullName",
    "@type": "xsd:string"
   },
   "ZI020_GE4": {
    "@id": "neo:GeopoliticalEntityDesig.gencShortUrnBasedIdentifier",
    "@type": "xsd:string"
   },
   "ZI026_CTUC": {
    "@id": "neo:IntegerInterval.intervalClosureType",
    "@type": "xsd:integer"
   },
   "ZI026_CTUL": {
    "@id": "neo:IntegerInterval.lowerValue",
    "@type": "xsd:integer"
   },
   "ZI026_CTUU": {
    "@id": "neo:IntegerInterval.upperValue",
    "@type": "xsd:integer"
   },
   "ZSAX_RS0": {
    "@id": "ic:SecurityAttributesGroupType.resClassification",
    "@type": "xsd:string"
   },
   "geometry": "neo:FeatureEntity.place",
   "asWKT": {
    "@id": "geosparql:asWKT",
    "@type": "geosparql:wktLiteral"
   }
  }
 }
```

The following table describes how the values within the JSON-LD @context document (see Listing 14) are used, similar to how it is done for the JSON-LD @context from the example in the overview section (see Listing 11). As noted in that section, please refer to the official specification(s) for a full

description of JSON-LD keywords and structure.

Table 4. GeoJSON example - description of the JSON-LD @context

@context value	Description
"@version": 1.1	The JSON-LD version for which the @context document was created. This example actually uses new features that JSON-LD 1.0 does not support.
"ic": "http://api.nsgreg.nga.mil/ontology/ic/ism/V13/ISM-Public#", "geojson": "https://purl.org/geojson/vocab#", "geosparql": "http://www.opengis.net/ont/geosparql#", "neo": "http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#", "sf": "http://www.opengis.net/ont/sf#", "xsd": "http://www.w3.org/2001/XMLSchema#"	Define mappings of prefixes to IRIs. The prefixes are used by other terms, resulting in compact IRIs.
"@base": "http://example.org/base/neo/"	Used to construct absolute IRIs from relative IRIs contained in the object identifier key (here: "UFI"). If not set explicitly in the @context, the location of the JSON document would be used as base, which may not be desirable, particularly with respect to a possible change of the document URL in the future. If the JSON data already had an absolute IRI as identifier, @base would not be used/needed to augment the IRI.
"UFI": "@id"	Identifies the key "UFI" as identifier for the object. Since in the example the "UFI" does not contain an absolute IRI, the document base is used to construct an absolute IRI. That base is explicitly set in the @context using "@base". As result, we get the IRI http://example.org/base/neo/588fb71e-5965-11e4-863e-7845c4f8683b as identifier of the first GeoJSON feature shown in Listing 13.
"features": "geojson:features"	In order for the objects within the feature collection to be recognized by the JSON-LD processor, the key "features" from the feature collection object must be mapped to a term. An ontology of GeoJSON terms is available at https://purl.org/geojson/vocab, and appears to be a suitable choice for the mapping.

@context value	Description
"properties": "@nest"	The GeoJSON format results in a "properties" key that contains a JSON object with all feature properties (except its geometry). This level of nesting is not desired for mapping to NEO. The JSON-LD 1.1 keyword @nest is used to instruct the JSON-LD parser to ignore the nesting created by "properties", and process the content as if it were declared directly within the containing object.
"F_CODE": "@type", "GB030": "neo:Helipad", "GB075": "neo:Taxiway"	Defines the key "F_CODE" to provide the type of the object. The @context defines mappings of the values of this key to compact IRIs, resulting in the desired mapping to IRIs of NEO classes.
"APT": { "@id": "neo:LandAerodrome.airfieldUse", "@type": "xsd:integer" }, …	Mappings of keys to NEO properties, also defining the value type. Multiple keys can be mapped to the same property, which can make sense in case that the original JSON data has an artificial duplication of feature properties, to represent multiple values of that property (which appears to be the case for the keys "APT", "APT2", and "APT3").
"geometry": "neo:FeatureEntity.place"	Maps the key "geometry" to the compact IRI "neo:FeatureEntity.place". The key is a result of transforming the original GeoJSON geometry (that result is shown in Listing 13).
"asWKT": { "@id": "geosparql:asWKT", "@type": "geosparql:wktLiteral" }	Maps the key "asWKT" to the compact IRI "geosparql:asWKT", and identifies its data type as "geosparql:wktLiteral".

The result of applying the JSON-LD @context from Listing 14 to the (transformed) JSON data from the example (see Listing 13), and serializing as RDF/XML, is shown in Listing 15:

Listing 15. GeoJSON example - result of applying the JSON-LD @context and serializing as RDF

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:geo="http://www.opengis.net/ont/geosparql#"
 xmlns:geojson="https://purl.org/geojson/vocab#"
 xmlns:ism="http://api.nsgreg.nga.mil/ontology/ic/ism/V13/ISM-Public#"
 xmlns:neo="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <!-- =============== -->
 <!-- Helipad feature -->
 <!-- =============== -->
 <rdf:Description rdf:about="http://example.org/base/neo/588fb71e-5965-11e4-863e-
7845c4f8683b">
  <rdf:type rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#Helipad"/>
  <neo:LandAerodrome.airfieldUse rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >1</neo:LandAerodrome.airfieldUse>
```

```
  <neo:LandAerodrome.airfieldUse rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >-999999</neo:LandAerodrome.airfieldUse>
  <neo:AerodromeMoveArea.featureArea rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >150</neo:AerodromeMoveArea.featureArea>
  <neo:Helipad.controllingAuthority rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >16</neo:Helipad.controllingAuthority>
  <neo:AerodromeMoveArea.featureLength rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >50</neo:AerodromeMoveArea.featureLength>
  <neo:GeoNameInfo.fullName>No Information</neo:GeoNameInfo.fullName>
  <neo:GeopoliticalEntityDesig.gencShortUrnBasedIdentifier>ge:GENC:3:1-
2:USA</neo:GeopoliticalEntityDesig.gencShortUrnBasedIdentifier>
  <neo:IntegerInterval.upperValue rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >200000</neo:IntegerInterval.upperValue>
  <neo:IntegerInterval.intervalClosureType rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >5</neo:IntegerInterval.intervalClosureType>
  <neo:IntegerInterval.lowerValue rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >1</neo:IntegerInterval.lowerValue>
  <ism:SecurityAttributesGroupType.resClassification>
U</ism:SecurityAttributesGroupType.resClassification>
  <neo:FeatureEntity.place rdf:nodeID="N536254f6b4a4435f8fe88e2b01f48eaf"/>
 </rdf:Description>
 <!-- =============== -->
 <!-- Taxiway feature -->
 <!-- =============== -->
 <rdf:Description rdf:about="http://example.org/base/neo/94f2deb1-a88c-11e4-b9c7-
7845c4f86835">
  <rdf:type rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#Taxiway"/>
  <neo:AerodromeMoveArea.featureArea rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >-999999</neo:AerodromeMoveArea.featureArea>
  <neo:AerodromeMoveArea.aerodromeSurfaceStatus
   rdf:datatype="http://www.w3.org/2001/XMLSchema#integer"
   >2</neo:AerodromeMoveArea.aerodromeSurfaceStatus>
  <neo:AerodromeMoveArea.featureLength rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >165</neo:AerodromeMoveArea.featureLength>
  <neo:AerodromeMoveArea.featureWidth rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >12</neo:AerodromeMoveArea.featureWidth>
  <neo:GeoNameInfo.fullName>No Information</neo:GeoNameInfo.fullName>
  <neo:GeopoliticalEntityDesig.gencShortUrnBasedIdentifier>ge:GENC:3:1-
2:USA</neo:GeopoliticalEntityDesig.gencShortUrnBasedIdentifier>
  <ism:SecurityAttributesGroupType.resClassification>
U</ism:SecurityAttributesGroupType.resClassification>
```

```
  <neo:IntegerInterval.intervalClosureType rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >5</neo:IntegerInterval.intervalClosureType>
  <neo:IntegerInterval.lowerValue rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >1</neo:IntegerInterval.lowerValue>
  <neo:IntegerInterval.upperValue rdf:datatype=
"http://www.w3.org/2001/XMLSchema#integer"
   >25000</neo:IntegerInterval.upperValue>
  <neo:FeatureEntity.place rdf:nodeID="Nbdfa3d3e55f740268e2b4a883e8c25ea"/>
 </rdf:Description>
 <!-- ========================== -->
 <!-- GeoJSON feature collection -->
 <!-- ========================== -->
 <rdf:Description rdf:nodeID="N76d568a797d44c3eb4fa47c78bcd54f5">
  <geojson:features rdf:resource="http://example.org/base/neo/94f2deb1-a88c-11e4-b9c7-
7845c4f86835"/>
  <geojson:features rdf:resource="http://example.org/base/neo/588fb71e-5965-11e4-863e-
7845c4f8683b"
  />
 </rdf:Description>
 <!-- ================ -->
 <!-- Helipad geometry -->
 <!-- ================ -->
 <rdf:Description rdf:nodeID="N536254f6b4a4435f8fe88e2b01f48eaf">
  <rdf:type rdf:resource="http://www.opengis.net/ont/sf#Point"/>
  <geo:asWKT rdf:datatype="http://www.opengis.net/ont/geosparql#wktLiteral">POINT(-
74.0059731,
   40.7143528, 57.293799999999)</geo:asWKT>
 </rdf:Description>
 <!-- ================ -->
 <!-- Taxiway geometry -->
 <!-- ================ -->
 <rdf:Description rdf:nodeID="Nbdfa3d3e55f740268e2b4a883e8c25ea">
  <rdf:type rdf:resource="http://www.opengis.net/ont/sf#MultiLineString"/>
  <geo:asWKT rdf:datatype="http://www.opengis.net/ont/geosparql#wktLiteral"
   >MULTILINESTRING((-74.532791552 40.791092272 -50000, -74.532613521 40.791014823
-50000,
   -74.532477231 40.790955144 -50000, -74.53220331 40.79083545 -50000, -74.531973982
40.79073537
   -50000, -74.531327737 40.790453403 -50000, -74.531115507 40.790360699 -
50000))</geo:asWKT>
 </rdf:Description>
</rdf:RDF>
```

