Best Practice for OGC - UML to JSON Encoding Rules

Abstract

This best practice document on UML to JSON encoding rules is an initiative *of Geonovum. The aim is to come to a standardized encoding from UML to JSON, in order to achieve technical interoperability in the chain from conceptual models to JSON implementation. In this document, JSON implementation includes plain JSON, GeoJSON and JSON-FG.*

Application schemas in UML are used to model geospatial information for a given domain, as part of data specifications defining information and data content of a relevant universe of discourse. In the geospatial domain, UML profiles are defined by [ISO 19103:2015] and [ISO 19109:2015]. These profiles are also used in this document. Application schemas operate at the conceptual level. At the implementation or data level, JSON is one of the major data encodings used by current web applications. The UML to JSON encoding rules defined in the context of this document is a best practice. Eventually, this document may also support the development of an international standard for the conversion of UML to JSON (Schema) in the geospatial domain.

To facilitate a proper JSON encoding, an extension of the ISO 19103 and 19109 UML profiles is proposed, resulting in a UML-JSON encoding profile.

The encoding rules are structured in 40 requirements and a number of recommendations, subdivided into 18 core requirements, 1 requirement specific for plain JSON schema format, 5 for GeoJSON format, 5 for JSON-FG Schema format, 3 requirements for binding and referencing of elements, 2 for union constructs, 5 for code lists and 1 for encoding a dedicated entity property in JSON. The requirement classes are supported by UML examples and subsequent JSON encodings.

Preface

The project leading to this document was initiated by Geonovum. In collaboration with Interactive Instruments a best practice is developed on encoding UML to JSON. The project team consisted of the following persons:

- Clemens Portele (Interactive Instruments)
- Johannes Echterhoff (Interactive Instruments)
- Linda van den Brink (Geonovum)
- Paul Janssen (Geonovum)
- Pieter Bresters (Geonovum)
- Wilko Quak (Geonovum).

This document defines how a conceptual schema in UML, compliant to [ISO 19103:2015] and [ISO 19109:2015], can be encoded in JSON. A number of requirements classes are defined, which contain the necessary requirements and technical details.

JSON is one of the major data encodings used by current web applications. In order to achieve a high level of interoperability when exchanging JSON encoded data between such applications, especially when the applications are developed by different entities, the semantics and structure of the data need to be well defined. In the geospatial domain, conceptual schemas are used to define application relevant information. Typically, some additional schema language is used to define the structures for encoding the information. For JSON encoded data, such a schema language is JSON Schema. Ideally, the JSON Schema constructs can automatically be derived from a conceptual schema. For that task, a set of encoding rules is needed.

Within the OGC, the UML-to-GML Application Schema Pilot 2020 (UGAS-2020) was the first innovation initiative that produced a comprehensive set of encoding rules, for the conversion of ISO 19109 compliant application schemas in UML to JSON Schema. The UGAS-2020 Engineering Report documents the findings of that initiative.

This document is based on and extends the results of UGAS-2020 regarding JSON Schema encoding rules. It defines the conversion behavior in an implementation agnostic way, adhering to the requirements defined in [OGC 08-131r3] for writing OGC standards. This document thus represents the next step on the way to standardizing JSON Schema encoding rules in the geospatial domain.

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.

Security Considerations

No security considerations have been made for this document.

Scope

This document defines requirements for encoding application schemas in UML, which conform to the UML profile defined by [ISO 19103:2015] and potentially also [ISO 19109:2015], as JSON schemas. The requirements classes cover the creation of JSON schemas for:

- a plain JSON encoding;
- a GeoJSON-compliant encoding; and
- a JSON encoding compliant to JSON-FG 0.1

Additional requirements classes support encoding choices for unions, code list valued properties, property values given by reference, and entity types.

Conformance

This Best Practice defines a number of requirements classes, which specify encoding (elements of) an application schema in UML in JSON (Schema). The standardization target for all these classes are JSON (Schema) documents. Requirements classes for encoding an application schema in UML as a JSON Schema provides an overview of the requirements classes, together with their dependencies.

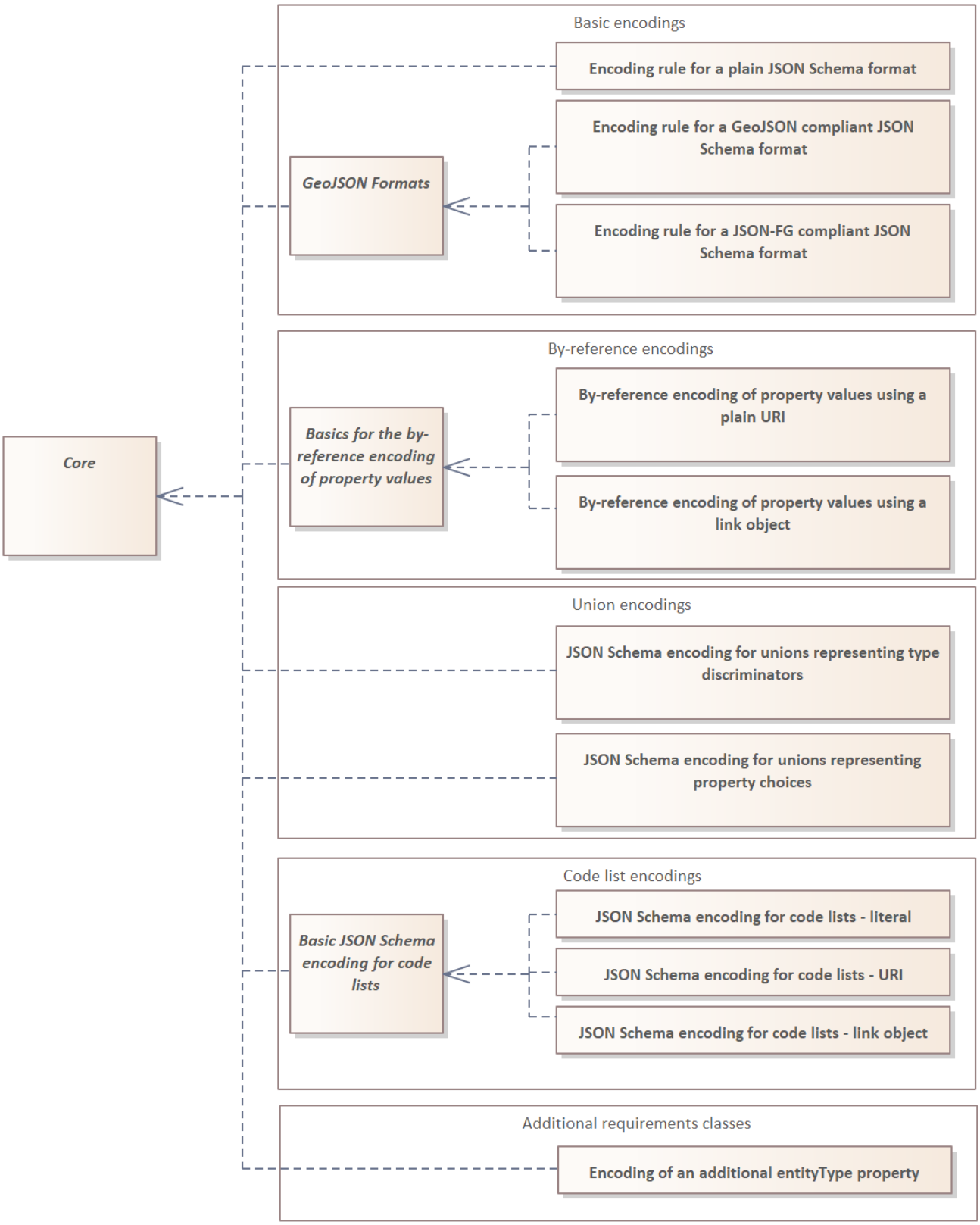

Figure 1. Requirements classes for encoding an application schema in UML as a JSON Schema

Common encoding behavior is defined in the core requirements class (Requirements class: Core). Additional requirements classes exist, which serve different purposes:

- Three requirements classes for basic JSON encodings are defined:
  - a plain encoding (Requirements class: Encoding rule for a plain JSON Schema format),
  - a GeoJSON encoding (Requirements class: Encoding rule for a GeoJSON compliant JSON

Schema format), and

    - a JSON-FG encoding (Requirements class: Encoding rule for a JSON-FG compliant JSON Schema format).

- There are two requirements classes that represent options for realizing a by-reference encoding of property values, one using URIs (Requirements class: by-reference encoding of property values using a plain URI reference, and one using link objects Requirements class: by-reference encoding of property values using a link object).
- Requirements classes also exist for the two ways in which «union» types are used in practice: as type discriminator (Requirements class: JSON Schema encoding for unions representing type discriminators) and as property choice (Requirements class: JSON Schema encoding for unions representing property choices).
- Code list valued properties can be represented in one of three ways, defined by requirements classes: as a literal (Requirements class: JSON Schema encoding for code lists - literal), as URI (Requirements class: JSON Schema encoding for code lists - URI), and as link object (Requirements class: JSON Schema encoding for code lists - link object).
- Finally, an additional requirements class supports the encoding of a JSON member for storing the name of the conceptual type that a JSON object encodes (Requirements class: Encoding of an additional entityType property).

A community can combine these requirements classes as needed to achieve a full JSON Schema encoding, which satisfies their specific JSON encoding requirements. For example, a community might choose a GeoJSON compliant encoding, together with the property choice encoding for unions, as well as using link objects for by-reference encoding and code values.

Requirements classes overview lists all requirements classes defined by this specification.

Table 1. Requirements classes overview

Requirements class	Clause
[http://www.opengis.net/spec/uml2json/1.0/req/core]	Requirements class: Core
[http://www.opengis.net/spec/uml2json/1.0/req/plain]	Requirements class: Encoding rule for a plain JSON Schema format
[http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats]	Requirements class: GeoJSON Formats
[http://www.opengis.net/spec/uml2json/1.0/req/geojson]	Requirements class: Encoding rule for a GeoJSON compliant JSON Schema format
[http://www.opengis.net/spec/uml2json/1.0/req/jsonfg]	Requirements class: Encoding rule for a JSON-FG compliant JSON Schema format
[http://www.opengis.net/spec/uml2json/1.0/req/by-reference-basic]	Requirements class: basics for the by-reference encoding of property values
[http://www.opengis.net/spec/uml2json/1.0/req/by-reference-uri]	Requirements class: by-reference encoding of property values using a plain URI reference

Requirements class	Clause
[http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object]	Requirements class: by-reference encoding of property values using a link object
[http://www.opengis.net/spec/uml2json/1.0/req/union-type-discriminator]	Requirements class: JSON Schema encoding for unions representing type discriminators
[http://www.opengis.net/spec/uml2json/1.0/req/union-property-choice]	Requirements class: JSON Schema encoding for unions representing property choices
[http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic]	Requirements class: Basic JSON Schema encoding for code lists
[http://www.opengis.net/spec/uml2json/1.0/req/codelists-literal]	Requirements class: JSON Schema encoding for code lists - literal
[http://www.opengis.net/spec/uml2json/1.0/req/codelists-uri]	Requirements class: JSON Schema encoding for code lists - URI
[http://www.opengis.net/spec/uml2json/1.0/req/codelists-link-object]	Requirements class: JSON Schema encoding for code lists - link object
[http://www.opengis.net/spec/uml2json/1.0/req/entitytype]	Requirements class: Encoding of an additional entityType property

This document does not define conformance classes. Such classes become relevant if and when this specification moves on in the OGC standardization process, i.e., if the document type changes from Best Practice to Standard. It is expected that the standardization process will involve further review and discussion, and may lead to changes within the specification. Once the specification has reached a stable state during that process, that would be a good time to define conformance classes.

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

References

- [IETF RFC 8259]IETF RFC 8259, The JavaScript Object Notation (JSON) Data Interchange Format
- [IETF RFC 7946]IETF RFC 7946, The GeoJSON Format
- [IETF RFC 6901]IETF RFC 6901, JavaScript Object Notation (JSON) Pointer
- [IETF I-D.draft-bhutton-json-schema-01]Internet Engineering Task Force (IETF). Draft draft-bhutton-json-schema-01: JSON Schema: A Media Type for Describing JSON Documents
- [IETF I-D.draft-bhutton-json-schema-validation-01]Internet Engineering Task Force (IETF). Draft draft-bhutton-json-schema-validation-01: JSON Schema Validation: A Vocabulary for Structural Validation of JSON
- [ISO 8601-2:2019]ISO 8601-2:2019, Date and time — Representations for information interchange — Part 1: Extensions
- [ISO 19103:2015]ISO 19103:2015, Geographic information – Conceptual schema language
- [ISO 19107:2003]ISO 19107:2003, Geographic information – Spatial schema
- [ISO 19109:2015]ISO 19109:2015, Geographic information – Rules for application schema

- [ECMA-262]ECMA-262, 11th edition specification, June 2020, https://www.ecma-international.org/ecma-262/11.0/index.html
- [OGC 21-045]OGC Features and Geometries JSON - Part 1: Core, draft 0.2.2, https://github.com/opengeospatial/ogc-feat-geo-json/releases/tag/v0.2.2
- [OGC 08-131r3]The Specification Model - A Standard for Modular specifications

Terms, definitions and abbreviated terms

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; and
- Include links to other, related URIs in the exposed data to improve discovery of other related information on the Web.

Source: W3C Semantic Web Wiki

feature type

A *feature type* as defined by the General Feature Model (see [ISO 19109:2015]). In an application schema, a *feature type* is typically modeled using stereotype «FeatureType», or a stereotype that maps to that stereotype.

object type

An *object type* is an interface or class. An *object type* is not a *feature type*. In an application schema, an *object type* is an interface, or a class with no stereotype, stereotype «Type», or a stereotype that maps to one of the two options.

data type

As defined by [ISO 19103:2015], section 6.10, a *data type* is a class with stereotype «DataType» (or a stereotype that maps to that stereotype), which is a set of properties that lacks identity.

type with identity

A class that is a *feature type* or an *object type*.

Abbreviated terms

API

Application Programming Interface

ECMA

European association for standardizing information and communication systems

GML

Geography Markup Language

HTTP

Hypertext Transfer Protocol

IETF

Internet Engineering Task Force

INSPIRE

Infrastructure for spatial information in Europe

IRI

Internationalized Resource Identifier

ISO

International Organization for Standardization

JSON

JavaScript Object Notation

JSON-FG

OGC Features and Geometries JSON

JSON-LD

JSON for Linked Data

OCL

Object Constraint Language

OGC

Open Geospatial Consortium

OWL

Web Ontology Language

RDF

Resource Description Framework

RDFS
: RDF Schema

UGAS
: UML to GML Application Schema

UML
: Unified Modeling Language

URI
: Uniform Resource Identifier

URL
: Unform Resource Locator

W3C
: World Wide Web Consortium

XML
: Extensible Markup Language

Conventions

General

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

Identifiers

The normative provisions in this document are denoted by the URI

`http://www.opengis.net/spec/uml2json/1.0`

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

JSON Schema URLs

This document uses the following base URLs for relevant JSON Schema files:

- GeoJSON: https://geojson.org/schema
- JSON-FG: https://beta.schemas.opengis.net/json-fg
- UML2JSON (defined by this specification): https://register.geostandaarden.nl/jsonschema/uml2json/0.1

NOTE	The JSON-FG schemas have not been published at https://beta.schemas.opengis.net/json-fg yet (as of Dec 12, 2022). The schema definition from this specification - see JSON Schema definitions - has been published at a temporary location. In both cases, the schema locations are expected to change during the OGC publication process.

Stereotype Names

Stereotype names within figures are written in lowerCamelCase, whereas stereotype names in the text are written in UpperCamelCase. In a future version of this document, all stereotype names should be written in UpperCamelCase, to follow UML 2 practice.

Overview

ISO / TC 211 defines Standards in the field of digital geographic information. A couple of these Standards, especially ISO 19109, are used by the geospatial community to define so called application schemas. An application schema is a conceptual schema for data required by one or more applications. It is typically defined using the Unified Modeling Language (UML).

OGC 07-036r1 defines rules for encoding an application schema in XML. The result is an XML Schema, which defines the structure for encoding application data in XML. Applications would use this XML as a format for interoperable information exchange.

JSON is a prominent format for encoding and exchanging data on the web. JSON Schema can be used to validate syntactical constraints for - i.e., the structure of - a specific JSON format. This document defines a set of requirements classes for encoding an application schema in UML in JSON Schema. The UML profiles defined by [ISO 19103:2015] and [ISO 19109:2015] are used as the base UML profile in this document.

NOTE	The results and findings from [OGC 20-012] have been an important foundation for the UML to JSON Schema encoding requirements defined in this specification.

UML to JSON Schema Encoding

Introduction

This chapter defines requirements classes for the encoding of application schemas in UML as JSON Schema, and likewise for the encoding of application data as JSON data.

UML profile

The stereotypes as well as tagged values that are relevant for JSON Schema encodings are listed in Stereotypes relevant for JSON Schema encodings and Tagged values relevant for JSON Schema encodings.

Table 2. Stereotypes relevant for JSON Schema encodings

Stereotype / keyword	Model element	Description
applicationSchema	Package	A conceptual schema for data required by one or more applications. Source: [ISO 19109:2015], especially chapter 8.2.
schema	Package	This stereotype is typically used in abstract schemas defined by ISO TC 211. For further details on abstract schemas, see [ISO 19103:2015], chapter 6.2 and figure 4. An abstract schema and an application schema are both conceptual schemas, but they are on different levels of abstraction. The stereotype «Schema» has been introduced for schemas that conform to ISO 19103, but do not follow the rules for application schemas from ISO 19109, but still need a stereotype on the schema package for adding tagged values.
featureType	Class	A feature type as defined by [ISO 19109:2015].