The resulting RDF contains the following resources:

- the helipad and taxiway features
- the GeoJSON feature collection (represented by a blank node)
- the helipad and taxiway geometries (represented by blank nodes)

The resource that represents the GeoJSON feature collection can typically be ignored, since it is an artifact of the conversion from the original JSON data to RDF. Of primary interest are the two features, their properties, and their geometries.

On first glance, the result looks suitable. However, a number of issues have been identified, which need further consideration.

6.2.2. Identified issues

6.2.2.1. Mismatch between simple JSON structure and complex NEO structure

The GeoJSON features from the example in Listing 12 (a feature collection containing aeronautical data) have a simple structure. The values of the GeoJSON feature properties are either strings or numbers. Only the value of the GeoJSON feature geometry is a JSON object.

NOTE	In general, a GeoJSON feature can have properties with complex values (i.e. values that are JSON objects).

In contrast, the structure of the NEO classes to which the two GeoJSON features are mapped is much more complex. The annotated UML diagram in Figure 5 containing NAS elements that are relevant for the GeoJSON example illustrates this.

NOTE	As described on https://nsgreg.nga.mil/neo/, the NEO is derived from the NAS. The NEO encoding represents most of the UML classes as OWL classes. The UML attributes and association roles are represented as OWL properties. Therefore, using the NAS to provide a general overview of the complex structure of the NEO is appropriate. For details about the NEO encoding rule, please refer to the OGC Testbed-12 ShapeChange Engineering Report [http://docs.opengeospatial.org/per/16-020.html#rdf_NAS_encoding_rule].

NOTE	Figure 5 provides a significantly simplified view of the properties of the depicted classes. Attributes and associations that are irrelevant for the example are hidden. However, some associations have been kept in order to visualize the relationships between the classes. Furthermore, some of the value types of attributes are truncated, to avoid unnecessarily wide classes in the diagram.