Stereotype / keyword	Model element	Description
interface	Interface	An abstract classifier with operations, attributes and associations, which can only inherit from or be inherited by other interfaces. Other classifiers may realize an interface by implementing its operations and supporting its attributes and associations (at least through derivation). Source: [ISO 19103:2015] This stereotype is typically used in conceptual schemas from ISO TC 211. It should not be used in application schemas, as these are on a different conceptual level than classifiers with this stereotype.
dataType	DataType	A set of properties that lack identity (independent existence and the possibility of side effects). A data type is a classifier with no operations, whose primary purpose is to hold information. Source: [ISO 19103:2015]
union	Class	Either used as *A structured data type without identity where exactly one of the properties of the type is present in an instance* (property choice) or *type consisting of one and only one of several alternative datatypes* (type discriminator; source: [ISO 19103:2015]) - in both cases the options are listed as member attributes.

Stereotype / keyword	Model element	Description
enumeration	Enumeration	A fixed list of valid identifiers of named literal values. Attributes of an enumerated type may only take values from this list. Source: [ISO 19103:2015]
codeList	Class	A flexible enumeration that uses string values for expressing a list of potential values. Source: [ISO 19103:2015]
property	Property (attribute [not of an enumeration or code list] or association role)	A property of a schema type which is not an enumeration or code list.

NOTE Communities may use aliases for the stereotypes listed above.

NOTE For backwards-compatibility to UML 1 schemas (that comply with earlier versions of [ISO 19103:2015]), a class with stereotype «Type» can be used instead of an interface.

NOTE Some conceptual schemas and application schemas do not make use of the stereotypes «property», but still attach certain tagged values to according properties in their UML model. That approach is supported by some UML modeling tools, even though tagged values typically belong to a certain stereotype. For the purposes of this specification, the stereotype «property» is assumed to be applied in schemas that shall be converted from UML to JSON Schema. Nevertheless, it is allowed for schemas to omit the stereotypes, and just use the associated tagged values (see below) on according model elements. This kind of use implies the presence of an ad-hoc stereotype, which is considered to represent the «property» stereotype.

Tagged values relevant for JSON Schema encodings lists the tagged values relevant for the JSON Schema encodings, together with the stereotype(s) they apply to, as well as relevant requirements class(es).

Table 3. Tagged values relevant for JSON Schema encodings

Applicable stereotype(s)	Tagged value	Relevant requirements class(es)	Comment
«applicationSchema» and «schema»	jsonDocument	[http://www.opengis.net/spec/uml2json/1.0/req/core]	see Definitions schema
	jsonId	[http://www.opengis.net/spec/uml2json/1.0/req/core]	
«Enumeration» and «CodeList»	literalEncodingType	[http://www.opengis.net/spec/uml2json/1.0/req/core] and [http://www.opengis.net/spec/uml2json/1.0/req/codelists-literal]	see Enumeration and Requirements class: JSON Schema encoding for code lists - literal

Applicable stereotype(s)	Tagged value	Relevant requirements class(es)	Comment
«property»			

		net/spec/uml2json/1.0/req/jsonfg]	temporal information
Applicable stereotype(s)	**Tagged value** Interval	**Relevant requirements Class(es)** [http://www.opengis.net/spec/uml2json/1.0/req/jsonfg]	**Comment**
	unit	[http://www.opengis.net/spec/uml2json/1.0/req/core]	see External types

Requirements class: Core

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core

target

JSON (Schema) documents

inherit

[IETF I-D.draft-bhutton-json-schema-01]

inherit

[IETF I-D.draft-bhutton-json-schema-validation-01]

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/definitions-schema

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/schema-references

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/iso19103-primitive-types

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/iso19103-measure-types

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/class-name

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/abstract-types

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/generalization

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/feature-and-object-types

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/data-types

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/enumerations

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/basic-types

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/properties

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/property-inline

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/property-multiplicity

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/property-fixed-readonly

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/property-derived

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/property-initial-value

requirement

http://www.opengis.net/spec/uml2json/1.0/req/core/association-class

recommendation

http://www.opengis.net/spec/uml2json/1.0/req/core/id-characteristics

recommendation

http://www.opengis.net/spec/uml2json/1.0/req/core/format-and-pattern

Definitions schema

Schema packages have the stereotype «applicationSchema», «schema», or an alias (e.g., using a specific language, like in German: «anwendungsschema»). An «applicationSchema» package represents an application schema according to ISO 19109. The stereotype «schema» has been introduced for packages that should be treated like application schemas, but do not contain feature types.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/definitions-schema

A UML application schema and its classes shall be converted into a single *definitions schema*.

NOTE	A definitions schema is a JSON Schema that uses the "$defs" keyword.

The file name of the *definitions schema* shall be constructed as follows: If tagged value *jsonDocument* is set on the application schema package, with a value that is not blank (i.e., purely whitespace characters), then the tag value shall be used as the file name of the definitions schema. Otherwise, the package name shall be used as fallback, replacing all spaces and forward slashes with underscores, and appending '.json'.

The "$schema" keyword shall be added to the definitions schema. Its value shall be "https://json-schema.org/draft/2020-12/schema".

The definitions schema shall have an "$id" member, whose value is the value of tag *jsonId*, as defined on the application schema package.

The "$defs" keyword shall have a JSON object as value, where each member represents the JSON Schema definition of a class from the application schema.

NOTE	When encoding the content of an application schema in a single *definitions schema*, it is straightforward to assign the JSON Schema URL whenever such a reference is required for one of the application schema classes. If the content of the application schemas was distributed over multiple *definitions schema* files, it would be necessary to maintain a mapping for each application schema class, to the URL of the JSON Schema that contains the definition of that class.
NOTE	The "$id" identifies the schema resource with its canonical URI. The URI is an identifier and not necessarily a resolvable URL. If the "$id" is a URL, there is no expectation that the JSON Schema can be downloaded at that URL.
NOTE	If the value of tag *jsonId* is a URI with a path to a named file, then the file name given in tag *jsonDocument* should match the file name in tag *jsonId*. It is not required to do so, because the JSON Schema file can be re-named during the publication process.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/id-characteristics

statement

It is recommended that the "$id" URI should be stable, persistent, and globally unique.

JSON Schema example of a definitions schema

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/some-definitions-schema.json",
  "$defs": {
    "Class1": {
      "type": "object",
      "properties": {
        "prop1": {"type": "string"}
      },
      "required": ["prop1"]
    },
    "Class2": {
      "type": "object",
      "properties": {
        "prop2": {"type": "number"}
      },
      "required": ["prop2"]
    }
  }
}
```

NOTE	The "$id" of the definitions schema has been omitted in some examples within this chapter. Declaring an absolute, non-existent URL as "$id" in these examples can prevent the examples from working, when testing them using certain JSON Schema validators, for instance on https://www.jsonschemavalidator.net/.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/schema-references

statement

References from types of the application schema to other types - within the same or within an external schema - shall be encoded as references to the according definitions schemas, using the JSON Schema keyword "$ref" - see References between JSON Schemas using $ref with JSON pointers as values.

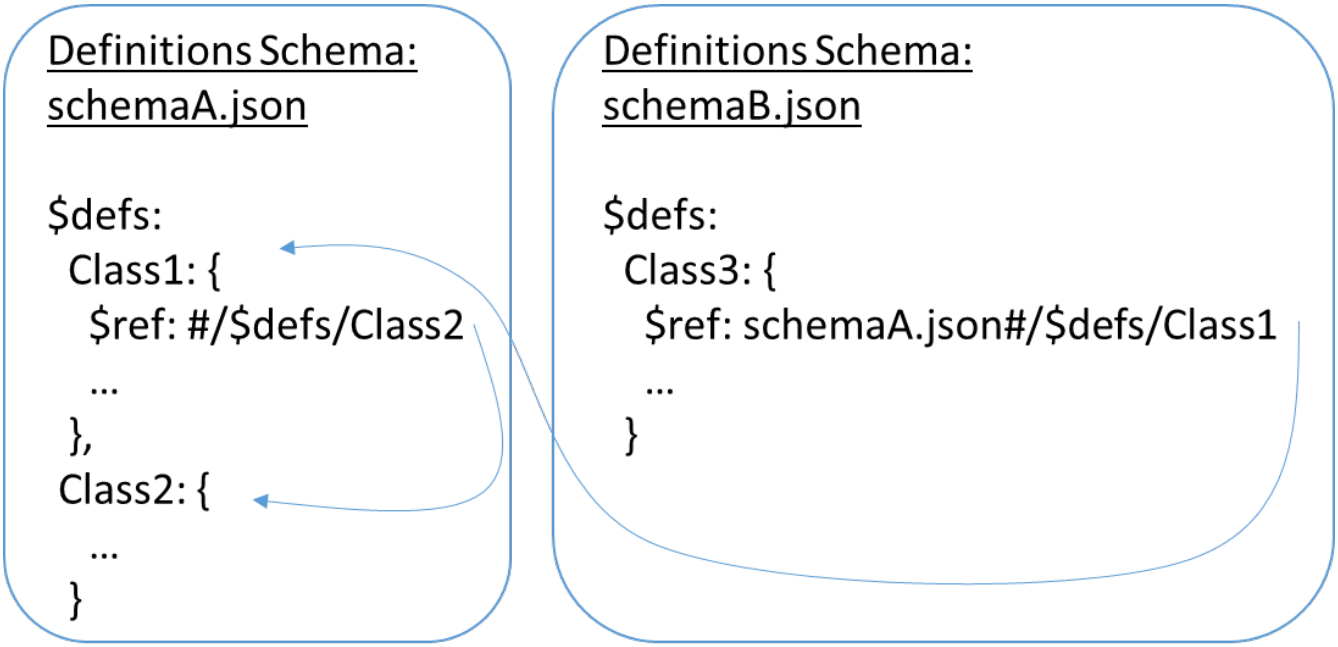

Figure 2. References between JSON Schemas using $ref with JSON pointers as values

It is up to the encoder to use an absolute or relative URI for a reference to a schema definition within an external JSON Schema file. A reference to a schema definition within the same JSON Schema file should be encoded as a relative URL that consists of a fragment identifier, either using a JSON Pointer (e.g., #/$defs/XYZ) or an anchor (e.g., #XYZ).

A link to a particular definition within a definitions schema requires the use of a JSON Pointer or an anchor in the fragment identifier of the link URL. JSON Pointer, chapter 6, explicitly states that the media type in which a JSON value is provided needs to support this kind of fragment identifier, and that this is not the case for the media type application/json. If a JSON Schema was published with this media type, then it is possible that the application ignores a fragment identifier (because the media type does not support fragment identifiers). If a JSON Schema is published with media type application/schema+json, using anchors and JSON Pointers as fragment identifiers is supported.

The JSON Schema should be published with media type application/schema+json - which is defined by the JSON Schema specification. The media type application/schema+json supports JSON Pointers and plain names as fragment identifiers. For further details, see JSON Schema core, chapter 5.

The JSON Schema with which 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, section 9.5).

NOTE

Specific formats may encode such links in the JSON data itself. Also see https://ietf-wg-httpapi.github.io/mediatypes/draft-ietf-httpapi-rest-api-mediatypes.html and the "schema" parameter defined there. Some tools (e.g., the oXygen editor) use a "$schema" member in JSON (instance) data to reference the applicable JSON Schema. However, that is a tool-specific approach. In principle, tool-specific approaches are allowed. However, the choice of re-using the "$schema" keyword in that way is problematic, since it does not reflect the intent of "$schema" as defined by JSON Schema core, section 8.1.1. The relationship between a JSON document and the JSON Schema for validation can also be defined explicitly by an application, i.e., in an application specific way.

Documentation

Descriptive information of application schema elements (packages, classes, attributes and association roles) may be encoded via JSON Schema *annotations*.

NOTE

Annotations represent one category of JSON Schema keywords (for further details, see JSON Schema core, section 7). *Annotations* attach information that applications can use as they see fit. The other categories are *assertions*, which validate that a JSON instance satisfies constraints, and *applicators*, which apply subschemas to parts of the instance and combine their results.

The documentation of an application schema element may be encoded using the JSON Schema "description" annotation. Additional annotations, such as "title" and "examples", may be used as well, where applicable.

NOTE

Potential reasons for NOT using JSON Schema annotations are:

1. Omitting the documentation will result in significantly smaller JSON Schema documents. The reduction of file size is preferable for processes that need to download the schema in order to apply validation. This is even more important if cross-references between JSON Schemas exist.
2. When validating JSON data against a JSON Schema, a JSON Schema validator typically focuses on the JSON Schema assertions and applicators, and will ignore most JSON Schema annotations - especially meta-data annotations, such as "title" and "description."

Types

External types

Application schemas typically use types from other schemas, for example the types defined by ISO 19103 and ISO 19107. External types can be used as value types of properties, and as supertypes for types defined in the application schema that is being converted.

Whenever an external type is used, its JSON Schema definition is needed. Either an external type is implemented as one of the simple JSON value types (e.g., string - maybe with a certain format or pattern), or it is defined by a particular JSON Schema. In case of a JSON Schema, the URL of that schema needs to be known during the encoding process. If the schema is a definitions schema, then the URL typically needs to be augmented with a fragment identifier that includes a JSON Pointer or an anchor reference within the schema.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/iso19103-primitive-types

statement

If a UML property is encoded in JSON Schema, and the value type is one of the ISO 19103 primitive types listed in JSON Schema implementation of ISO 19103 primitive types, then the simple JSON Schema type as well as the JSON Schema keywords listed in JSON Schema implementation of ISO 19103 primitive types shall be used in the JSON Schema definition of the property.

Table 4. JSON Schema implementation of ISO 19103 primitive types

UML class	JSON Schema simple type	JSON Schema keywords
Boolean	boolean	
CharacterString	string	
Date	string	format=date
DateTime	string	format=date-time
Decimal	number	
Integer	integer	
Number	number	
Real	number	
Time	string	format=time
URI	string	format=uri

Some JSON Schema validators do not support or ignore the JSON Schema keyword "format". That can be an issue, especially if a JSON Schema definition represented a choice (e.g., using the JSON Schema keyword "oneOf") between simple JSON Schema types. In that case, such a validator might complain that the choice cannot be made because both options match the simple type definition. The following recommendation is meant to prevent that issue.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/format-and-pattern

statement

Whenever an external type is implemented by a simple JSON Schema type with specific "format", it is recommended that the type definition be accompanied by a "pattern" member, whose value should contain a regular expression that is sufficient to represent the intended format.

Regular expressions for some ISO 19103 types, to be used in the JSON Schema 'pattern' keyword provides a list of regular expressions for a number of types from [ISO 19103:2015]. If the "pattern" keyword is used, these expressions should be used. However, applications may also use different regular expressions. For example, a community may choose to only allow date time values in Zulu time (i.e., requiring the time zone designator to always be 'Z').

Table 5. Regular expressions for some ISO 19103 types, to be used in the JSON Schema 'pattern' keyword

UML class	**Regular expression for use in JSON Schema 'pattern' keyword**
Date	`^\d{4}-\d{2}-\d{2}$`
DateTime	`^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d)?(Z|((\+|-)\d{2}:\d{2}))$`
Time	`^\d{2}:\d{2}:\d{2}(\.\d)?(Z|((\+|-)\d{2}:\d{2}))$`
URI	`^(([^:/?#]+):)?(\/\/([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?$`

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/iso19103-measure-types

statement

If a UML property is encoded in JSON Schema, and the value type is one of the ISO 19103 measure types (e.g., Measure, Length, Speed, Angle, Area, or Volume), then the JSON Schema definition of the property shall be constructed as follows:

part

If tagged value *unit* is defined on the UML property, with a non-blank value, then member "type" with value "number", and member "unit", with value being the value of tag *unit*, shall be encoded in the definition. If the multiplicity upper bound of the UML property is greater than 1, then as defined by [http://www.opengis.net/spec/uml2json/1.0/req/core/property-multiplicity], "type": "number" will be moved into the "items" member; however, the "unit" member shall still be encoded in the definition schema for the UML property, and not in the "items" member.

part

Otherwise, i.e., tag *unit* is undefined on the property, member "$ref" shall be added to the definition, with value "https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$defs/Measure" (the JSON Schema for measure is defined in JSON Schema definitions).

NOTE Tag *unit* identifies that the UML property has a fixed unit of measure. Having a fixed unit for a given property is highly beneficial for practical applications, for example when writing queries and filter statements. Note that it is perfectly valid for an application schema to still use a measure type for an attribute with a fixed unit, instead of just type 'Number' or 'Real'. The reason is that the application schema is defined on the conceptual level. From that point of view, a measure typed attribute with a fixed unit still has a measure as value type, not just a number. It is on the implementation level that the simplification of just using a number value for a measure typed property with fixed unit makes sense.

NOTE If ISO TC 211 defines a JSON Schema for ISO 19103 measure types, then that JSON Schema definition can be used.

NOTE If an external type is not covered by the mapping tables defined in this document, then a suitable mapping needs be found on a case by case basis. For example, if CI_Citation from ISO 19115 was used by an application schema, a reference to a suitable JSON Schema definition needs to be identified. Such a schema definition could be created manually, or (semi-) automatically, for example using the encoding behavior defined in this specification.

Class name

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/class-name

statement

The name of a class shall be encoded as value of an "$anchor" member in the schema definition of the class (within the definitions schema).

NOTE	Schema definitions that have an "$anchor" can be referenced using the plain text value of the anchor as fragment identifier, instead of using a more complex JSON Pointer.

Example of a JSON Schema with $anchor members

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "TypeA": {
      "$anchor": "TypeA",
      "...": "..."
    },
    "TypeB": {
      "$anchor": "TypeB",
      "...": "..."
    }
  }
}
```

Examples of referring to the schema definition of "TypeA" from a "$ref" member:

- within the JSON schema itself:
  - using the "$anchor" value: "$ref" = "#TypeA"
  - using a JSON Pointer: "$ref" = "#/$defs/TypeA"
- from another JSON Schema:
  - using the "$anchor" value: "$ref" = "https://example.org/schemas/schema_definitions.json#TypeA"
  - using JSON Pointer: "$ref" = "https://example.org/schemas/schema_definitions.json#/$defs/TypeA"
    - NOTE: If the referenced schema is a draft 07 JSON Schema, the JSON Pointer would have to change as follows: https://example.org/schemas/schema_definitions.json#/definitions/TypeA

NOTE	Keep in mind that the use of a fragment identifier with anchor or JSON Pointer value in $ref references can depend upon the media type with which the referenced JSON schema is published, as is explained in more detail here.