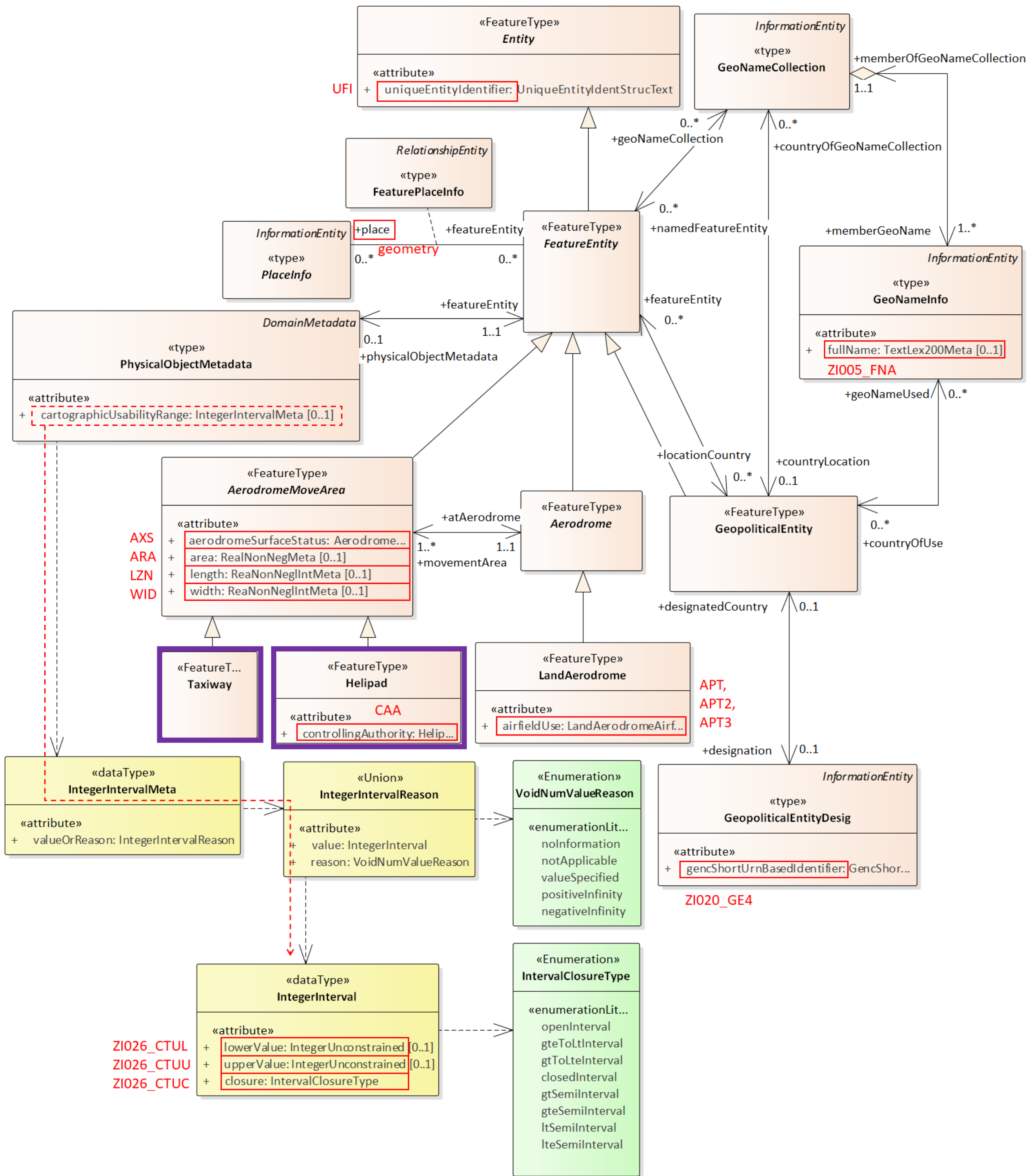

Figure 5. NAS elements relevant for conversion of aeronautical features from GeoJSON example

The feature types *Taxiway* and *Helipad* are highlighted with a purple border. The diagram shows that these feature types are part of a generalization hierarchy. As such, they inherit a set of properties from supertypes, for example the *aerodromeSurfaceStatus*, which is the conceptual basis for the NEO object property http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#AerodromeMoveArea.aerodromeSurfaceStatus, to which the JSON key "AXS" is mapped by the JSON-LD @context in Listing 14. The diagram also shows that some of the JSON keys are mapped to properties of related classes. For example, the JSON keys "APT", "APT2", and "APT3" are mapped to http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#LandAerodrome.airfieldUse, which has been derived from property *airfieldUse* of the conceptual class *LandAerodrome*. That class is not a supertype of

either *Taxiway* or *Helipad*. Instead, these feature types are related through an association that exists between their supertypes, *AerodromeMoveArea* and *Aerodrome*. The diagram also shows the existence of an association class between the types *PlaceInfo* and *FeatureEntity*. Finally, note that the keys "ZI026_CTUL", "ZI026_CTUU", and "ZI026_CTUC" are mapped to the properties *lowerValue*, *upperValue*, and *closure* of the general data type *IntegerInterval*.

Apparently, the properties of the GeoJSON features from the example in Listing 12 are mapped to properties of a wide range of classes in the NEO (which, again, is derived from the NAS). These properties do not only belong to superclasses of the two classes to which the GeoJSON features are mapped, but also to related classes. In the NEO, most properties belong to a particular domain and have a specific range. For all RDF properties contained in the RDF data that was converted from GeoJSON, the domain and/or range do not match the definition of the NEO property!

For the properties of superclasses of Taxiway and Helipad, the domain is correct. For all other properties, it is incorrect.

For example, the domain of LandAerodrome.airfieldUse is LandAerodrome (see Listing 16). Note that in the NEO, LandAerodrome is disjoint with Helipad and Taxiway (since their superclasses Aerodrome and AerodromeMoveArea are declared by NEO as being disjoint). Thus, the RDF data converted from the GeoJSON data is inconsistent with the NEO.

Listing 16. Definition of NEO property LandAerodrome.airfieldUse

```
<owl:ObjectProperty rdf:about="http://api.nsgreg.nga.mil/ontology/neo-ent/1-
7#LandAerodrome.airfieldUse">
    <rdfs:range rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-
7#LandAerodromeAirfieldUseCodeMeta"/>
    <rdfs:domain rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-
7#LandAerodrome"/>
    <rdfs:label xml:lang="en">LandAerodrome.airfieldUse</rdfs:label>
    <skos:prefLabel xml:lang="en">Land Aerodrome : Airfield Use</skos:prefLabel>
    <rdfs:isDefinedBy rdf:resource="http://nsgreg.nga.mil/as/view?i=106352"/>
    <skos:definition xml:lang="en">Definition: A primary use of an airfield.
Description: [None Specified]</skos:definition>
</owl:ObjectProperty>
```

This example also shows that the type of the converted RDF property (xsd:integer) is inconsistent with the range as defined by the NEO. In the NEO, the range is a class (neo:LandAerodromeAirfieldUseCodeMeta), rather than a data type.

6.2.2.2. Numeric code values

The NEO represents enumerations and code lists as OWL classes, which have a restricted set of individuals that represent the allowed code values. An example of this representation is given in Listing 17, for the enumeration VoidValueReason.

NOTE	The RDF representation in Listing 17 has been shortened (wherever you see '…'). The full representation of neo:VoidValueReason can be accessed at http://api.nsgreg.nga.mil/ontology/neo-enum/1-7/VoidValueReason.

Listing 17. Definition of enumeration VoidValueReason in the NEO

```
<rdf:RDF
    xmlns:dct="http://purl.org/dc/terms/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:skos="http://www.w3.org/2004/02/skos/core#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:e="http://api.nsgreg.nga.mil/ontology/neo-enum/1-7/"
    xmlns:neox="http://api.nsgreg.nga.mil/ontology/neox/1.0/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xml:base="http://api.nsgreg.nga.mil/ontology/neo-enum/1-7">
  <owl:Class rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason">
    <owl:oneOf rdf:parseType="Collection">
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/noInformation">
        <skos:topConceptOf>
          <skos:ConceptScheme rdf:about="http://api.nsgreg.nga.mil/ontology/neo-
enum/1-7/VoidValueReason_ConceptScheme">
            <!-- ... -->
          </skos:ConceptScheme>
        </skos:topConceptOf>
        <skos:inScheme rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason_ConceptScheme"/>
        <rdfs:label xml:lang="en">noInformation</rdfs:label>
        <skos:prefLabel xml:lang="en">No Information (Void Value
Reason)</skos:prefLabel>
        <rdfs:isDefinedBy rdf:resource="http://nsgreg.nga.mil/as/view?i=132011"/>
        <skos:definition xml:lang="en">...</skos:definition>
      </e:VoidValueReason>
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/notApplicable">
        <skos:topConceptOf rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-
enum/1-7/VoidValueReason_ConceptScheme"/>
        <skos:inScheme rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason_ConceptScheme"/>
        <rdfs:label xml:lang="en">notApplicable</rdfs:label>
        <!-- ... -->
      </e:VoidValueReason>
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/other">
        <!-- ... -->
      </e:VoidValueReason>
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/valueSpecified">
        <!-- ... -->
      </e:VoidValueReason>
    </owl:oneOf>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2004/02/skos/core#Concept"/>
    <skos:prefLabel xml:lang="en">Void Value Reason</skos:prefLabel>
```

```
    <rdfs:isDefinedBy rdf:resource="http://nsgreg.nga.mil/as/view?i=100940"/>
    <skos:definition xml:lang="en">Definition: The condition due to which the
attribute value may be missing or otherwise not fulfil the specification of the
attribute value domain.  Description: For example, it may be the case that the
attribute value is unknown or that it is known but due to policy considerations it
cannot be given.</skos:definition>
    <rdfs:label xml:lang="en">VoidValueReason</rdfs:label>
    <dct:isPartOf rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-enum/1-7"/>
  </owl:Class>
  <owl:AllDifferent>
    <owl:distinctMembers rdf:parseType="Collection">
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/noInformation"/>
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/notApplicable"/>
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/other"/>
      <e:VoidValueReason rdf:about="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidValueReason/valueSpecified"/>
    </owl:distinctMembers>
  </owl:AllDifferent>
</rdf:RDF>
```