Abstractness

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/abstract-types

statement

An abstract class shall be encoded like a non-abstract class.

NOTE	JSON Schema does not directly support abstractness.

Encoding an abstract class as a schema definition allows that definition to be referenced from the schema definitions that are created for the subclasses of the abstract class.

Inheritance

JSON Schema does not support the concept of inheritance itself. In practice, an inheritance relationship is important in two areas:

- when defining the structure of a subtype, which inherits the properties of its supertypes through the generalization relationships to those supertypes, and
- when using a supertype as UML property value; in that case, subtypes can be used as property value, too, and validation is typically expected to check a value based upon its actual type - especially if a subtype is used as value.

Generalization can be represented in JSON Schemas. Validation of property values that are subtypes of the defined property value type cannot fully be represented in JSON Schema.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/generalization

statement

The generalization relationship of a subtype to its supertype shall be encoded by combining the structural constraints of the subtype and its supertype using the "allOf" JSON Schema keyword in the JSON Schema definition of the subtype.

Multiple inheritance is supported by adding all supertypes as elements of "allOf."

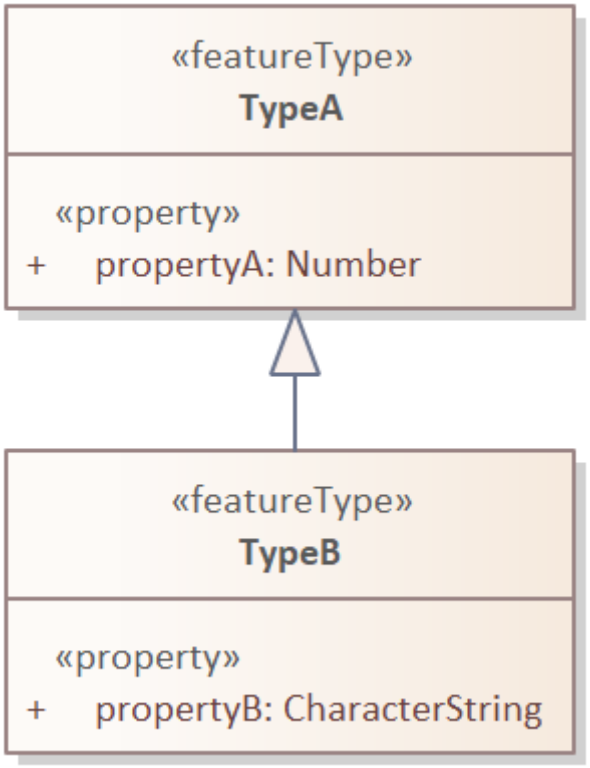

Figure 3. Example of type inheritance

JSON Schema example for realizing generalization using "allOf"

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "TypeA": {
      "properties": {
        "propertyA": {
          "type": "number"
        }
      },
      "required": [
        "propertyA"
      ]
    },
    "TypeB": {
      "allOf": [
        {
          "$ref": "#/$defs/TypeA"
        },
        {
          "type": "object",
          "properties": {
            "propertyB": {
              "type": "string"
            }
          },
          "required": [
            "propertyB"
          ]
        }
      ]
    }
  },
  "$ref": "#/$defs/TypeB"
}
```

This JSON object is valid against the schema:

```
{
  "propertyA": 2,
  "propertyB": "x"
}
```

This JSON object is invalid (because "propertyA" is missing) against the schema:

```
{
  "propertyB": "x"
}
```

NOTE	This also works for an encoding where the properties of a class are nested within a key-value pair (like "properties" for a GeoJSON encoding).
NOTE	The case where a property from a supertype is redefined by a property from the subtype is supported. Redefinition in UML requires that the value type of the subtype property is "kind of" the type of the redefined property of the supertype. Therefore, the property value, when encoded in JSON, would satisfy the JSON Schema constraints defined by both the subtype property and the redefined supertype property.

This approach to converting a generalization relationship has the following restrictions.

- The JSON Schema keyword "additionalProperties" cannot be set to false in the definitions of both the super- and the subtype.
- The approach is only defined for generalization relationships of feature, object, and data types. For unions, enumerations, and code lists, generalization relationships are not defined by [ISO 19103:2015].
- It only converts the generalization relationship from subtype to supertype. It does not support the other direction of an inheritance relationship, i.e., specialization. Given a JSON object that encodes a subtype, and the JSON Schema of the supertype, then by validating the JSON object against that JSON Schema, only the constraints of the supertype are checked, but not all the constraints that apply to the subtype. That is an issue when encoding a UML property whose value type is or could be a supertype (via a subtype that is added by an external, so far unknown schema). Conceptually, the actual value of that property can be a supertype object, but it could just as well be an object whose type is a subtype of that supertype. This issue can only be solved to a certain degree with JSON Schema, as explained in [OGC 20-012], section Class Specialization and Property Ranges.

Common base schema

It is often useful to encode all classes that have a certain stereotype with a common base type. The generalization relationship to such a base type is often implied with the stereotype, for a given encoding. In GML, for example, the common base type for classes with stereotype «FeatureType» is gml:AbstractFeature. Rather than explicitly modeling such a base type (e.g., *AnyFeature* defined by ISO 19109), as well as explicitly modeling generalization relationships to the base type, the encoding rule typically takes care of adding that relationship to relevant schema types. Requirements class [http://www.opengis.net/spec/uml2json/1.0/req/core] does not declare specific common base types. That is left to other requirements classes.

Feature and object type

In the conceptual model, feature and object types represent objects that have identity. That

differentiates these types from, for example, data types. Other than that, feature and object types - in the following summarily called types with identity - are encoded as JSON objects, just like a data type.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/feature-and-object-types

statement

The feature and object types of an application schema shall be converted to JSON Schema definitions of JSON objects. These definitions shall be added to the definitions schema, using the type name as definition key.

NOTE	ISO 19109 requires class names to be unique within the scope of an application schema.

The conversion of the class properties is defined in Properties. General type conversion rules, such as those documented in Class name, may apply.

The conceptual model of a type with identity often does not contain an identifier property (a UML property whose value for field "isId" is set to true), whose value is used by applications to identify objects of that type. Instead, the according information is added or defined in platform specific encodings. For example, a GML application schema offers the gml:id attribute as well as the gml:identifier element to encode identifying information. In a web publishing context, the URI at which a JSON object is published can be used as its identifier. Requirements class [http://www.opengis.net/spec/uml2json/1.0/req/core] does not declare any specific mechanism for adding an identifier property. That could be achieved through the definition of a common base schema (see Common base schema). However, requirements regarding such a base schema are left to other requirements classes. Likewise, this requirements class does not define any requirements regarding the number and characteristics of identifier properties. Again, that is left to other requirements classes.

Data type

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/data-types

statement

A «DataType» shall be converted to a JSON Schema definition of a JSON object. That definition shall be added to the definitions schema, using the type name as definition key.

Enumeration

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/enumerations

An «Enumeration» shall be converted to a JSON Schema definition with a type defined by evaluating tagged value *literalEncodingType* on the enumeration.

The tagged value *literalEncodingType* identifies the conceptual type that applies to the enumeration values. If the tagged value is not set on the enumeration, or has an empty value, then the literal encoding type is defined to be CharacterString.

The JSON Schema definition shall use the "enum" keyword to restrict the value to one of the enums from the enumeration. The "enum" value shall be an array with one element per enum defined by the enumeration. For each enum, the array element shall be the initial value of the enum, if defined, otherwise it shall be the name of the enum.

The literal encoding type is one of the types from ISO 19103, which are implemented as a simple JSON Schema type - see Literal encoding type.

Table 6. Literal encoding type

Conceptual type from ISO 19103	**simple JSON Schema type**
CharacterString	string
Real, Number	number
Integer	integer

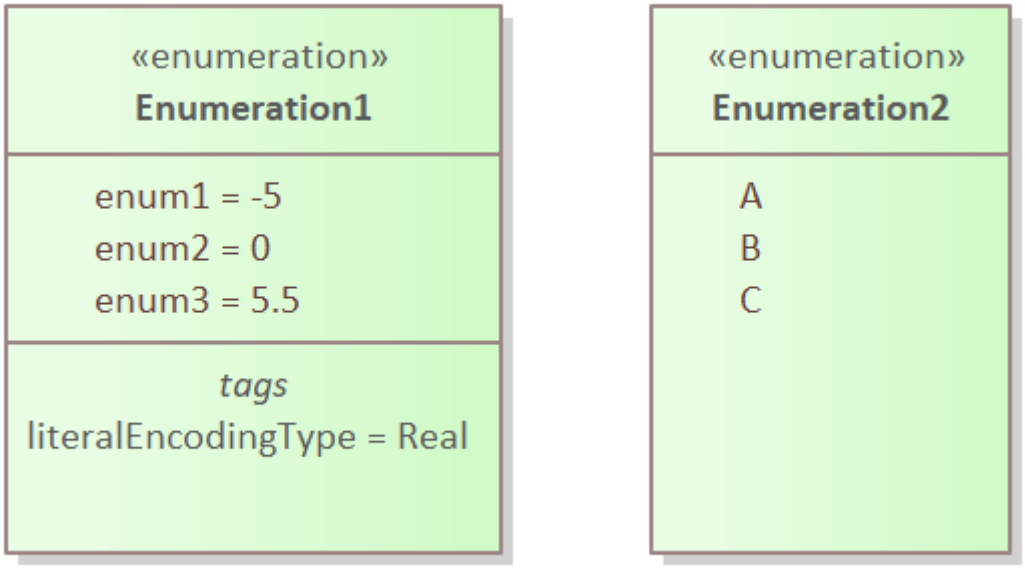

Figure 4. «Enumeration» example

Example of enumerations encoded in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Enumeration1": {
      "type": "number",
      "enum": [-5, 0, 5.5]
    },
    "Enumeration2": {
      "type": "string",
      "enum": ["A","B","C"]
    }
  }
}
```

Basic Type

If a direct or indirect supertype of an application schema class is represented in JSON Schema by one of the simple JSON Schema types *string*, *number*, *integer*, or *boolean*, then that class represents a so called *basic type*. A basic type does not define a JSON object. It represents a simple data value, e.g., a string. The JSON Schema definition of a basic type thus defines a simple JSON Schema type.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/basic-types

If the direct supertype of a basic type is implemented as one of the simple JSON Schema types, then the JSON Schema definition of the basic type shall have a "type" member with that simple JSON Schema type as value, and potentially additional JSON Schema keywords - especially "format" - which may be defined for the JSON Schema implementation of the supertype (for further details, see External types). Otherwise, the JSON Schema definition of the basic type shall reference the JSON Schema definition of its supertype using the "$ref" member.

NOTE If an official JSON Schema was published for the types defined in [ISO 19103:2015], then the definitions of that schema could be referenced, instead of creating a "type" member.

For each tag listed in Basic type restrictions, add the corresponding JSON Schema keyword (as defined in the table) to the JSON Schema definition of the basic type, with the tag value as value, if all of the following conditions are met:

- The tag is defined on the basic type and has a non-blank value.
- The simple JSON Schema type with which the basic type (or its direct or indirect supertype) is implemented is one of the simple JSON Schema types for which the JSON Schema keyword is applicable (as defined in Basic type restrictions).

If one or more JSON Schema keywords listed in Basic type restrictions are added to the JSON Schema definition of the basic type, and that definition does not declare a "type" member - i.e., it references the JSON Schema definition of its supertype via "$ref" - then an "allOf" keyword shall be used to combine the referenced schema definition and the list of additional JSON Schema keywords.

Table 7. Basic type restrictions

tagged value (to define a restriction)	JSON Schema keyword	applicable JSON Schema type(s)
jsonFormat	format	string, number, integer
maxLength	maxLength	string
minLength	minLength	string
jsonPattern	pattern	string
minInclusive	minimum	number, integer
minExclusive	exclusiveMinimum	number, integer
maxInclusive	maximum	number, integer
maxExclusive	exclusiveMaximum	number, integer

NOTE

The JSON Schema keyword "format" is defined in chapter 7 of JSON Schema Validation: A Vocabulary for Structural Validation of JSON. The formats defined there (e.g., "date-time", "uri", and "json-pointer") apply to JSON values of type string. Custom formats could apply to JSON values of type number and integer.

NOTE

JSON Schema Validation: A Vocabulary for Structural Validation of JSON defines the JSON Schema keyword "pattern". According to that specification, the value of the keyword should be a regular expression according to the [ECMA-262] regular expression dialect. JSON Schema: A Media Type for Describing JSON Documents defines a number of recommendations for writing regular expressions in JSON Schema.

NOTE

If the "format" keyword is used to restrict the structure of a JSON string, so that it matches a certain regular expression, then it is useful to add the "pattern" keyword as well, explicitly defining that regular expression (given that the regular expression follows an [ECMA-262] regular expression dialect). The reason is that the "format" is first and foremost an annotation, so can be ignored by JSON Schema validators, whereas the "pattern" keyword will be evaluated by a JSON Schema validator. JSON Schema validators may treat the "format" keyword like an assertion, but that is not guaranteed. In any case, the "format" keyword helps to convey more information about the specific type of a JSON value (e.g., "date" instead of just "string"), and thus should not be omitted if a certain, well-known (i.e., defined by a JSON Schema vocabulary) format is applicable to a JSON value.

Basic types example provides a detailed example that illustrates a number of cases. The JSON Schema encoding is shown in Example of basic types encoded in JSON Schema.

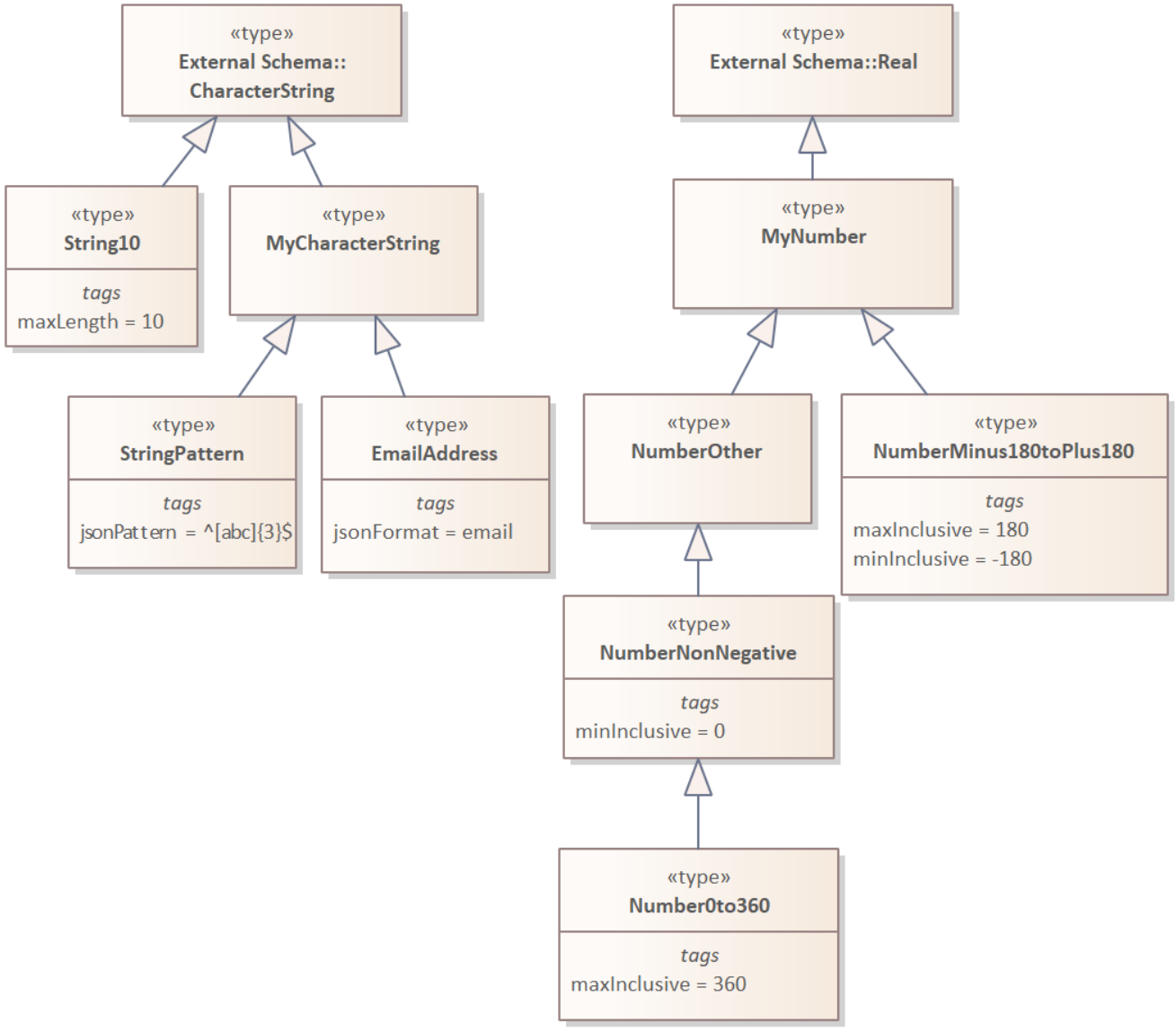

Figure 5. Basic types example

Example of basic types encoded in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "EmailAddress": {
      "allOf": [
        {
          "$ref": "#/$defs/MyCharacterString"
        },
        {
          "format": "email"
        }
      ]
    },
    "MyCharacterString": {
      "type": "string"
    },
    "MyNumber": {
```