As illustrated, a code is represented in the NEO as a resource. For example, the code "noInformation" is represented by the resource with IRI http://api.nsgreg.nga.mil/ontology/neo-enum/1-7/VoidValueReason/noInformation.

NOTE	One can actually retrieve the RDF representation of the resource from that URL.

However, in the GeoJSON example (Listing 12), that code is given as a primitive value: the strings "No information" and "noInformation", or the number -999999. JSON-LD supports expanding a specific string value to an IRI.

To do so, use a JSON-LD @context like the following:

```
{
 "@context": {
  "ex": "http://example.org/my/namespace/",
  "key_string_value": "ex:Resource/code",
  "json_key": {
   "@id": "ex:Property",
   "@type": "@vocab"
  }
 },
 "json_key": "key_string_value"
}
```

NOTE

With a scoped context (a new feature of JSON-LD 1.1) one can even expand the same string value to different IRIs:

```
{
 "@context": {
  "@version": 1.1,
  "ex": "http://example.org/my/namespace/",
  "key_string_value": "ex:Resource/codeA",
  "json_key_1": {
   "@id": "ex:PropertyA",
   "@type": "@vocab"
  },
  "json_key_2": {
   "@id": "ex:PropertyB",
   "@type": "@vocab",
   "@context": {"key_string_value": "ex:Resource/codeB"}
  }
 },
 "json_key_1": "key_string_value",
 "json_key_2": "key_string_value"
}
```

These examples can be tested on the JSON-LD development playground [https://json-ld.org/playground-dev/].

However, numbers are not allowed as JSON keys! Thus, numeric values - like -999999 - cannot be mapped to NEO codes (which need to be represented by IRIs).

NOTE This is a restriction of JSON-LD.

6.2.2.3. NAS/NEO value or reason pattern

In the NAS and the NEO, a (feature attribute) property either has a value or a reason for the

absence of the value. In the NAS, this is modelled as a <<union>> - see for example the IntegerIntervalReason class in Figure 5. When deriving the NEO from the NAS, these unions are transformed, and the two properties (value(s) and reason) end up being part of the corresponding XxxMeta class. Listing 18 and Listing 19 illustrate this for IntegerIntervalMeta.value and -.reason.

Listing 18. Definition of property IntegerInterval.value in the NEO

```
<?xml version="1.0" encoding="UTF-8"?>
<owl:ObjectProperty xmlns:skos="http://www.w3.org/2004/02/skos/core#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:owl=
"http://www.w3.org/2002/07/owl#"
 xml:base="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7"
 rdf:about="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#IntegerIntervalMeta.value">
 <rdfs:range rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-
7#IntegerInterval"/>
 <rdfs:domain rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-
7#IntegerIntervalMeta"/>
 <rdfs:label xml:lang="en">IntegerIntervalMeta.value</rdfs:label>
 <skos:prefLabel xml:lang="en">Integer Interval Value</skos:prefLabel>
 <rdfs:isDefinedBy rdf:resource="http://nsgreg.nga.mil/as/view?i=180261"/>
 <skos:definition xml:lang="en">Definition: An integer-interval domain value.
  Description: [None Specified]</skos:definition>
</owl:ObjectProperty>
```

Listing 19. Definition of property IntegerInterval.reason in the NEO

```
<?xml version="1.0" encoding="UTF-8"?>
<owl:ObjectProperty xmlns:skos="http://www.w3.org/2004/02/skos/core#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:owl=
"http://www.w3.org/2002/07/owl#"
 xml:base="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7"
 rdf:about="http://api.nsgreg.nga.mil/ontology/neo-ent/1-7#IntegerIntervalMeta.reason
">
 <rdfs:range rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-enum/1-
7/VoidNumValueReason"/>
 <rdfs:domain rdf:resource="http://api.nsgreg.nga.mil/ontology/neo-ent/1-
7#IntegerIntervalMeta"/>
 <rdfs:label xml:lang="en">IntegerIntervalMeta.reason</rdfs:label>
 <skos:prefLabel xml:lang="en">Value Reason</skos:prefLabel>
 <rdfs:isDefinedBy rdf:resource="http://nsgreg.nga.mil/as/view?i=180260"/>
 <skos:definition xml:lang="en">Definition: The condition due to which the attribute
value may be missing or otherwise not fulfil the specification of the attribute value
domain. Description: For example, it may be the case that the attribute value is
unknown or that it is known but due to policy considerations it cannot be
given.</skos:definition>
</owl:ObjectProperty>
```

Thus, in the NEO, the value of a property (that has an XxxMeta class as range) is given either by the

XxxMeta.value or the XxxMeta.reason property.

In the GeoJSON example (Listing 12), the value of a single feature property can be used to encode an actual value or the reason for its absence. The allowed reason codes are represented by specific string values (e.g. "No Information" or "noInformation") and numeric values (e.g. -999999). Also, the feature property length ("LZN") has an actual value, while properties area ("ARA") and full name ("ZI005_FNA") do not (instead, code values define the reason for the absence as 'no information').

NOTE The NEO representation of a code does not directly contain integer codes. However, the definition of these codes is sometimes contained in the resources referenced by the rdfs:isDefinedBy predicate. For example, for code http://api.nsgreg.nga.mil/ontology/neo-enum/1-7/VoidValueReason/noInformation (see Listing 17), the predicate refers to http://nsgreg.nga.mil/as/view?i=132011, which states that the numeric value for this code is -999999.

JSON-LD does not support a value-based mapping of a single JSON key to two different RDF properties. Therefore, the NAS/NEO value or reason pattern is not directly supported by JSON-LD.

6.2.2.4. NEO geometry representation does not use GeoSPARQL

NEO classes that represent geometries are aligned with geometry classes produced by the ISO TC211 Group for Ontology Management (GOM). For example, neo:PointPositionInfo is a subclass of http://def.isotc211.org/iso19107/2003/GeometricPrimitive#GM_Point. The ontology files for ISO 19107:2003 are available in the GitHub repository ISO-TC211/GOM [https://github.com/ISO-TC211/GOM]. However, none of these classes is a subclass of geosparql:Geometry (or one of its subclasses). This has two consequences:

- Applications that support GeoSPARQL cannot perform spatial computations with NEO RDF data.
- The transformation of geometries encoded in JSON, as defined before, to a GeoSPARQL-friendly form, and subsequent RDF serialization using JSON-LD, does not result in an RDF encoding that is compliant with the current NEO. The ability to natively perform spatial computations with RDF data using GeoSPARQL-capable applications suggests that a future version, or variant, of the NEO could be aligned with GeoSPARQL rather than the ISO TC211 GOM geometry classes.

6.2.3. Potential solutions

As described in the previous sections, the significant structural differences between the GeoJSON data from the example in Listing 12 and the NEO encoding cause a number of issues when attempting to map the GeoJSON feature data and their properties to NEO classes and properties through a JSON-LD @context, and serializing the JSON-LD data as RDF. The following sections present potential solutions for these issues.

6.2.3.1. Semantically-enable JSON, without serializing as RDF

As specified in the overview section of this chapter, the primary goal of this analysis is to use JSON-LD to define the semantics of JSON objects as well as their key-value pairs. Serialization as RDF is a secondary concern.

In order to semantically enable JSON data, a JSON-LD @context can just be used to identify the

concepts represented by JSON objects and their keys. In addition, the context can be used to define key value types in more detail.

The NSG Standards Registry [https://nsgreg.nga.mil] contains the NSG Core Vocabulary (NCV) Register [https://nsgreg.nga.mil/voc/registers.jsp?register=NCV], which defines the NCV as follows:

> The NSG Core Vocabulary (NCV) specifies a controlled vocabulary of terms (i.e., defined lexical items) that are intended for use in the National System for Geospatial Intelligence (NSG) community to consistently and unambiguously refer to elements of shared Geospatial Intelligence (GEOINT).

The NCV is based on SKOS. As such, it defines SKOS concepts (and concept schemes) that define the meaning of terms. A JSON-LD @context based on the NCV for the GeoJSON example is shown in Listing 20.

NOTE The concepts defined by the NCV are OWL individuals, not RDF/OWL class or property definitions. As such, a serialization of JSON data as RDF, based upon a mapping to NCV, would not result in consistent RDF data, and thus should not be attempted.

Listing 20. GeoJSON example - JSON-LD @context defining the semantics of the aeronautical features based on the NCV

```
{"@context": {
 "@version": 1.1,
 "ncv": "http://api.nsgreg.nga.mil/vocabulary/ncv/",
 "xsd": "http://www.w3.org/2001/XMLSchema#",
 "geojson": "https://purl.org/geojson/vocab#",
 "features": "geojson:features",
 "properties": "@nest",
 "F_CODE": "@type",
 "GB030": "ncv:Helipad",
 "GB075": "ncv:Taxiway",
 "APT": {
  "@id": "ncv:airfieldUse",
  "@type": "xsd:integer"
 },
 "APT2": {
  "@id": "ncv:airfieldUse",
  "@type": "xsd:integer"
 },
 "APT3": {
  "@id": "ncv:airfieldUse",
  "@type": "xsd:integer"
 },
 "ARA": {
  "@id": "ncv:featureArea",
  "@type": "xsd:integer"
```

```
  },
  "AXS": {
   "@id": "ncv:aerodromeSurfaceStatus",
   "@type": "xsd:integer"
  },
  "CAA": {
   "@id": "ncv:controllingAuthority",
   "@type": "xsd:integer"
  },
  "LZN": {
   "@id": "ncv:featureLength",
   "@type": "xsd:integer"
  },
  "UFI": {
   "@id": "ncv:uniqueEntityIdentifier",
   "@type": "xsd:string"
  },
  "WID": {
   "@id": "ncv:featureWidth",
   "@type": "xsd:integer"
  },
  "ZI005_FNA": {
   "@id": "ncv:fullName",
   "@type": "xsd:string"
  },
  "ZI020_GE4": {
   "@id": "ncv:gencShortUrnBasedIdentifier",
   "@type": "xsd:string"
  },
  "ZI026_CTUC": {
   "@id": "ncv:cartographicUsabilityRange",
   "@type": "xsd:integer"
  },
  "ZI026_CTUL": {
   "@id": "ncv:cartographicUsabilityRange",
   "@type": "xsd:integer"
  },
  "ZI026_CTUU": {
   "@id": "ncv:cartographicUsabilityRange",
   "@type": "xsd:integer"
  },
  "ZSAX_RS0": {
   "@id": "ncv:resClassification",
   "@type": "xsd:string"
  },
  "geometry": "ncv:entityPlace"
 }}
```

Listing 21 shows the result of applying the NCV based JSON-LD @context to the original GeoJSON data (shown in Listing 12), in expanded form [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#expanded-

document-form].

Listing 21. GeoJSON example - resulting JSON-LD when applying the NCV based @context, in expanded form


```
[
  {
    "https://purl.org/geojson/vocab#features": [
      {
        "http://api.nsgreg.nga.mil/vocabulary/ncv/entityPlace": [
          {}
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/airfieldUse": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 1
          },
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": -999999
          },
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": -999999
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/featureArea": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 150
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/controllingAuthority": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 16
          }
        ],
        "@type": [
          "http://api.nsgreg.nga.mil/vocabulary/ncv/Helipad"
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/featureLength": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 50
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/uniqueEntityIdentifier": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#string",
            "@value": "588fb71e-5965-11e4-863e-7845c4f8683b"
          }
```

```
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/fullName": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#string",
            "@value": "Metropolitan Hospital"
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/gencShortUrnBasedIdentifier": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#string",
            "@value": "ge:GENC:3:1-2:USA"
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/cartographicUsabilityRange": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 5
          },
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 1
          },
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 200000
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/resClassification": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#string",
            "@value": "U"
          }
        ]
      },
      {
        "http://api.nsgreg.nga.mil/vocabulary/ncv/entityPlace": [
          {}
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/featureArea": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": -999999
          }
        ],
        "http://api.nsgreg.nga.mil/vocabulary/ncv/aerodromeSurfaceStatus": [
          {
            "@type": "http://www.w3.org/2001/XMLSchema#integer",
            "@value": 2
          }
        ],
        "@type": [
```

```
            "http://api.nsgreg.nga.mil/vocabulary/ncv/Taxiway"
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/featureLength": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#integer",
              "@value": 165
            }
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/uniqueEntityIdentifier": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#string",
              "@value": "94f2deb1-a88c-11e4-b9c7-7845c4f86835"
            }
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/featureWidth": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#integer",
              "@value": 12
            }
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/fullName": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#string",
              "@value": "No Information"
            }
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/gencShortUrnBasedIdentifier": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#string",
              "@value": "ge:GENC:3:1-2:USA"
            }
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/cartographicUsabilityRange": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#integer",
              "@value": 5
            },
            {
              "@type": "http://www.w3.org/2001/XMLSchema#integer",
              "@value": 1
            },
            {
              "@type": "http://www.w3.org/2001/XMLSchema#integer",
              "@value": 25000
            }
          ],
          "http://api.nsgreg.nga.mil/vocabulary/ncv/resClassification": [
            {
              "@type": "http://www.w3.org/2001/XMLSchema#string",
              "@value": "U"
            }
```

```
        ]
      }
    ]
  }
]
```

> **NOTE**
>
> In this example, UFI is treated as an ordinary, string-valued feature property. As a consequence, the features do not have an ID in form of a URL. For establishing links between features, that may be insufficient. If UFI was declared as providing the object identifier, the objects that result from applying the JSON-LD @context can be referenced using the resulting ID. Declaring UFI as an identifier can be done as in the previous examples ("UFI": "@id" - in combination with "@base", for example "@base": "http://example.org/base/ncv/", the helipad would have the ID "http://example.org/base/ncv/588fb71e-5965-11e4-863e-7845c4f8683b").