```
      "type": "number"
    },
    "Number0to360": {
      "allOf": [
        {
          "$ref": "#/$defs/NumberNonNegative"
        },
        {
          "maximum": 360.0
        }
      ]
    },
    "NumberMinus180toPlus180": {
      "allOf": [
        {
          "$ref": "#/$defs/MyNumber"
        },
        {
          "minimum": -180.0,
          "maximum": 180.0
        }
      ]
    },
    "NumberNonNegative": {
      "allOf": [
        {
          "$ref": "#/$defs/NumberOther"
        },
        {
          "minimum": 0.0
        }
      ]
    },
    "NumberOther": {
      "$ref": "#/$defs/MyNumber"
    },
    "String10": {
      "allOf": [
        {
          "type": "string"
        },
        {
          "maxLength": 10
        }
      ]
    },
    "StringPattern": {
      "allOf": [
        {
          "$ref": "#/$defs/MyCharacterString"
        },
```

```
        {
          "pattern": "^[abc]{3}$"
        }
      ]
    }
  }
}
```

Properties

General

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/properties

statement

A UML property shall be converted to a member of a JSON object - unless the encoding rule defines a different behavior for the type that owns the property (e.g., for enumerations, unions, and code lists).

NOTE By default, UML properties are converted to keys within the "properties" member of the JSON Schema definition for the type that owns the property. Additional requirements may override this encoding (e.g., the type discriminator encoding of «union» properties), or augment the encoding (e.g., encoding the properties under the "properties" member of a GeoJSON-based feature).

The default result of converting a UML property, therefore, is a key within the "properties" key of the JSON Schema definition for the type that owns the property, with the key name being the name of the UML property, and the value being a JSON Schema with constraints and annotations that define the property (value type, multiplicity, etc).

The following figure and listing provide an example: UML type used to exemplify JSON Schema encoding of UML properties shows a feature type with a number of properties. Encoding UML properties in JSON Schema illustrates how the UML properties are represented within the "properties" of the JSON Schema that defines that type.

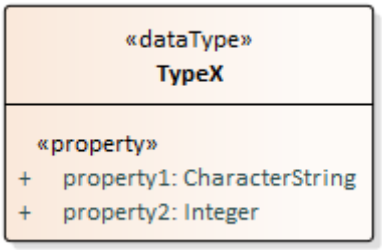

Figure 6. UML type used to exemplify JSON Schema encoding of UML properties

Encoding UML properties in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "TypeX": {
      "type": "object",
      "properties": {
        "property1": {"type": "string"},
        "property2": {"type": "number"}
      },
      "required": [
        "property1", "property2"
      ]
    }
  }
}
```

Value Type

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/property-inline

statement

The value type of a UML property shall be encoded as a JSON Schema constraint, as follows:

part

If the value type of a UML property is an external type, and the JSON Schema definition of that external type is a simple JSON value type, i.e., "string", "number", "integer", or "boolean", then a "type" key shall be added to the JSON Schema definition of the property, with the simple JSON value type as value;

part

Otherwise, a "$ref" key shall be added to the JSON Schema that constrains the property. The "$ref" value shall be a reference to the JSON Schema definition of the value type, within a particular definitions schema. The reference can be absolute or relative, and typically contains a fragment identifier to identify the definition of the value type.

Examples:

- using the "$anchor" value "TypeX" as fragment identifier: https://example.org/schemas/schema_definitions.json#TypeX
- using JSON Pointer as fragment identifier: https://example.org/schemas/schema_definitions.json#/$defs/TypeX
  - NOTE: If the referenced schema is a draft 07 JSON Schema, the JSON Pointer would have to change as follows: https://example.org/schemas/schema_definitions.json#/definitions/TypeX

NOTE	Keep in mind that the use of a fragment identifier with anchor or JSON Pointer value in $ref references can depend upon the media type with which the referenced JSON schema is published, as is explained in more detail here.

The behavior described in [http://www.opengis.net/spec/uml2json/1.0/req/core/property-inline] covers the case of an inline encoding of the property value. That is sufficient for simple application schemas. For more complex schemas, typically ones that contain associations between feature types, it can be necessary or desired to encode property values by-reference, i.e., using links. However, multiple options exist for realizing a by-reference encoding. These options are defined in separate requirements classes - see Additional requirements classes for the by-reference encoding of property values. A particular JSON Schema encoding of a given application schema needs to choose one of these options, in order to enable by-reference encoding for relevant properties.

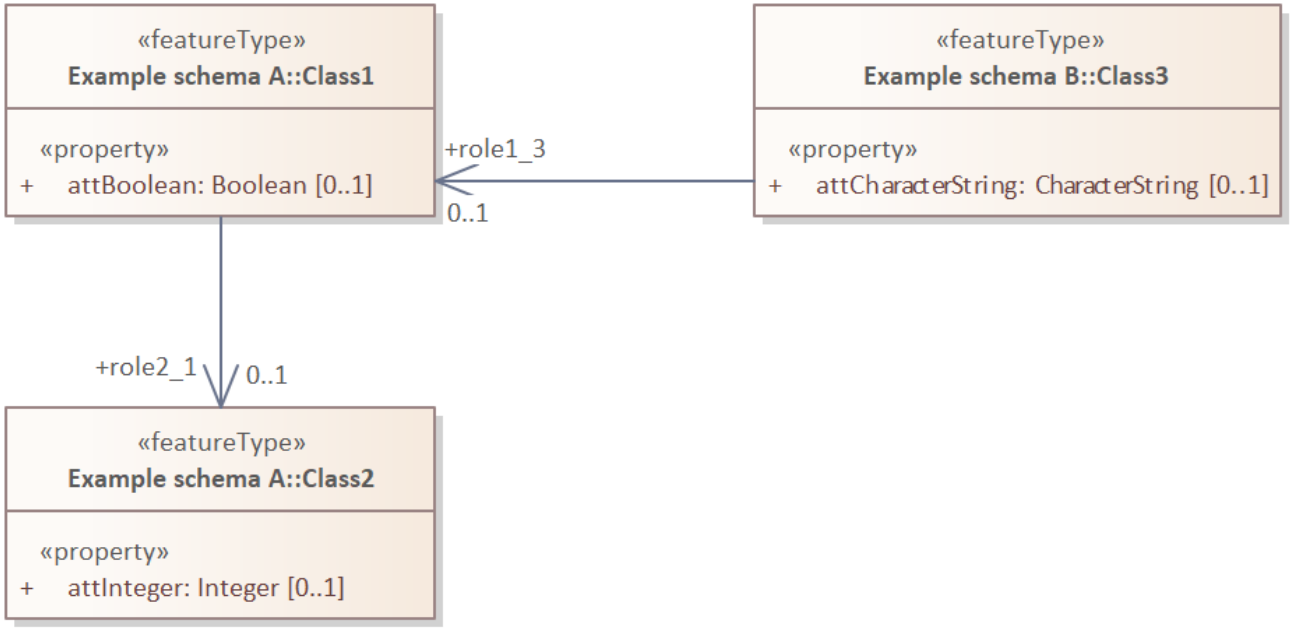

Figure 7. Examples of classes from two application schemas with properties that are implemented as simple JSON Schema types and as schema references

Example schema A, encoded as JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/schema/schemaA.json",
  "$defs": {
    "Class1": {
      "$anchor": "Class1",
      "type": "object",
      "properties": {
        "attBoolean": {
          "type": "boolean"
        },
        "role2_1": {
          "$ref": "#/$defs/Class2"
        }
      }
    },
    "Class2": {
      "$anchor": "Class2",
      "type": "object",
      "properties": {
        "attInteger": {
          "type": "integer"
        }
      }
    }
  }
}
```

Example schema B, encoded as JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/schema/schemaB.json",
  "$defs": {
    "Class3": {
      "$anchor": "Class3",
      "type": "object",
      "properties": {
        "role1_3": {
          "$ref": "schemaA.json#/$defs/Class1"
        },
        "attCharacterString": {
          "type": "string"
        }
      }
    }
  }
}
```

This JSON object is valid against the definition of "Class1" from Example schema A, encoded as JSON Schema:

```
{
  "attBoolean": true,
  "role2_1": {
    "attInteger": 2
  }
}
```

This JSON object is invalid (because "attInteger" has a string value, where an integer value is expected) against the schema from Example schema A, encoded as JSON Schema:

```
{
  "attBoolean": true,
  "role2_1": {
    "attInteger": "X"
  }
}
```

Multiplicity

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/property-multiplicity

If the multiplicity lower bound of a UML property is 1 or greater, and the class that owns the property is not a «union», then the property shall be listed under the "required" properties of the JSON object to which the property belongs.

In addition, if the multiplicity upper bound of the property is greater than 1, then the JSON Schema definition for the property shall be created as follows.

- The "type" of the JSON property is set to "array", with the "items" keyword containing the JSON Schema constraints that are created to represent the value type of the property.
- If the multiplicity lower bound is greater than 0, it is encoded using the "minItems" keyword.
- If the multiplicity upper bound is not unbounded, it is encoded using the "maxItems" keyword.
- If the values of the property are defined to be unique (which is the default for UML properties), then that is represented by adding `"uniqueItems": true`.

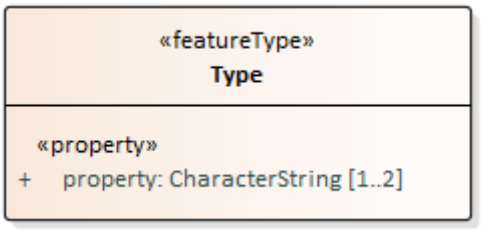

Figure 8. UML type used to exemplify JSON Schema encoding of multiplicity

Example for encoding multiplicity in JSON Schema

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Type": {
      "type": "object",
      "properties": {
        "property": {
          "type": "array",
          "minItems": 1,
          "maxItems": 2,
          "items": {
            "type": "string"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "property"
      ]
    }
  },
  "$ref": "#/$defs/Type"
}
```

This JSON object is valid against the schema from Example for encoding multiplicity in JSON Schema:

```
{
"property": ["a","b"]
}
```

This JSON object is invalid (because "property" has three values, which exceeds the maximum amount of allowed values) against the schema from Example for encoding multiplicity in JSON Schema:

```
{
"property": ["a","b","c"]
}
```

NOTE	All arrays in JSON are ordered, thus that the values of a UML property are ordered is always represented, and that the values of such a property are unordered cannot be represented. However, the latter should not matter to an application that does not expect ordered values for a certain property.

Fixed / readOnly

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/property-fixed-readonly

statement

The JSON Schema definition of a UML property that is marked as read only or fixed shall include the "readOnly" annotation with JSON value true.

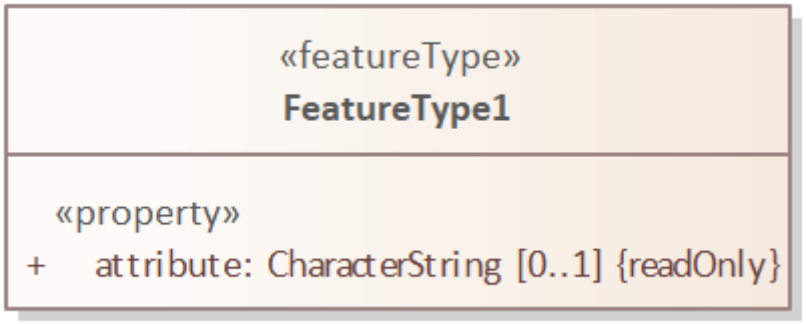

Figure 9. UML type used to exemplify JSON Schema encoding of a readOnly property

Example for encoding a readOnly property in JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "FeatureType1": {
      "$anchor": "FeatureType1",
      "type": "object",
      "properties": {
        "attribute": {
          "type": "string",
          "readOnly": true
        }
      }
    }
  }
}
```

Derived

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/property-derived

statement

The JSON Schema definition of a UML property that is marked as derived shall include the "readOnly" annotation with JSON value true.

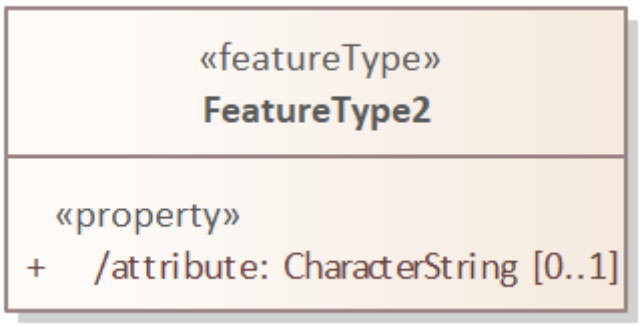

Figure 10. UML type used to exemplify JSON Schema encoding of a derived property

Example for encoding a derived property in JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "FeatureType2": {
      "$anchor": "FeatureType2",
      "type": "object",
      "properties": {
        "attribute": {
          "type": "string",
          "readOnly": true
        }
      }
    }
  }
}
```

Initial Value

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/property-initial-value

statement

A UML attribute that has an initial value, is owned by a type with identity or a «DataType», and whose value type is encoded as one of the simple JSON Schema types "string", "number", "integer", or "boolean", shall be encoded as follows:

The JSON Schema definition of the UML attribute shall include the "default" annotation with the initial value as value.

The value of the annotation can have any JSON value type. The initial value shall be encoded accordingly:

- quoted, if the JSON Schema type is "string";
- unquoted if the JSON Schema type is "number" or "integer"; and
- true if the JSON Schema type is "boolean" and the initial value is equal to, ignoring case, "true"; otherwise the value is false.

NOTE Theoretically, the default value can also be a JSON array or object, but that cannot be represented in UML and thus is not a relevant use case.

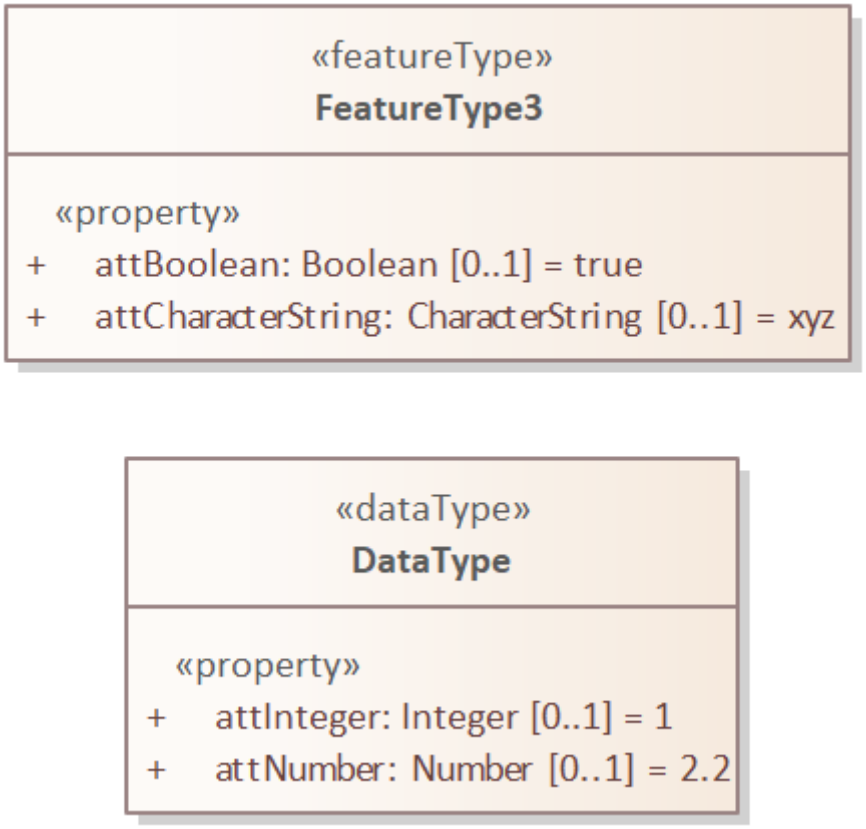

Figure 11. UML type used to exemplify JSON Schema encoding of properties with initial value

Example for encoding properties with initial value in JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "DataType": {
      "$anchor": "DataType",
      "type": "object",
      "properties": {
        "attInteger": {
          "type": "integer",
          "default": 1
        },
        "attNumber": {
          "type": "number",
          "default": 2.2
        }
      }
    },
    "FeatureType3": {
      "$anchor": "FeatureType3",
      "type": "object",
      "properties": {
        "attBoolean": {
          "type": "boolean",
          "default": true
        },
        "attCharacterString": {
          "type": "string",
          "default": "xyz"
        }
      }
    }
  }
}
```

Association class

Standard UML supports the concept of association class, i.e., an association that has properties. There is no native representation for association classes in JSON or JSON Schema. Association classes therefore need to be converted to "intermediate" classes, before being serialized to JSON Schema. The conversion is illustrated in the following figures, with Model with association classes showing the original conceptual model, and Association classes transformed to intermediate classes showing the conversion result.

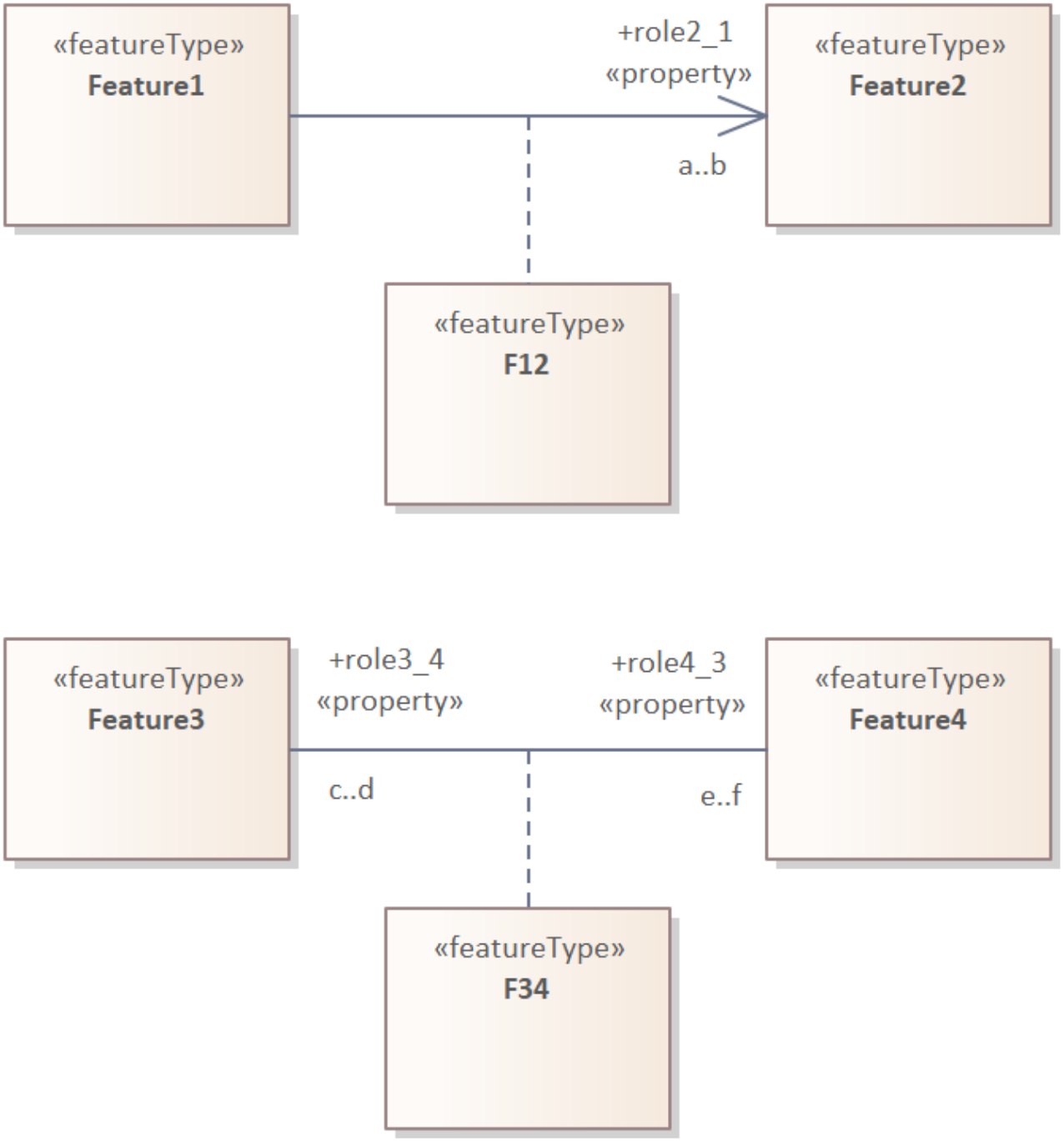

Figure 12. Model with association classes

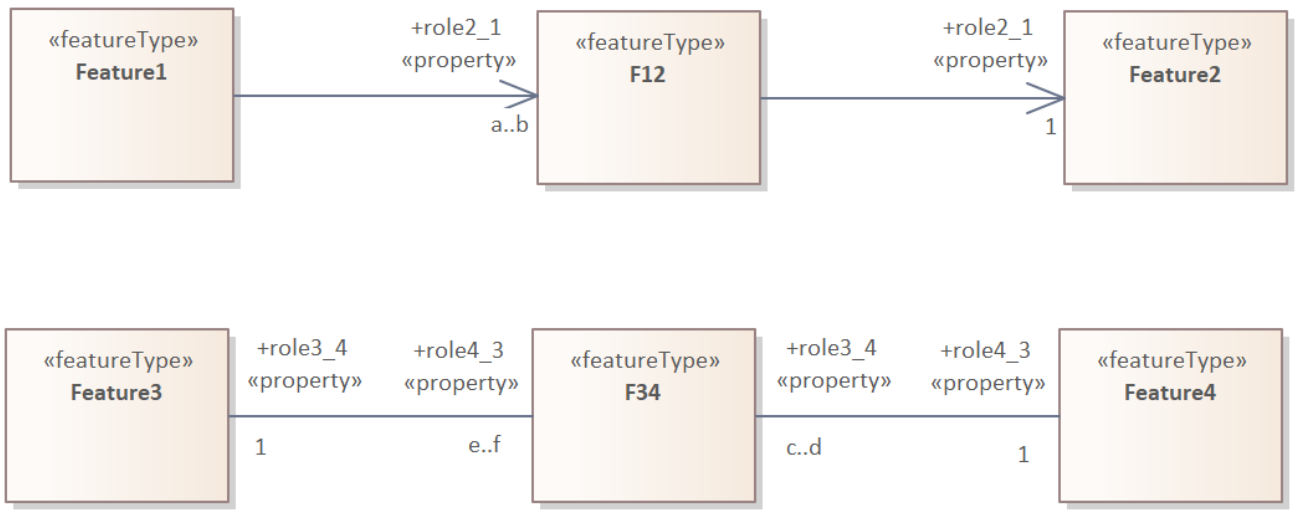

Figure 13. Association classes transformed to intermediate classes

identifier

http://www.opengis.net/spec/uml2json/1.0/req/core/association-class

statement

Before applying the conversion to JSON Schema, a UML association class that is a type with identity shall be transformed as follows (in the following description the source class of the association is called S and the target class is called T):

- The association class A is transformed into a regular class with the same name, stereotype, tagged values, constraints, attributes, and relationships.
- The association is replaced by two associations, one from S to A ("SA"), and one from A to T ("AT").
- The characteristics of the association end (in particular role name, navigability, multiplicity, documentation) of the original association class at T are used for association ends at A of SA and at T of AT, with the exception that the multiplicity at the association end at T of association AT is set to 1.
- The characteristics of the association end of the original association class at S are used for association ends at S of SA and at A of AT, with the exception that the multiplicity at the association end at S of association SA is set to 1.

Constraints

OCL constraints can be used to enrich a conceptual model with requirements that cannot be expressed in UML alone. However, this specification does not define any requirements for converting OCL constraints to JSON Schema definitions, or to any other format with which the constraints can be checked on a JSON dataset.

Conceptual model transformations

The conceptual schema may need to be transformed, in order to deal with model elements:

- that cannot be represented in a certain JSON format (e.g., a Solid - a 3D geometry type - as value for the "geometry" member of a GeoJSON feature); or
- that are not (well) supported by client software (e.g., complex attribute values for styling, processing, and filtering).

No specific model transformation requirements and recommendations are defined in this document. Examples of model transformations are given in [OGC 20-012] and in the GitHub repository with model transformation rules by the INSPIRE community.

Primary geometry

[OGC 23-058r1] defines the concept of *primary geometry*:

> **primary geometry**
>
> the geometry that the publisher considers as the most important spatial characteristic of a **feature**
>
> **NOTE** A feature can be described by multiple spatial properties. For example, a radio tower can have a property with a point value that describes the location of the tower and another property with a multi-polygon value that describes the area of coverage. Some feature formats can represent only a single geometry per feature. In those cases, the primary geometry will be used when the feature is encoded in such a format.
>
> **NOTE** The primary geometry of a feature can also vary depending on the zoom level. At a smaller scale, the primary geometry could be a point while a polygon could be used at a larger scale.
>
> — OGC API - Features - Part 5: Schemas, ref_ogcapifeatures_part5_schemas,section=4.1

The concept is generally applicable to feature types. Examples of data formats to which the concept applies are GeoJSON and JSON-FG. Requirements regarding the encoding of the primary geometry are defined in the according requirements classes ([http://www.opengis.net/spec/uml2json/1.0/req/geojson] and [http://www.opengis.net/spec/uml2json/1.0/req/jsonfg]).

In order to identify the UML property that represents the primary geometry of a feature type, the following approach is used in this specification:

- If a single (direct or inherited, but ignoring redefined) UML property of the feature type has tag "primaryGeometry" with value equal to and ignoring case "true", then that property is the primary geometry of the feature type.

- Otherwise, if the set of (direct and inherited, but ignoring redefined) UML properties of the feature type only contains a single UML property with a geometric type, and that property is directly owned by the feature type, and that property does not have tag "primaryGeometry" with value equal to, ignoring case, "false", then that property is the primary geometry of the feature type.
- Otherwise, no primary geometry is defined for the feature type.

NOTE	A feature type that has multiple UML properties with tag "primaryGeometry" = true is not modeled correctly.

NOTE	Setting tagged value "primaryGeometry" = false can be useful in cases of geometric properties of classes that are (expected to be) subtyped, with the subtypes defining their own primary geometry properties. If the supertype had a geometric property without such a tagged value, the second part of the rule (for determining the primary geometry) would apply, thereby incorrectly identifying the supertype property as primary geometry. That can lead to undesired JSON Schema constraints.

Primary temporal information

[OGC 23-058r1] defines the concept of *primary temporal information*:

> **primary temporal information**
>
> the time instant or time interval that the publisher considers as the most important temporal characteristic of a **feature**
>
> | | |
> |---|---|
> | **NOTE** | A feature can be described by multiple temporal properties. For example, an event can have a property with an instant or interval when the event occurred or will occur and another property when the event was recorded in the dataset. The primary temporal information can also be built from two properties, e.g., when the feature has two properties describing the start and end instants of an interval. |
>
> — OGC API - Features - Part 5: Schemas, ref_ogcapifeatures_part5_schemas,section=4.1

The concept is generally applicable to feature types. An example of a data format to which the concept applies is JSON-FG. Requirements regarding the encoding of primary temporal information are defined in the according requirements class ([http://www.opengis.net/spec/uml2json/1.0/req/jsonfg]).

In order to identify the UML property that represents the primary geometry of a feature type, the following approach is used in this specification:

- A UML property that is owned by the feature type and that has tag "primaryInstant" with value equal to and ignoring case "true" is the primary-instant of the feature type.

- A UML property that is owned by the feature type and that has tag "primaryInterval" with value equal to and ignoring case:
  - "interval" is the primary-interval of the feature type.
  - "start" is the primary-interval-start of the feature type.
  - "end" is the primary-interval-end of the feature type.

NOTE The value types of UML properties that represent or contribute to the primary interval should be compatible with that use. For example, properties marked as primary-interval-start or primary-interval-end can have value type "Date", "DateTime", or "TM_Instant", whereas a property marked as primary-interval can have value type "TM_Period".

NOTE

A feature type that does not satisfy the following conditions is not modeled correctly:

- At most one of the (direct or inherited, but ignoring redefined) properties has tag "primaryInterval" = "interval".
- At most one of the (direct or inherited, but ignoring redefined) properties has tag "primaryInterval" = "start".
- At most one of the (direct or inherited, but ignoring redefined) properties has tag "primaryInterval" = "end".
- The use of "interval" and "start"/"end" are mutually exclusive within the (direct or inherited, but ignoring redefined) properties of the feature type:
  - If one property has tag "primaryInterval" = "interval", then no other property has tag "primaryInterval" equal to "start" or "end".
  - Likewise, if one property has tag "primaryInterval" equal to "start" or "end", then no other property has tag "primaryInterval" = "interval".

Requirements class: Encoding rule for a plain JSON Schema format

identifier

http://www.opengis.net/spec/uml2json/1.0/req/plain

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

inherit

[IETF RFC 7946]

recommendation

http://www.opengis.net/spec/uml2json/1.0/req/plain/iso19107-types

NOTE	An example of an application schema encoded in plain JSON Schema format is given in Example schema in plain JSON encoding.

Common base schema

As described in the core requirements class, common base types or - for the purposes of this encoding rule - common JSON Schema definitions can be added to the schema definition of certain kinds of classes, for example, all feature types. This requirements class does not specify any such common JSON Schema definitions. As a consequence, if the types with identity defined by an application schema do not contain attributes that convey the identity of an actual object, according JSON objects cannot be identified using information from property values. Additional requirements classes, which depend on [http://www.opengis.net/spec/uml2json/1.0/req/plain], may add requirements regarding a common base.

Implementation of ISO 19107 types

identifier

http://www.opengis.net/spec/uml2json/1.0/req/plain/iso19107-types

statement

If a UML property is encoded in JSON Schema, and the value type is one of the ISO 19107 geometry types listed in the first column of JSON Schema implementation of types defined by ISO 19107 for the plain JSON Schema encoding rule, then it is recommended that the JSON schema definition in the second column of that table be used in the JSON Schema definition of the property.

Table 8. JSON Schema implementation of types defined by ISO 19107 for the plain JSON Schema encoding rule

UML class	JSON Schema reference
GM_Point	https://geojson.org/schema/Point.json
GM_Curve	https://geojson.org/schema/LineString.json
GM_Surface	https://geojson.org/schema/Polygon.json
GM_MultiPoint	https://geojson.org/schema/MultiPoint.json
GM_MultiCurve	https://geojson.org/schema/MultiLineString.json
GM_MultiSurface	https://geojson.org/schema/MultiPolygon.json
GM_Aggregate	https://geojson.org/schema/GeometryCollection.json
GM_Object	https://geojson.org/schema/Geometry.json

NOTE JSON Schema implementation of types defined by ISO 19107 for the plain JSON Schema encoding rule uses geometry types defined by [ISO 19107:2003]. While this specification does not define mapping tables for newer versions of ISO 19107, application schemas may use geometry types from a newer version of ISO 19107. The mappings would then need to be adjusted accordingly (finding correct replacements for the types mentioned in the first column of the table).

NOTE For geometry typed properties whose value type is not covered in JSON Schema implementation of types defined by ISO 19107 for the plain JSON Schema encoding rule, a suitable mapping needs to be defined, as explained in External types.

NOTE Other geometry encodings are allowed for the plain JSON Schema format, for example a WKT string or a JSON-FG geometry. Such geometry encodings may be useful in application scenarios where tools do not (only) support GeoJSON.

Identifier property

This requirements class does not define a means to add an identifier property - i.e., a UML property that is modeled with "isId" = true - to the JSON Schema encoding of a feature type, if that type does not declare such a property. The application schema would need to be transformed, in order to add an identifier property, where necessary.

Requirements class: GeoJSON Formats

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

inherit

[IETF RFC 7946]

requirement

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats/identifier

requirement

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats/nesting-feature-type-properties

Identifier property

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats/identifier

statement

If at least one UML property of a feature type is modeled with "isId" = true, then the top-level "id" member of GeoJSON features that encode instances of the feature type shall have a value.

NOTE A UML property of a feature type which is modeled with "isId" = true, is mapped to the "id" member of a GeoJSON feature. The mapping algorithm is community-specific. If a feature type has multiple UML properties where "isId" = true, or if the value type of such a property is not a simple type, some community specific conversion mechanism needs to be defined, for mapping the identifier value(s) to a simple string or number, which can be used as value of the "id" member of a GeoJSON feature.

NOTE The UML properties with "isId" = true are encoded as any other property, in addition to being mapped to the top-level "id" member. That is especially useful in case multiple such properties exist in a feature type, because applications that know the conceptual schema can read these dedicated properties directly, in order to gather information about the ID of a given feature, rather than having to decode the "id" member.

Nesting feature type properties

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats/nesting-feature-type-properties

statement

Properties of a feature type shall be encoded within the GeoJSON "properties" member, i.e., within a nested "properties" member.

NOTE Additional requirements can override this behavior, by omitting certain UML properties or by mapping certain UML properties to first-level members of the resulting JSON object. An example is the identifier property.

NOTE Properties of object types are encoded as first-level properties of the resulting JSON object. If object types should be encoded as feature types, then the object types would need to be transformed accordingly, before a JSON Schema encoding is created.

Requirements class: Encoding rule for a GeoJSON compliant JSON Schema format

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats

requirement

http://www.opengis.net/spec/uml2json/1.0/req/geojson/common-base

requirement

http://www.opengis.net/spec/uml2json/1.0/req/geojson/iso19107-types-for-geometry-member

requirement

http://www.opengis.net/spec/uml2json/1.0/req/geojson/primary-geometry

NOTE An example of an application schema encoded in GeoJSON compliant JSON Schema format is given in Example schema in GeoJSON-compliant encoding.