In the resulting JSON-LD data (Listing 21), *entityPlace* is shown with an empty JSON object as value. The reason for this is that the NCV-based @context (Listing 20) does not define any terms to map the keys of the GeoJSON geometry value: "type" and "coordinates". However, the JSON-LD @context defines the meaning of GeoJSON "geometry" as *entityPlace*, which achieves the primary goal (to define the semantics of JSON data). An application that parses the data may be able to read the GeoJSON geometry, and thus know, for example, that the *entityPlace* of the helipad feature is located at (-74.0059731,40.7143528,57.293799999999).

Furthermore, *cartographicUsabilityRange* is shown as having three values. These values represent the components of the interval that defines the range, i.e. the lower value, upper value, and interval closure type. The NCV defines the term http://api.nsgreg.nga.mil/vocabulary/ncv/intervalClosureType but has no terms for lower and upper value. Therefore, the mapping of "ZI026_CTUC", "ZI026_CTUL", and "ZI026_CTUU" to *cartographicUsabilityRange* is (inevitably) imprecise. While all these keys have something to do with *cartographicUsabilityRange*, each has a different meaning concerning that range.

> **NOTE**
>
> While developing the JSON-LD @context for this potential solution, the need to provide additional metadata - for example human readable comments - for term definitions was identified. This led to a request to the JSON-LD working group for adding a way to provide such metadata. As a result, a new keyword might be added to the new version of the JSON-LD W3C standard. For further details, see the last paragraph in section JSON-LD keywords.

6.2.3.2. Purpose built intermediate ontology

The structural differences between the GeoJSON encoding from the example in Listing 12 and the NEO result in inconsistencies when mapping the GeoJSON data to NEO and serializing as RDF. When JSON data shall be converted to RDF data using a JSON-LD @context and the JSON-LD to RDF serialization, then structural differences can be avoided by mapping to an ontology that fits the structure of the JSON data. Such an ontology may need to be created first.

Turning the resulting RDF data into NEO compliant data would then require a mapping between the two ontologies. However, that work would only require RDF(S)/OWL tools; JSON would no

longer be part of the equation. Figure 6 illustrates this workflow.

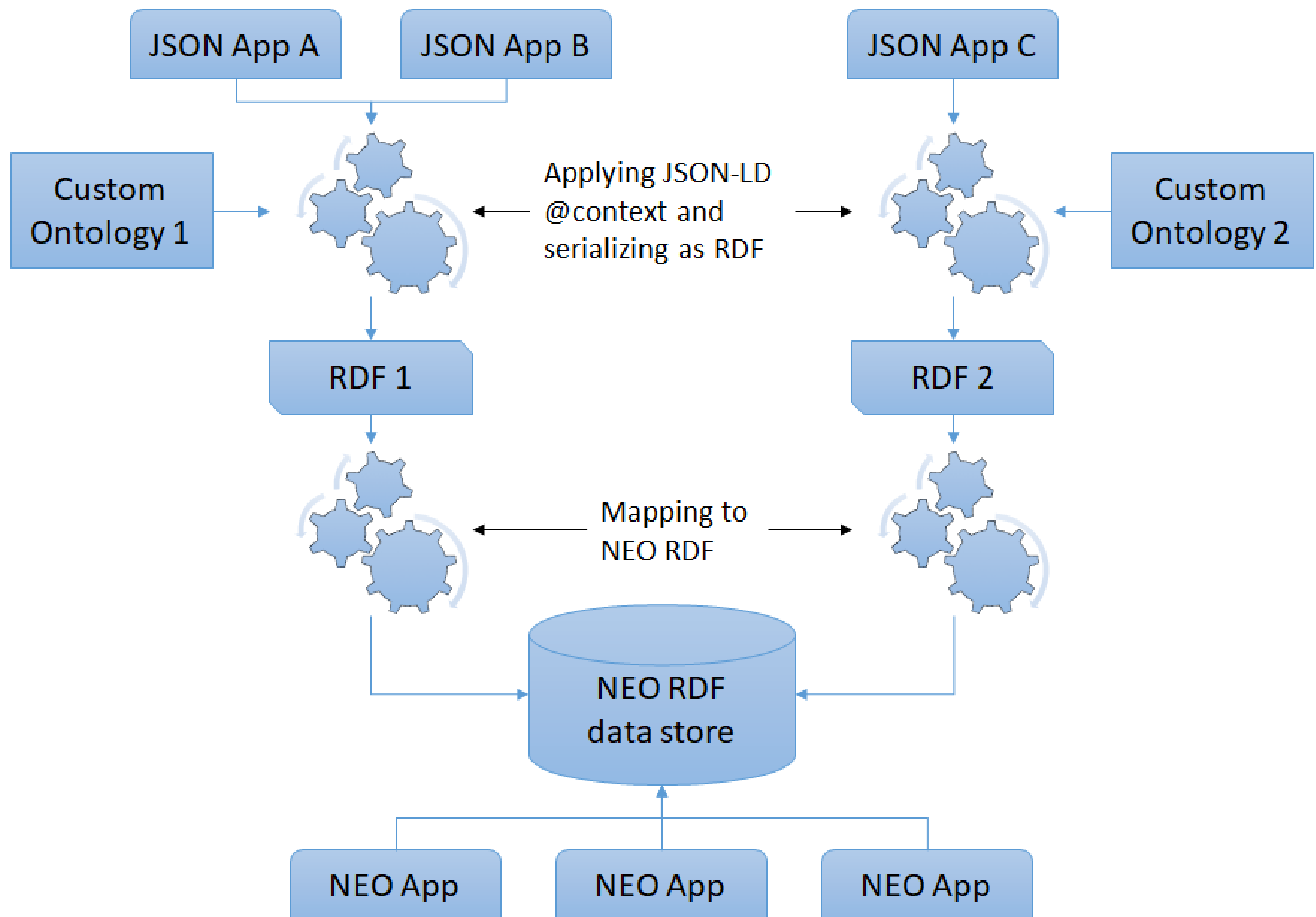

Figure 6. Converting JSON data to NEO RDF data using custom ontologies

A benefit of this approach would be that JSON data from multiple sources can be transformed into NEO compliant RDF data. A NEO RDF data store could then be used as a primary data source by multiple applications (see Figure 4 from the overview section).

6.3. Recommendations and best practices

The JSON-LD analysis performed in OGC Testbed-14 identified several recommendations and best practices for defining the semantics of JSON objects and their key-value pairs, and also for an (optional) subsequent serialization to RDF.