Common base schema

As described in the core requirements class, common base types or - for the purposes of this encoding rule - common JSON Schema definitions can be added to the schema definition of certain kinds of classes, for example, all feature types.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson/common-base

statement

All feature types shall use the GeoJSON Feature definition - https://geojson.org/schema/Feature.json - as common base.

part

The relationship to the GeoJSON Feature definition schema shall be implemented by converting a feature type to a JSON Schema that consists of an "allOf" with two subschemas: the first being a "$ref" with value "https://geojson.org/schema/Feature.json", the second being the schema produced by applying the other conversion rules to the feature type. However, if one of the supertypes of the feature type already has the GeoJSON Feature definition in its JSON Schema definition, then the JSON Schema definition of the feature type itself shall not define it again.

part

If the feature type is encoded with an "allOf" for the GeoJSON Feature definition, then the "$anchor" member (see Class name) shall be encoded in the schema that contains the "allOf", instead of within the second subschema.

NOTE No common base schema is defined for object types. Such types need to be transformed to feature types if they should be encoded as GeoJSON features.

Example of a feature type hierarchy in an application schema illustrates a feature type hierarchy, and Example for encoding the common base schema for feature types shows how these feature types are encoded in JSON Schema using the common base schema. Note that the definitions of the individual feature types still state "type": "object" in order to illustrate the place where object properties would be defined. However, such properties have been omitted in the example to avoid unnecessary complexity.

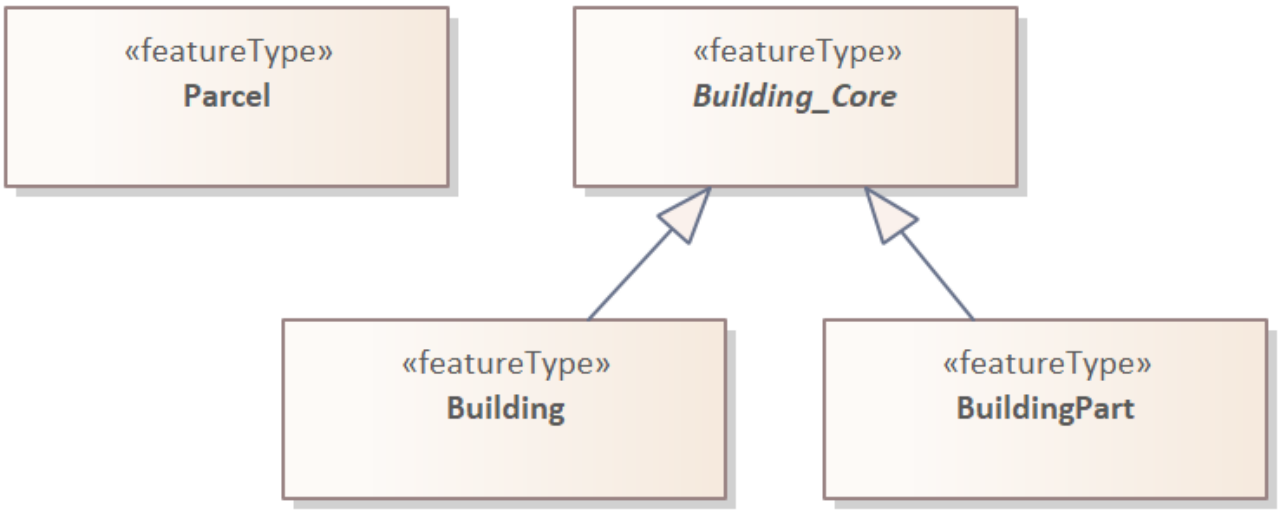

Figure 14. Example of a feature type hierarchy in an application schema

Example for encoding the common base schema for feature types

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/schema/infra.json",
  "$defs": {
    "Building": {
      "$anchor": "Building",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object"
        }
      ]
    },
    "BuildingPart": {
      "$anchor": "BuildingPart",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object"
        }
      ]
    },
    "Building_Core": {
      "$anchor": "Building_Core",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
```

```
          "type": "object"
        }
      ]
    },
    "Parcel": {
      "$anchor": "Parcel",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
          "type": "object"
        }
      ]
    }
  }
}
```

Implementation of ISO 19107 types for the "geometry" member

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson/iso19107-types-for-geometry-member

statement

If a UML property is encoded in JSON Schema, in the "geometry" top-level member of a JSON object that represents a type width identity, and the value type is one of the ISO 19107 geometry types listed in the first column of JSON Schema implementation of types defined by ISO 19107, for the "geometry" member in the GeoJSON encoding rule, then the JSON schema definition in the second column of that table shall be used in the JSON Schema definition of the property.

Table 9. JSON Schema implementation of types defined by ISO 19107, for the "geometry" member in the GeoJSON encoding rule

UML class	JSON Schema reference
GM_Point	https://geojson.org/schema/Point.json
GM_Curve	https://geojson.org/schema/LineString.json
GM_Surface	https://geojson.org/schema/Polygon.json
GM_MultiPoint	https://geojson.org/schema/MultiPoint.json
GM_MultiCurve	https://geojson.org/schema/MultiLineString.json
GM_MultiSurface	https://geojson.org/schema/MultiPolygon.json
GM_Aggregate	https://geojson.org/schema/GeometryCollection.json

UML class	JSON Schema reference
GM_Object	https://geojson.org/schema/Geometry.json

NOTE JSON Schema implementation of types defined by ISO 19107, for the "geometry" member in the GeoJSON encoding rule uses geometry types defined by [ISO 19107:2003]. While this specification does not define mapping tables for newer versions of ISO 19107, application schemas may use geometry types from a newer version of ISO 19107. The mappings would then need to be adjusted accordingly (finding correct replacements for the types mentioned in the first column of the table).

NOTE For geometry typed properties that are not mapped to the "geometry" top-level member, a suitable mapping needs to be defined, as explained in External types.

Primary geometry

identifier

http://www.opengis.net/spec/uml2json/1.0/req/geojson/primary-geometry

If a feature type has a primary geometry property (identified following the rules in Primary geometry), and that property is directly owned by the feature type, and the JSON Schema implementation of the property type is one of the GeoJSON geometry schemas (i.e., one of the JSON Schema references listed in JSON Schema implementation of types defined by ISO 19107, for the "geometry" member in the GeoJSON encoding rule), then:

- In the JSON Schema definition of the feature type, the primary geometry property shall be encoded as a type restriction for the top-level "geometry" member. If the primary geometry property is optional, then the schema restriction for the "geometry" member shall define a choice - using the "oneOf" JSON Schema keyword - between a null value and the geometry schema definition for the value type of the geometry property. The primary geometry property shall not be encoded within the "properties" member.
- In instance data, the value of the primary geometry property shall be encoded within the top-level "geometry" member of the JSON object that represents the feature type.

NOTE UML properties of other kinds of classes - object types, data types, and unions - are not considered by this requirement. Object types are not encoded as GeoJSON features. Data types and unions may be used by other classes, which prevents a general exclusive mapping to the GeoJSON top-level "geometry" member. Only a direct property of a «FeatureType» can be mapped in this way.

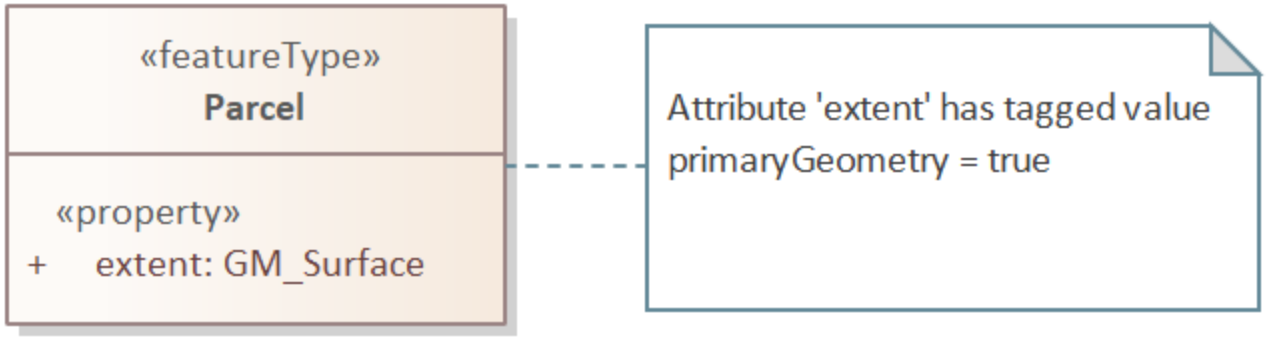

Figure 15. Example of a feature type with an attribute designated as primary geometry

Example for encoding a feature type with primary geometry

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Parcel": {
      "$anchor": "Parcel",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
          "type": "object",
          "properties": {
            "geometry": {
              "$ref": "https://geojson.org/schema/Polygon.json"
            }
          }
        }
      ]
    }
  }
}
```

Requirements class: Encoding rule for a JSON-FG compliant JSON Schema format

identifier

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/geojson-formats

inherit

http://www.opengis.net/spec/json-fg-1/0.1/req/core

requirement

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/common-base

requirement

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/iso19107-types-for-place-member

requirement

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/primary-geometry

requirement

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/primary-temporal-information

NOTE	An example of an application schema encoded in JSON-FG compliant JSON Schema format is given in Example schema in JSON-FG-compliant encoding.

Common base schema

As described in the core requirements class, common base types or - for the purposes of this encoding rule - common JSON Schema definitions can be added to the schema definition of certain kinds of classes, for example, all feature types.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/common-base

statement

All feature types shall use the JSON-FG Feature definition - https://beta.schemas.opengis.net/json-fg/feature.json - as common base.

part

The relationship to the JSON-FG Feature definition schema shall be implemented by converting a feature type to a JSON Schema that consists of an "allOf" with two subschemas: the first being a "$ref" with value "https://beta.schemas.opengis.net/json-fg/feature.json", the second being the schema produced by applying the other conversion rules to the feature type. However, if one of the supertypes of the feature type already has the JSON-FG Feature definition in its JSON Schema definition, then the JSON Schema definition of the feature type itself shall not define it again.

part

If the feature type is encoded with an "allOf" for the JSON-FG Feature definition, then the "$anchor" member (see Class name) shall be encoded in the schema that contains the "allOf", instead of within the second subschema.

NOTE	No common base schema is defined for object types. Such types need to be transformed to feature types if they should be encoded as JSON-FG features.

An example of encoding feature types with a common base schema is given in Common base schema. It can easily be adapted to match [http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/common-base] by exchanging "https://geojson.org/schema/Feature.json" with "https://beta.schemas.opengis.net/json-fg/feature.json".

Implementation of ISO 19107 types for the "place" member

identifier

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/iso19107-types-for-place-member

statement

If a UML property is encoded in JSON Schema, in the "place" top-level member of a JSON object that represents a type with identity, and the value type is one of the ISO 19107 geometry types listed in the first column of JSON Schema implementation of types defined by ISO 19107, for the "place" member, then the JSON schema definition in the second column of that table shall be used in the JSON Schema definition of the property.

Table 10. JSON Schema implementation of types defined by ISO 19107, for the "place" member

UML class	JSON Schema reference
GM_Point	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/Point
GM_Curve	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/LineString
GM_Surface	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/Polygon
GM_Solid	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/Polyhedron
GM_MultiPoint	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/MultiPoint
GM_MultiCurve	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/MultiLineString
GM_MultiSurface	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/MultiPolygon
GM_MultiSolid	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/MultiPolyhedron
GM_Aggregate	https://beta.schemas.opengis.net/json-fg/geometry-objects.json#/$defs/GeometryCollection
GM_Object	https://beta.schemas.opengis.net/json-fg/geometry.json

NOTE JSON Schema implementation of types defined by ISO 19107, for the "place" member uses geometry types defined by [ISO 19107:2003]. While this specification does not define mapping tables for newer versions of ISO 19107, application schemas may use geometry types from a newer version of ISO 19107. The mappings would then need to be adjusted accordingly (finding correct replacements for the types mentioned in the first column of the table).

NOTE For geometry typed properties that are not mapped to the "place" top-level member, a suitable mapping needs to be defined, as explained in External types.

Primary geometry

identifier

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/primary-geometry

If a feature type has a primary geometry property (identified following the rules in Primary geometry), and that property is directly owned by the feature type, and the JSON Schema implementation of the property type is one of the JSON-FG geometry schemas (i.e., one of the JSON Schema references listed in JSON Schema implementation of types defined by ISO 19107, for the "place" member), then:

- In the JSON Schema definition of the feature type, the primary geometry property shall be encoded as a type restriction for the top-level "place" member. The schema restriction for the "place" member shall define a choice - using the "oneOf" JSON Schema keyword - between a null value and the geometry schema definition for the value type of the UML property. The primary geometry property shall not be encoded within the "properties" member.

NOTE In instance data, the value of such a property is typically encoded within the (JSON-FG) top-level "place" member of the JSON object that represents the «FeatureType». However, there can also be cases where the value is encoded in the top-level "geometry" member. For further details, see [OGC 21-045], section "7.5 Geometry".

NOTE UML properties of other kinds of classes - object types, data types, and unions - are not considered by this requirement. Object types are not encoded as JSON-FG features. Data types and unions may be used by other classes, which prevents a general exclusive mapping to the JSON-FG top-level "place" member. Only a direct property of a «FeatureType» can be mapped in this way.

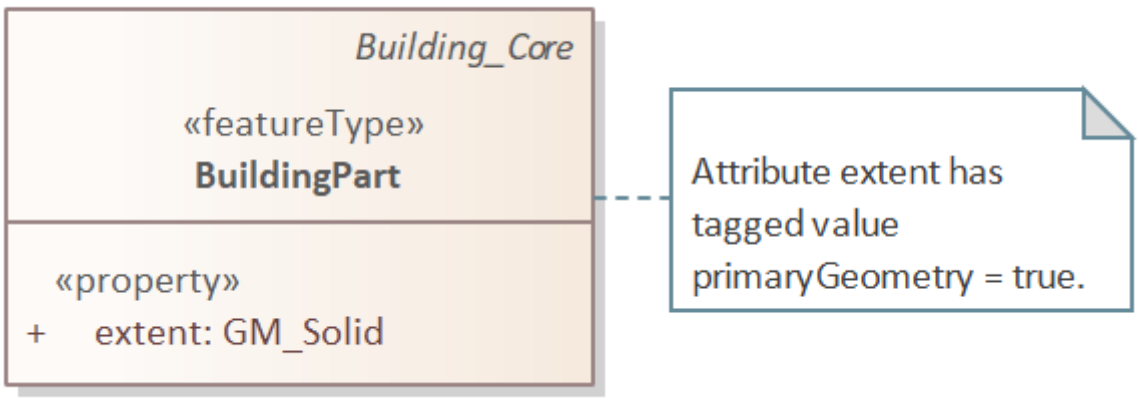

Figure 16. Example of a feature type with an attribute designated as primary place

Example for encoding a feature type with primary place

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "BuildingPart": {
      "$anchor": "BuildingPart",
      "allOf": [
        {
          "$ref": "http://example.org/schema/infra.json#Building_Core"
        },
        {
          "type": "object",
          "properties": {
            "place": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref": "https://beta.schemas.opengis.net/json-fg/geometry-
objects.json#/$defs/Polyhedron"
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

Primary temporal information

identifier

http://www.opengis.net/spec/uml2json/1.0/req/jsonfg/primary-temporal-information

In the JSON Schema definition of a feature, the primary-instant, primary-interval, primary-interval-start, and primary-interval-end properties (identified following the rules in Primary temporal information) shall not be encoded within the "properties" member.

In instance data, the value of such a property shall be encoded within the (JSON-FG) "time" member of the JSON object that represents the feature type.

NOTE

UML properties of other kinds of classes - object types, data types, and unions - are not considered by this requirement. Object types are not encoded as JSON-FG features. Data types and unions may be used by other classes, which prevents a general exclusive mapping to the JSON-FG top-level "time" member. Only a direct property of a feature type can be mapped in this way.

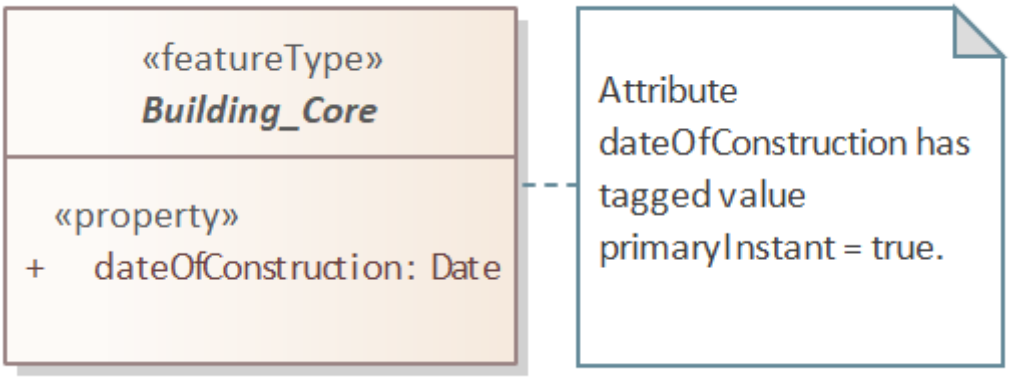

Figure 17. Example of a feature type with an attribute designated as primary instant

Example for encoding a feature type with primary instant

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Building_Core": {
      "$anchor": "Building_Core",
      "allOf": [
        {
          "$ref": "https://beta.schemas.opengis.net/json-fg/feature.json"
        },
        {
          "type": "object"
        }
      ]
    }
  }
}
```

Additional requirements classes for the by-reference encoding of property values

Overview

Requirements class [http://www.opengis.net/spec/uml2json/1.0/req/core] specifies an inline encoding of property values. In the case that the value type of a UML property is a type with identity (that is not implemented as a simple JSON Schema type), it can be preferable and maybe even necessary to encode the value by reference. In other cases, both options should be offered.

That is similar to what the GML Application Schema encoding rules support (for further details, see OGC 07-036r1, Annex E, section E.2.4.11).

NOTE An example where a reference to an object is needed, is when the object is the value of properties from multiple other objects that are encoded within the same JSON document. For example, a feature referenced from several other features. In such a situation, it is often desirable not to encode the object inline multiple times - especially if that object also references other objects.

NOTE Some applications may prefer to reference types with identity using a code (of type string or number) instead of using a URI. That code could be seen as a foreign key. In such cases, a model transformation should be applied first, which, for all relevant properties whose value type is a type with identity, replaces the value type with *CharacterString* or *Number*.

Multiple options exist for realizing the by-reference encoding of property values. A requirements class is available for each option:

- [http://www.opengis.net/spec/uml2json/1.0/req/by-reference-uri] - by-reference encoding of property values using a plain URI (reference, i.e., an absolute or relative URI)
- [http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object] - by-reference encoding of property values using a link object

NOTE The conversion behavior does not support by reference encoding for value types that are data types. In general, a data type does not have identity, and therefore a data type value should always be encoded inline, not by reference.

Requirements class: basics for the by-reference encoding of property values

identifier

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-basic

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

requirement

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-basic/inline-or-by-reference-tag

identifier

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-basic/inline-or-by-reference-tag

statement

For a UML property, whose value type is a type with identity that is not implemented as a simple JSON Schema type, the tag *inlineOrByReference*, if set, shall have one of three values: *inlineOrByReference*, *byReference*, or *inline*. If the tag is not set on a UML property, or has an empty value, then the following value shall be assumed as default value:

- *inline*, in case that the UML property is an attribute
- *byReference*, in case that the UML property is an association role

NOTE

The default value for tag *inlineOrByReference* is different in GML. For the JSON Schema encoding, the default values have been chosen in order to reduce the degrees of freedom and to reduce the schema complexity. The separation into default value *inline* for UML attributes, and *byReference* for UML association roles has been made since that reflects the typical modeling approach, where association roles have a value type that is usually encoded by reference, and attributes have a value type that is usually encoded inline - especially if the attribute value type is a type with identity (e.g., an ISO 19107 geometry type).

Requirements class: by-reference encoding of property values using a plain URI reference

identifier

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-uri

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-basic

requirement

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-uri/encoding

identifier

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-uri/encoding

statement

If the value of tag *inlineOrByReference* of a UML property - whose value type is a type with identity that is not implemented as a simple JSON Schema type - is not *inline*:

part

If the tag value is *byReference*, then the JSON Schema definition of the property shall contain a "type" member with value "string", as well as a "format" member with value "uri-reference";

part

Otherwise - the tag value is *inlineOrByReference* - the inline and by-reference encoding cases shall be combined in the JSON Schema definition of the property using the "oneOf" keyword.

NOTE The result is an XOR type of check, i.e., a value can either be given inline or by reference, but not both. This is different to GML, where in the case of *inlineOrByReference* and a type with identity as value type, a value can be encoded both inline and by reference.

Figure 18. Example of an association between two feature types, where the association roles are to be encoded by reference

The JSON Schema encoding for the example in Example of an association between two feature types, where the association roles are to be encoded by reference, using URIs to realize by-reference encoding of property values, is given in Example for encoding association roles by-reference using URIs.

Example for encoding association roles by-reference using URIs

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Parcel": {
      "$anchor": "Parcel",
      "type": "object",
      "properties": {
        "owner": {
          "type": "array",
          "minItems": 1,
          "items": {
            "type": "string",
            "format": "uri-reference"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "owner"
      ]
    },
    "Person": {
      "$anchor": "Person",
      "type": "object",
      "properties": {
        "owns": {
          "type": "array",
          "items": {
            "type": "string",
            "format": "uri-reference"
          },
          "uniqueItems": true
        }
      }
    }
  }
}
```

This JSON object is valid against the schema definition of "Parcel" from Example for encoding association roles by-reference using URIs:

```
{
  "owner": ["http://example.org/Person/d024i42sl"]
}
```

Requirements class: by-reference encoding of property values using a link object

identifier

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-basic

requirement

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object/encoding

identifier

http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object/encoding

statement

If the value of tag *inlineOrByReference* of a UML property - whose value type is a type with identity that is not implemented as a simple JSON Schema type - is not *inline*:

part

If the tag value is *byReference*, then the JSON Schema definition of the property shall contain a "$ref" member with value "https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$defs/LinkObject" (the JSON Schema for that link object is defined in JSON Schema definitions);

part

Otherwise - the tag value is *inlineOrByReference* - the inline and by-reference encoding cases shall be combined in the JSON Schema definition of the property using the "oneOf" keyword.

NOTE The result is an XOR type of check, i.e., a value can either be given inline or by reference, but not both. This is different to GML, where in the case of *inlineOrByReference* and a type with identity as value type, a value can be encoded both inline and by reference.

The JSON Schema encoding for the example in Example of an association between two feature types, where the association roles are to be encoded by reference, using link objects to realize by-reference encoding of property values, is given in Example for encoding association roles by-reference using link objects.

Example for encoding association roles by-reference using link objects

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Parcel": {
      "$anchor": "Parcel",
      "type": "object",
      "properties": {
        "owner": {
          "type": "array",
          "minItems": 1,
          "items": {
            "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "owner"
      ]
    },
    "Person": {
      "$anchor": "Person",
      "type": "object",
      "properties": {
        "owns": {
          "type": "array",
          "items": {
            "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
          },
          "uniqueItems": true
        }
      }
    }
  }
}
```

This JSON object is valid against the schema definition of "Parcel" from Example for encoding association roles by-reference using link objects:

```
{
  "owner": [
    {
      "title": "John Doe",
      "href": "http://example.org/Person/d024i42sl"
    }
  ]
}
```

Additional requirements classes for the encoding of union types

Overview

Application schemas have two ways of using types with stereotype «union».

- According to ISO 19103:2015, a «union» type consists *"of one and only one of several alternative datatypes (listed as member attributes). This is similar to a discriminated union in many programming languages"*. According to this definition, only the types of the UML attributes defined for a «union» are of interest.
- In practice, unions defined in application schemas are also used differently, defining a choice between a number of options, where each option is modeled as a UML attribute. In other words, the attribute itself has meaning - not just its value type. Multiple options can have the same value type. Options can have different maximum multiplicity (especially greater than 1). The UML-to-GML application schema encoding rules support this way of using unions (see OGC 07-036r1, section E.2.4.10).

The following sections document requirements classes for union encodings that support these two approaches.

Requirements class: JSON Schema encoding for unions representing type discriminators

identifier

http://www.opengis.net/spec/uml2json/1.0/req/union-type-discriminator

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

requirement

http://www.opengis.net/spec/uml2json/1.0/req/union-type-discriminator/encoding

identifier

http://www.opengis.net/spec/uml2json/1.0/req/union-type-discriminator/encoding

A «union» shall be encoded as a JSON Schema definition that represents a choice between the value types of the union properties.

- If the value types are only simple, without a specific format definition or other restrictions defined by JSON Schema keywords, then the JSON Schema shall only contain a "type" member, with an array of the simple types.
- Otherwise, a "oneOf" member shall be added to the JSON Schema definition, with:
  - one "$ref" per non-simple type,
  - one "type" for all simple types without specific keywords, and
  - one "type" per simple type with specific keywords.

The result of applying this encoding to the unions from Example of type discriminator unions is shown in Example of a JSON Schema for unions, encoding them as type discriminators.

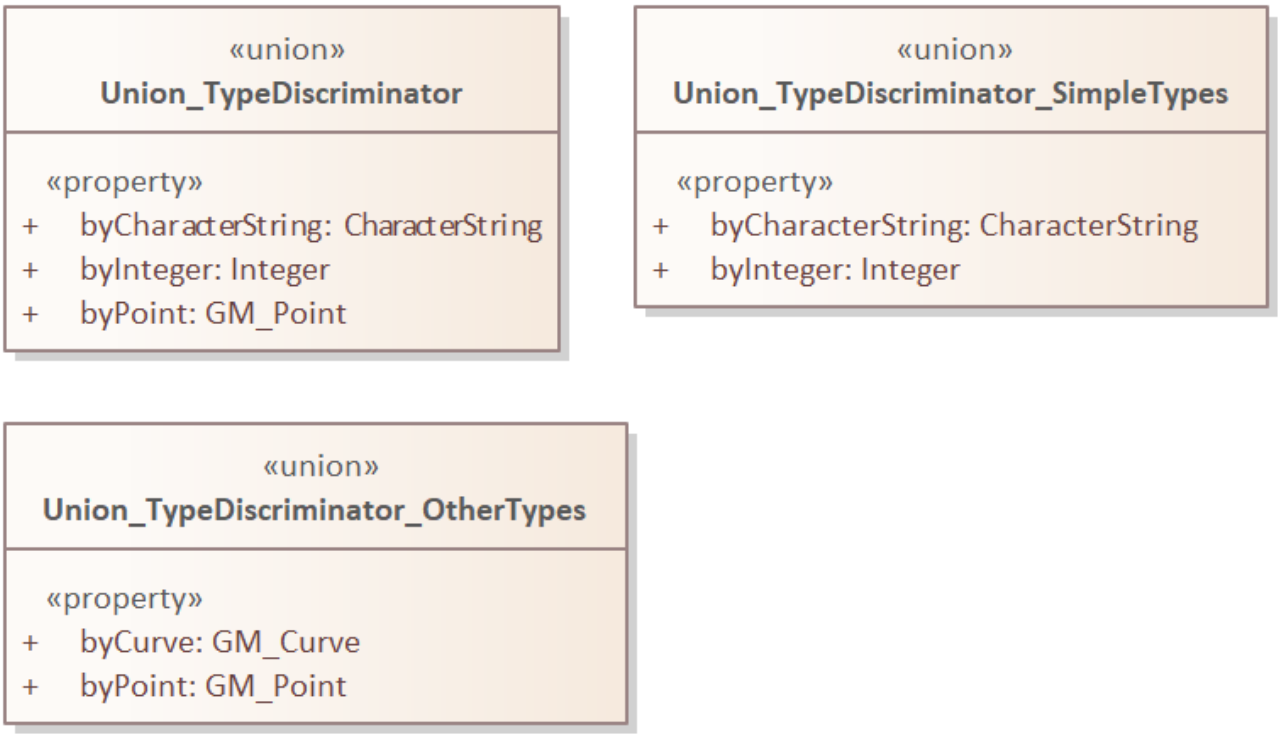

Figure 19. Example of type discriminator unions

Example of a JSON Schema for unions, encoding them as type discriminators

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Union_TypeDiscriminator": {
      "oneOf": [
        {
          "type": [
            "string",
            "integer"
          ]
        },
        {
          "$ref": "https://geojson.org/schema/Point.json"
        }
      ]
    },
    "Union_TypeDiscriminator_OtherTypes": {
      "oneOf": [
        {
          "$ref": "https://geojson.org/schema/LineString.json"
        },
        {
          "$ref": "https://geojson.org/schema/Point.json"
        }
      ]
    },
    "Union_TypeDiscriminator_SimpleTypes": {
      "type": [
        "string",
        "integer"
      ]
    }
  }
}
```

WARNING

Care must be taken, that the type choices of a type discriminator union are separate value spaces. Otherwise, validation may fail, if an actual value matches more than one of the type choices. That would break the rule of the "oneOf" JSON Schema keyword, that one and only one of its component schemas is satisfied. If, for example, the "Union_TypeDiscriminator" in Example of a JSON Schema for unions, encoding them as type discriminators had another option for a date value ("type": "string", "format": "date"), and value "2022-12-09" was validated against the resulting JSON Schema definition, validation would fail - because that value matches both the string-or-number case and the string-with-format-date case.

Requirements class: JSON Schema encoding for unions representing property choices

identifier

http://www.opengis.net/spec/uml2json/1.0/req/union-property-choice

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

requirement

http://www.opengis.net/spec/uml2json/1.0/req/union-property-choice/encoding

identifier

http://www.opengis.net/spec/uml2json/1.0/req/union-property-choice/encoding

part

A «union» shall be encoded as a JSON Schema definition of a JSON object, where each union option is represented as an optional member of the JSON object.

part

The choice between the options defined by the union shall be encoded using "maxProperties" = "minProperties" = 1. That is, the number of members that are allowed for the JSON object is restricted to exactly one.

part

An `"additionalProperties": false` shall be used to prevent any undefined properties.

The result of applying this encoding to the union from «union» example is shown in Example of a JSON Schema for a «union» class, representing the property choice using "minProperties" and "maxProperites".

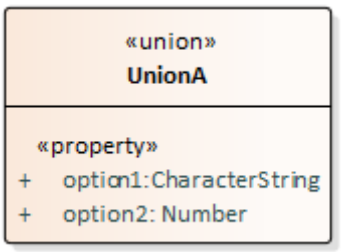

Figure 20. «union» example

Example of a JSON Schema for a «union» class, representing the property choice using "minProperties" and "maxProperites"

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "UnionA": {
      "type": "object",
      "properties": {
        "option1": {
          "type": "string"
        },
        "option2": {
          "type": "number"
        }
      },
      "additionalProperties": false,
      "minProperties": 1,
      "maxProperties": 1
    }
  },
  "$ref": "#/$defs/UnionA"
}
```

NOTE	An alternative approach would be using the "oneOf" keyword, with one subschema per union property, which only defines that property, and requires it (but does not perform any other checks). This option is more verbose, harder to read and understand and, therefore, not recommended.

This JSON object is valid against the schema:

```
{
  "option1": "x"
}
```

This JSON object is invalid (because "option2" has a string value, rather than a numeric value) against the schema:

```
{
  "option2": "x"
}
```

Additional requirements classes for the encoding of code list types

Overview

This specification defines three approaches for encoding the values of properties that have a «CodeList» as value type:

- using a simple literal value, e.g., a string or number that represents a code,
- using a URI as code value, and
- using a link object to link to a code representation.

The following sections document requirements classes for code list encodings that support these three approaches. All of them inherit requirements from a common requirements class, which is defined in Requirements class: Basic JSON Schema encoding for code lists.

Requirements class: Basic JSON Schema encoding for code lists

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

requirement

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic/schema-definition

requirement

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic/codelist-tag

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic/schema-definition

statement

A «CodeList» shall be converted to a JSON Schema definition of a JSON object. That definition shall be added to the definitions schema, using the type name as definition key.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic/codelist-tag

statement

If the «CodeList» has tag *codeList* (which is defined by [ISO 19103:2015]), with a non-blank value, then a "codeList" member shall be added to the JSON Schema definition of the code list, with the tag value as value.

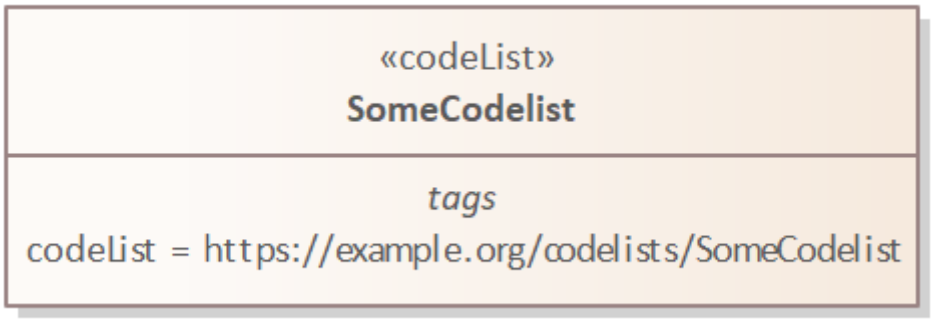

Figure 21. Example of a «CodeList» type with tagged value 'codeList'

Example of the basic JSON Schema encoding of a «CodeList» type with tagged value 'codeList'

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "SomeCodelist": {
      ...
      "codeList": "http://example.org/codelists/SomeCodelist",
      ...
    }
  }
}
```

Requirements class: JSON Schema encoding for code lists - literal

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-literal

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic

requirement

http://www.opengis.net/spec/uml2json/1.0/req/codelists-literal/type

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-literal/type

statement

The JSON Schema definition of a «CodeList» shall have a "type" member defined by evaluating tagged value *literalEncodingType*. The tagged value *literalEncodingType* identifies the conceptual type that applies to the code values. If the tagged value is not set on the code list, or has an empty value, then the literal encoding type is defined to be CharacterString.

The literal encoding type is one of the types from ISO 19103, which are implemented as a simple JSON Schema type - see Literal encoding type in Enumeration.