WARNING	The analysis was performed based upon the JSON-LD 1.1 final community group report [https://www.w3.org/2018/jsonld-cg-reports/json-ld/]. The feature set documented in that report may be slightly different in the final W3C standard.

6.3.1. Context dependent mappings

Mapping the same key (or value) to different IRIs in the same @context is not possible. This would be an issue if a GeoJSON feature collection contained different feature types that have properties with same name but different semantics. In such a situation, a @context cannot unambiguously define the semantics of these properties. One way to avoid this issue is to ensure that feature collections only contain a single type of feature.

If, on the other hand, a key (or value) has different meaning on different levels of a JSON structure, then it is possible to define the exact meaning for each level via scoped @contexts [https://www.w3.org/2018/jsonld-cg-reports/json-ld/#scoped-contexts] (a JSON-LD 1.1 feature). Consider the example in Listing 22.

Listing 22. JSON-LD example for context dependent mappings of keys and values - input

```
{
 "@context": {
  "@version": 1.1,
  "ex": "http://example.org/my/namespace/",
  "xsd": "http://www.w3.org/2001/XMLSchema#",
  "type": "@type",
  "CLS": "ex:ClassA",
  "key_1": {
   "@id": "ex:PropertyA",
   "@context": {
    "art": "@type",
    "CLS": "ex:ClassB",
    "key_2": {
     "@id": "ex:PropertyB_Level2",
     "@type": "@vocab"
    },
    "valueX": "ex:ValueX_Level2",
    "key_3": {
     "@id": "ex:PropertyC",
     "@context": {
      "key_2": {
       "@id": "ex:PropertyB_Level3",
       "@type": "@vocab"
      },
      "valueX": "ex:ValueX_Level3"
     }
    }
   }
  },
  "key_2": {
   "@id": "ex:PropertyB_Level1",
   "@type": "@vocab"
  },
  "valueX": "ex:ValueX_Level1",
  "valueY": "ex:ValueY"
 },
 "type": "CLS",
 "key_1": {
  "art": "CLS",
  "key_2": "valueX",
  "key_3": {"key_2": "valueX"}
 },
 "key_2": "valueY",
 "key_3": "unmapped"
}
```

Listing 23 shows the result of expanding the JSON-LD input from Listing 22.

Listing 23. JSON-LD example for context dependent mappings of keys and values - expanded

```
[
  {
    "http://example.org/my/namespace/PropertyA": [
      {
        "@type": [
          "http://example.org/my/namespace/ClassB"
        ],
        "http://example.org/my/namespace/PropertyB_Level2": [
          {
            "@id": "http://example.org/my/namespace/ValueX_Level2"
          }
        ],
        "http://example.org/my/namespace/PropertyC": [
          {
            "http://example.org/my/namespace/PropertyB_Level3": [
              {
                "@id": "http://example.org/my/namespace/ValueX_Level3"
              }
            ]
          }
        ]
      }
    ],
    "http://example.org/my/namespace/PropertyB_Level1": [
      {
        "@id": "http://example.org/my/namespace/ValueY"
      }
    ],
    "@type": [
      "http://example.org/my/namespace/ClassA"
    ]
  }
]
```

Context dependent mappings are applied for the key "key_2", and the values "CLS" as well as "valueX". The JSON-LD input from Listing 22 can be modified and the result tested on the JSON-LD development playground [https://json-ld.org/playground-dev/].

6.3.2. Handling geometry

In order for RDF applications to correctly process geometry information, geometry data encoded in JSON needs to be converted into an RDF format that the application understands. The list of potential options includes, but is not limited to:

- a GeoSPARQL geometry with either Well-Known Text (WKT) or GML based geometry serialization,
    - NOTE: In Testbed-14, that was the approach for converting JSON encoded geometry

information into a useful RDF representation. Further details are provided in the earlier analysis of developing a JSON-LD @context, here.

    ◦ The WKT serialization supports ISO 19125 Simple Feature geometries, which support many use cases. If the JSON geometry is more complex than a simple feature geometry, use the GML serialization.

- a schema.org type, such as GeoCoordinates [https://schema.org/GeoCoordinates], GeoShape [https://schema.org/GeoShape], Place [https://pending.schema.org/Place], or GeospatialGeometry [https://pending.schema.org/GeospatialGeometry],
- a Basic Geo [https://www.w3.org/2003/01/geo/] Point, and
- a Location Core (LOCN) geometry [https://www.w3.org/ns/locn#locn:Geometry], which can be encoded in a number of ways (for further details, see the LOCN specification).

The conversion of the geometry encoded in JSON can be performed by a data transformation. A JSON-LD @context can be defined for and applied to the result, to subsequently perform the RDF serialization.

6.3.3. JSON-LD keywords

The following list documents the keywords defined by the JSON-LD 1.1 final community group report [https://www.w3.org/2018/jsonld-cg-reports/json-ld/], including recommendations and observations that resulted from the work in Testbed-14:

- @base – Used to set the base IRI against which to resolve those relative IRIs interpreted relative to the document.
    - Use this keyword to construct absolute IRIs from relative IRIs contained in a JSON object identifier key (which is defined in the @context as mapping to "@id"). If @base is not set explicitly in the @context, the location of the JSON document would be used as base, which may not be desirable, particularly with respect to a possible change of the document URL in the future. If the JSON object identifier key value is an absolute IRI, @base is not used/needed to augment the IRI.
- @container – Used to set the default container type for a term.
- @context – Used to define the short-hand names that are used throughout a JSON-LD document. These short-hand names are called terms and help developers to express specific identifiers in a compact manner.
- @graph – Used to express a graph.
- @id – Used to uniquely identify things that are being described in the document with IRIs or blank node identifiers.
- @index – Used to specify that a container is used to index information and that processing should continue deeper into a JSON data structure.
- @language – Used to specify the language for a particular string value or the default language of a JSON-LD document.
- @list – Used to express an ordered set of data.
- @nest (new in JSON-LD 1.1) – Collects a set of nested properties within a node object.

- This keyword can be useful in case that the JSON format uses a key to nest a set of key-value pairs which would be more useful (particularly for a subsequent RDF serialization) to have in the JSON object that contains the nesting key. An example for a nesting key is the key "properties" of a GeoJSON feature object.

- @none (new in JSON-LD 1.1) – Used as an index value in an id map, language map, type map or elsewhere where a dictionary is used to index into other values.
- @prefix (new in JSON-LD 1.1) – With the value true, allows this term to be used to construct a compact IRI when compacting.
- @reverse – Used to express reverse properties.
- @set – Used to express an unordered set of data and to ensure that values are always represented as arrays.
- @type – Used to set the data type of a node or typed value.
  - When defining a node type: If the value does not expand to an absolute IRI, the value is combined with the value of "@vocab", if defined by the @context, otherwise with the document base.
  - When defining a value type: For keys that are mapped to properties whose value is a literal, the value type should be declared explicitly in the JSON-LD @context. That will allow a JSON-LD application to parse the value as this specific type, instead of as one of the set of general types that JSON supports (string, number, boolean).
- @value – Used to specify the data that is associated with a particular property in the graph.
- @version (new in JSON-LD 1.1) – Used in a context definition to set the processing mode. New features since JSON-LD 1.0 are only available when processing mode has been explicitly set to json-ld-1.1.
- @vocab – Used to expand properties and values in @type with a common prefix IRI.
  - Use this keyword to expand a string value to an IRI, as shown in the examples contained in the note in the analysis of numeric code values.
  - Do not use @vocab to define a global base for terms, especially if some JSON keys shall be ignored (e.g. if such a key is an application specific artifact). It is better to explicitly define the mappings of all relevant keys and values using compact IRIs.

During the analysis of JSON-LD performed in OGC Testbed-14, the need to provide additional metadata in a JSON-LD @context was identified. Such metadata could provide human readable documentation for a JSON-LD @context, and its term definitions - for example to provide further explanations for the mappings of JSON keys and values. A test revealed that the JSON-LD development playground did not support additional keywords like "@derivedBy" and "uom" in term definitions. Therefore, an issue [https://github.com/w3c/json-ld-syntax/issues/32] was raised in the GitHub repository of the JSON-LD working group, to ask if JSON-LD 1.1 provides a way to define additional metadata in a JSON-LD @context. A result of the discussion of the issue was that a new keyword would be desirable and helpful for users. It remains to be seen if such a keyword will be added to version 1.1 of the W3C JSON-LD standard.

6.4. Enhancing ShapeChange to derive JSON-LD @context documents

In order for ShapeChange to produce useful JSON-LD @context documents, ShapeChange needs to know:

1. the structure of the JSON data for which the @context shall be created, and
2. mappings of class and property names represented in JSON, to a semantic definition (e.g. an RDFS/OWL class, datatype, or property).

The ShapeChange JSON Schema target knows the structure of JSON data, to the extent defined by the JSON Schema encoding rule. Consequently, this target would be a good place to create a JSON-LD @context document.

> **NOTE** In order to support custom encoding rules, a new version of the JSON Schema target would need to be developed (for more details, see the future work item).

The mappings of class and property names represented in JSON would be defined using map entries. The *RdfTypeMapEntry* [https://shapechange.net/targets/ontology/uml-rdfowl-based-isois-19150-2/#RdfTypeMapEntry] and *RdfPropertyMapEntry* [https://shapechange.net/targets/ontology/uml-rdfowl-based-isois-19150-2/#RdfPropertyMapEntry] of the ShapeChange ontology target [https://shapechange.net/targets/ontology/uml-rdfowl-based-isois-19150-2/] appear to be suited to convey the information that is necessary for the mappings. These map entries can be created manually. However, it would also be possible to extend the ontology target to also produce RDF map entries for each ontology that is derived by the target (e.g. the NEO). Extending the ontology target in this way would have two benefits:

- The RDF map entries could be used by the JSON Schema target to create JSON-LD @context documents.
- The RDF map entries could also be used by the ontology target itself, when deriving ontologies for application schemas that have dependencies on application schemas for which ontologies and the map entries have already been derived.

However, using the RDF map entries derived by the ontology target may not work in all cases. An example would be that transformations change the names of application schema elements before the JSON Schema target or the ontology target is executed. Then there would be a mismatch between the UML model element names that the JSON Schema target encounters and the UML model element names that are found in RDF map entries (produced by the ontology target). The flattened structure of the GeoJSON example from section Converting GeoJSON data to NEO RDF data and the complex structure of the NEO directly illustrate such a mismatch. In that case, the JSON-LD @context could still provide stubs for properties for which no mapping was found, for example with value "FIXME" for the keys "@id" and "@type". The resulting JSON-LD @context would then need to be adjusted manually.

Chapter 7. Using both JSON Schema and JSON-LD

This section documents some theoretical considerations for the combined use of JSON Schema and JSON-LD.

The purpose of a JSON Schema is to validate the content of a JSON document. That document may contain JSON-LD data, i.e. contain JSON-LD keywords (such as @context, @id, @value, and @type). This analysis has primarily focused on JSON-LD @context documents being kept separate from the JSON data. The content of a JSON Schema for validating JSON-LD data would be significantly different, compared to a schema for validating pure JSON data.

The primary purpose of a JSON-LD @context document is to define the semantics and types of terms of a JSON document. The @context may also be used to serialize JSON data in RDF. The latter could be facilitated if the JSON structure would match the structure of classes in an RDF schema or OWL ontology. The structure of RDF triples is in some sense already built into JSON, due to the object-key-value encoding of JSON (which is leveraged by the JSON-LD RDF serialization algorithm). A JSON Schema could be created to ensure that the structure of the JSON data matches the structure of an RDF schema or OWL ontology. Additionally, some conventions for the JSON data would help when defining a JSON-LD @context document, for example requiring that a JSON object must contain a key that indicates the object type, and a key that defines its ID (if it represents an object with identity).

NOTE

The OGC Coverage Implementation Schema (CIS) v1.1 [http://docs.opengeospatial.org/is/09-146r6/09-146r6.html] contains an example for the combined use of JSON Schema and JSON-LD. The CIS JSON-Schema [http://schemas.opengis.net/cis/1.1/json/coverage-schema.json] checks - amongst other things - that id- and type-keys needed by JSON-LD @context documents are present in JSON data. The CIS JSON-LD @context documents [http://schemas.opengis.net/cis/1.1/rdf/] are used to derive RDF data (through serialization of JSON data). Note, however, that CIS does not define or identify a vocabulary or ontology to which that RDF data would be compliant to. This may be sufficient to enable linked data applications, but would be insufficient to perform reasoning on the RDF data.

Annex A: Revision History

Table 5. Revision History

Date	Editor	Release	Primary clauses modified	Descriptions
Oct 25, 2018	J. Echterhoff	1.0	all	Created this ER as result of splitting ER D022.
Nov 15, 2018	J. Echterhoff	1.0	1.2, 2, 6	Incorporate feedback from review by Geosemantics DWG
Nov 21, 2018	J. Echterhoff	1.0	throughout	Incorporate feedback from review by OGC IP team

Annex B: Bibliography

1. Portele, C.: OGC OWS-9 System Security Interoperability (SSI) UML-to-GML-Application-Schema (UGAS) Conversion Engineering Report. OGC (2013).
2. Echterhoff, J.: OGC Testbed-12 ShapeChange Engineering Report. OGC (2017).
3. Zyp, K., Court, G.: A JSON Media Type for Describing the Structure and Meaning of JSON Documents. IETF (2010).
4. Wright, A., Andrews, H.: JSON Schema: A Media Type for Describing JSON Documents. IETF (2018).
5. Wright, A., Andrews, H., Luff, G.: JSON Schema Validation: A Vocabulary for Structural Validation of JSON. IETF (2018).
6. Andrews, H., Wright, A.: JSON Hyper-Schema: A Vocabulary for Hypermedia Annotation of JSON. IETF (2018).
7. Bryan, P., Zyp, K., Nottingham, M.: JavaScript Object Notation (JSON) Pointer. IETF (2013).
8. Masó, J.: Testbed 11 Implementing JSON/GeoJSON in an OGC Standard Engineering Report. OGC (2015).