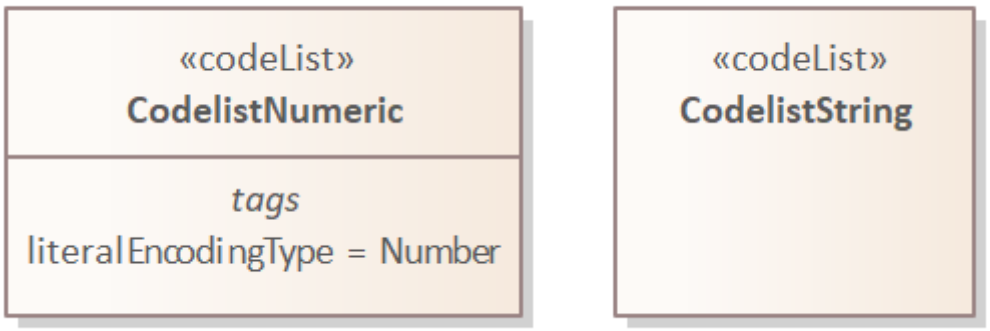

Figure 22. Example of «CodeList» types

Example of the JSON Schema encodings of «CodeList» types

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "CodelistNumeric": {
      "type": "number"
    },
    "CodelistString": {
      "type": "string"
    }
  }
}
```

Requirements class: JSON Schema encoding for code lists - URI

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-uri

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic

requirement

http://www.opengis.net/spec/uml2json/1.0/req/codelists-uri/type

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-uri/type

statement

The JSON Schema definition of a «CodeList» shall have a "type" member with value "string", as well as a "format" member with value "uri".

Example of the JSON Schema encodings of a «CodeList» type with the JSON Schema "type" being a URI

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "CodelistUriFormat": {
      "type": "string",
      "format": "uri"
    }
  }
}
```

Requirements class: JSON Schema encoding for code lists - link object

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-link-object

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/codelists-basic

requirement

http://www.opengis.net/spec/uml2json/1.0/req/codelists-link-object/schema-ref

identifier

http://www.opengis.net/spec/uml2json/1.0/req/codelists-link-object/schema-ref

statement

The JSON Schema definition of a «CodeList» shall have a "$ref" member with value "https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$defs/LinkObject".

That means that a code value is essentially encoded as a "link object" as specified by IETF RFC 8288 and implemented in the OGC API standards. The link object provides "href" and "title" members like the simple Xlinks in GML.

Requirements class: Encoding of an additional entityType property

identifier

http://www.opengis.net/spec/uml2json/1.0/req/entitytype

target

JSON (Schema) documents

inherit

http://www.opengis.net/spec/uml2json/1.0/req/core

requirement

http://www.opengis.net/spec/uml2json/1.0/req/entitytype/member

permission

http://www.opengis.net/spec/uml2json/1.0/req/entitytype/json-fg-feature-type

identifier

http://www.opengis.net/spec/uml2json/1.0/req/entitytype/member

part

If the class C that is being converted is a feature type, an object type, or a data type, then the JSON member "entityType" shall be added to the properties of the JSON object that represents the class. If, however, the JSON Schema encoding of any of the potentially existing supertypes of the class already defines the "entityType" member, then the "entityType" member shall not be added to the JSON representation of class C.

part

The "entityType" member shall be required and string-valued.

part

The "entityType" member shall be used to encode the name of the conceptual type (i.e., the class) that is represented by the JSON object.

identifier

http://www.opengis.net/spec/uml2json/1.0/req/entitytype/json-fg-feature-type

statement

For a feature type that is encoded as a JSON-FG feature, the "entityType" member may be omitted. That is due to the fact that a JSON-FG feature already has a top-level "featureType" member, which serves the same purpose as the "entityType" member.

NOTE

By default, the property value is not restricted using "const", because doing so would prevent JSON Schema constraints that support inheritance-related checks. However, if the application schema did not use inheritance, then such restrictions could be defined.

NOTE

Encoding the type name in JSON objects can be useful, since, as described in chapter 6 of the OGC Testbed-14: Application Schemas and JSON Technologies Engineering Report, having a key within a JSON object with a string value that identifies the type of the object allows that object to be mapped to RDF. More specifically, the string value can be mapped to an IRI that identifies the type of an RDFS resource.

NOTE

Encoding a JSON object that represents an abstract type, with the "entityType" having the abstract type name as value, would be useful with regards to linked data applications, and conversion of JSON data to RDF using JSON-LD. Abstractness is also not supported in RDF/OWL, so RDF resources can define the RDFS/OWL class or datatype, which represent an abstract type from the conceptual model, as their type. That makes sense for cases in which the exact type of a resource or "thing" is not known yet, but a more general type is.

JSON Schema example with property "entityType" used for identifying the type of a JSON object

```
{
  "$schema": "http://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "Person": {
      "properties": {
        "entityType": {
          "type": "string"
        },
        "name": {
          "type": "string"
        }
      },
      "required": [
        "entityType", "name"
      ]
    }
  },
  "$ref": "#/$defs/Person"
}
```

The following JSON instance is valid against the schema:

```
{
  "entityType": "Person",
  "name": "John Doe"
}
```

Annex A: Conformance Class Abstract Test Suite

NOTE	Conformance classes can be defined when this specification moves on in the OGC standardization process.

Annex B: Example application schema

Overview

This Annex illustrates the results of applying the JSON Schema encoding rules on the application schema example shown in Example application schema in UML.

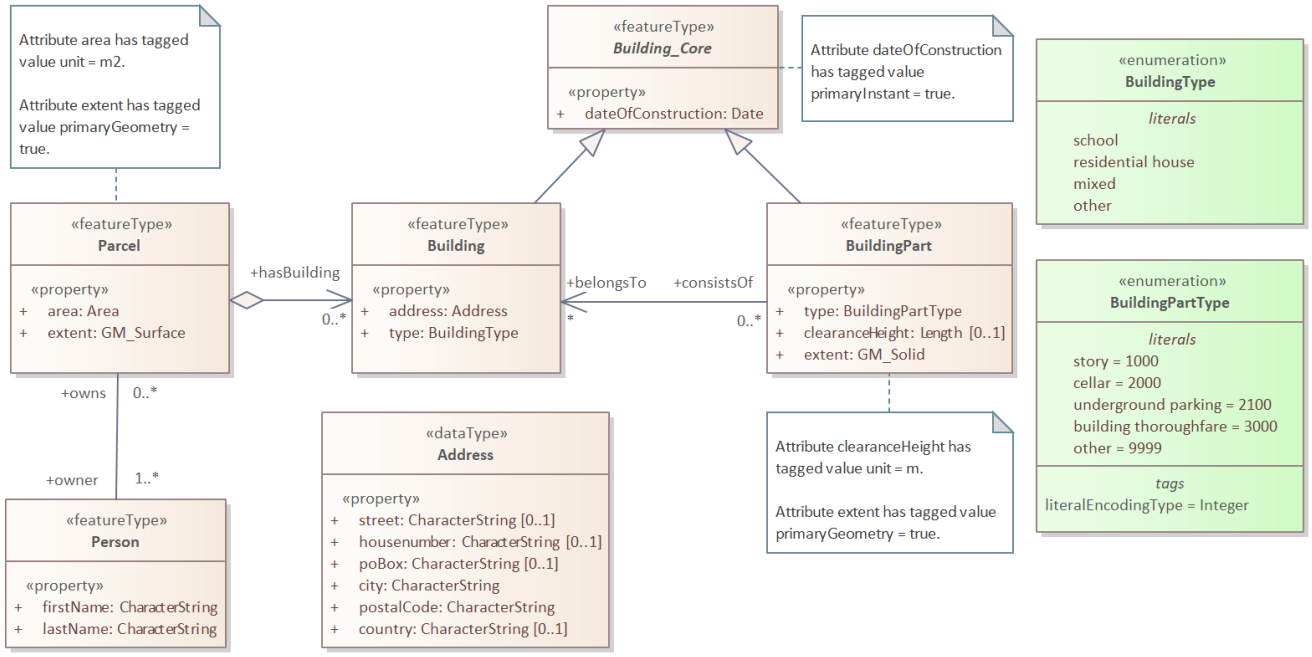

Figure 23. Example application schema in UML

NOTE The schema uses fixed units of measure for BuildingPart.clearanceHeight and Parcel.area. The units are defined in tagged value *unit*, using UCUM codes. BuildingPart.clearanceHeight has unit "m" (meter), and Parcel.area has unit "m2" (square meter).

Example schema in plain JSON encoding

The JSON Schema shown in JSON Schema for the example application schema - plain JSON encoding was created by applying requirements class [http://www.opengis.net/spec/uml2json/1.0/req/plain] and [http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object].

JSON Schema for the example application schema - plain JSON encoding

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/schema/infra.json",
  "$defs": {
    "Address": {
      "$anchor": "Address",
      "type": "object",
      "properties": {
        "street": {
          "type": "string"
        },
        "housenumber": {
          "type": "string"
        },
        "poBox": {
          "type": "string"
```

```
        },
        "city": {
          "type": "string"
        },
        "postalCode": {
          "type": "string"
        },
        "country": {
          "type": "string"
        }
      },
      "required": [
        "city",
        "postalCode"
      ]
    },
    "Building": {
      "$anchor": "Building",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object",
          "properties": {
            "address": {
              "$ref": "#/$defs/Address"
            },
            "type": {
              "$ref": "#/$defs/BuildingType"
            }
          },
          "required": [
            "address",
            "type"
          ]
        }
      ]
    },
    "BuildingPart": {
      "$anchor": "BuildingPart",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object",
          "properties": {
            "type": {
              "$ref": "#/$defs/BuildingPartType"
            },
```

```
            "clearanceHeight": {
              "type": "number",
              "unit": "m"
            },
            "extent": {
              "$ref": "https://beta.schemas.opengis.net/json-fg/geometry-
objects.json#/$defs/Polyhedron"
            },
            "belongsTo": {
              "type": "array",
              "items": {
                "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
              },
              "uniqueItems": true
            }
          },
          "required": [
            "extent",
            "type"
          ]
        }
      ]
    },
    "BuildingPartType": {
      "$anchor": "BuildingPartType",
      "type": "integer",
      "enum": [
        1000,
        2000,
        2100,
        3000,
        9999
      ]
    },
    "BuildingType": {
      "$anchor": "BuildingType",
      "type": "string",
      "enum": [
        "school",
        "residential house",
        "mixed",
        "other"
      ]
    },
    "Building_Core": {
      "$anchor": "Building_Core",
      "type": "object",
      "properties": {
        "dateOfConstruction": {
```

```
          "type": "string",
          "format": "date",
          "pattern": "^\\d{4}-\\d{2}-\\d{2}$"
        }
      },
      "required": [
        "dateOfConstruction"
      ]
    },
    "Parcel": {
      "$anchor": "Parcel",
      "type": "object",
      "properties": {
        "area": {
          "type": "number",
          "unit": "m2"
        },
        "extent": {
          "$ref": "https://geojson.org/schema/Polygon.json"
        },
        "hasBuilding": {
          "type": "array",
          "items": {
            "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
          },
          "uniqueItems": true
        },
        "owner": {
          "type": "array",
          "minItems": 1,
          "items": {
            "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "area",
        "extent",
        "owner"
      ]
    },
    "Person": {
      "$anchor": "Person",
      "type": "object",
      "properties": {
        "firstName": {
```

```
          "type": "string"
        },
        "lastName": {
          "type": "string"
        },
        "owns": {
          "type": "array",
          "items": {
            "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
          },
          "uniqueItems": true
        }
      },
      "required": [
        "firstName",
        "lastName"
      ]
    }
  }
}
```

Example schema in GeoJSON-compliant encoding

The JSON Schema shown in JSON Schema for the example application schema - GeoJSON-compliant encoding was created by applying requirements class [http://www.opengis.net/spec/uml2json/1.0/req/geojson] and [http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object].

JSON Schema for the example application schema - GeoJSON-compliant encoding

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/schema/infra.json",
  "$defs": {
    "Address": {
      "$anchor": "Address",
      "type": "object",
      "properties": {
        "street": {
          "type": "string"
        },
        "housenumber": {
          "type": "string"
        },
        "poBox": {
          "type": "string"
        },
        "city": {
```

```
          "type": "string"
        },
        "postalCode": {
          "type": "string"
        },
        "country": {
          "type": "string"
        }
      },
      "required": [
        "city",
        "postalCode"
      ]
    },
    "Building": {
      "$anchor": "Building",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object",
          "properties": {
            "properties": {
              "type": "object",
              "properties": {
                "address": {
                  "$ref": "#/$defs/Address"
                },
                "type": {
                  "$ref": "#/$defs/BuildingType"
                }
              },
              "required": [
                "address",
                "type"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    },
    "BuildingPart": {
      "$anchor": "BuildingPart",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
```

```
        {
          "type": "object",
          "properties": {
            "properties": {
              "type": "object",
              "properties": {
                "type": {
                  "$ref": "#/$defs/BuildingPartType"
                },
                "clearanceHeight": {
                  "type": "number",
                  "unit": "m"
                },
                "extent": {
                  "$ref": "https://beta.schemas.opengis.net/json-fg/geometry-
objects.json#/$defs/Polyhedron"
                },
                "belongsTo": {
                  "type": "array",
                  "items": {
                    "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                  },
                  "uniqueItems": true
                }
              },
              "required": [
                "extent",
                "type"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    },
    "BuildingPartType": {
      "$anchor": "BuildingPartType",
      "type": "integer",
      "enum": [
        1000,
        2000,
        2100,
        3000,
        9999
      ]
    },
    "BuildingType": {
```

```
      "$anchor": "BuildingType",
      "type": "string",
      "enum": [
        "school",
        "residential house",
        "mixed",
        "other"
      ]
    },
    "Building_Core": {
      "$anchor": "Building_Core",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
          "type": "object",
          "properties": {
            "properties": {
              "type": "object",
              "properties": {
                "dateOfConstruction": {
                  "type": "string",
                  "format": "date",
                  "pattern": "^\\d{4}-\\d{2}-\\d{2}$"
                }
              },
              "required": [
                "dateOfConstruction"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    },
    "Parcel": {
      "$anchor": "Parcel",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
          "type": "object",
          "properties": {
            "geometry": {
              "$ref": "https://geojson.org/schema/Polygon.json"
            },
            "properties": {
```

```
              "type": "object",
              "properties": {
                "area": {
                  "type": "number",
                  "unit": "m2"
                },
                "hasBuilding": {
                  "type": "array",
                  "items": {
                    "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                  },
                  "uniqueItems": true
                },
                "owner": {
                  "type": "array",
                  "minItems": 1,
                  "items": {
                    "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                  },
                  "uniqueItems": true
                }
              },
              "required": [
                "area",
                "owner"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    },
    "Person": {
      "$anchor": "Person",
      "allOf": [
        {
          "$ref": "https://geojson.org/schema/Feature.json"
        },
        {
          "type": "object",
          "properties": {
            "properties": {
              "type": "object",
              "properties": {
                "firstName": {
```

```
                  "type": "string"
                },
                "lastName": {
                  "type": "string"
                },
                "owns": {
                  "type": "array",
                  "items": {
                    "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                  },
                  "uniqueItems": true
                }
              },
              "required": [
                "firstName",
                "lastName"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    }
  }
}
```

Example schema in JSON-FG-compliant encoding

The JSON Schema shown in JSON Schema for the example application schema - JSON-FG-compliant encoding was created by applying requirements class [http://www.opengis.net/spec/uml2json/1.0/req/jsonfg] and [http://www.opengis.net/spec/uml2json/1.0/req/by-reference-link-object].

JSON Schema for the example application schema - JSON-FG-compliant encoding

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "http://example.org/schema/infra.json",
  "$defs": {
    "Address": {
      "$anchor": "Address",
      "type": "object",
      "properties": {
        "street": {
          "type": "string"
        },
```

```
        "housenumber": {
          "type": "string"
        },
        "poBox": {
          "type": "string"
        },
        "city": {
          "type": "string"
        },
        "postalCode": {
          "type": "string"
        },
        "country": {
          "type": "string"
        }
      },
      "required": [
        "city",
        "postalCode"
      ]
    },
    "Building": {
      "$anchor": "Building",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object",
          "properties": {
            "properties": {
              "type": "object",
              "properties": {
                "address": {
                  "$ref": "#/$defs/Address"
                },
                "type": {
                  "$ref": "#/$defs/BuildingType"
                }
              },
              "required": [
                "address",
                "type"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
```

```
    },
    "BuildingPart": {
      "$anchor": "BuildingPart",
      "allOf": [
        {
          "$ref": "#/$defs/Building_Core"
        },
        {
          "type": "object",
          "properties": {
            "place": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref": "https://beta.schemas.opengis.net/json-fg/geometry-
objects.json#/$defs/Polyhedron"
                }
              ]
            },
            "properties": {
              "type": "object",
              "properties": {
                "type": {
                  "$ref": "#/$defs/BuildingPartType"
                },
                "clearanceHeight": {
                  "type": "number",
                  "unit": "m"
                },
                "belongsTo": {
                  "type": "array",
                  "items": {
                    "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                  },
                  "uniqueItems": true
                }
              },
              "required": [
                "type"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
```

```
    },
    "BuildingPartType": {
      "$anchor": "BuildingPartType",
      "type": "integer",
      "enum": [
        1000,
        2000,
        2100,
        3000,
        9999
      ]
    },
    "BuildingType": {
      "$anchor": "BuildingType",
      "type": "string",
      "enum": [
        "school",
        "residential house",
        "mixed",
        "other"
      ]
    },
    "Building_Core": {
      "$anchor": "Building_Core",
      "allOf": [
        {
          "$ref": "https://beta.schemas.opengis.net/json-fg/feature.json"
        },
        {
          "type": "object"
        }
      ]
    },
    "Parcel": {
      "$anchor": "Parcel",
      "allOf": [
        {
          "$ref": "https://beta.schemas.opengis.net/json-fg/feature.json"
        },
        {
          "type": "object",
          "properties": {
            "place": {
              "oneOf": [
                {
                  "type": "null"
                },
                {
                  "$ref": "https://beta.schemas.opengis.net/json-fg/geometry-
objects.json#/$defs/Polygon"
                }
```

```
            ]
          },
          "properties": {
            "type": "object",
            "properties": {
              "area": {
                "type": "number",
                "unit": "m2"
              },
              "hasBuilding": {
                "type": "array",
                "items": {
                  "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                },
                "uniqueItems": true
              },
              "owner": {
                "type": "array",
                "minItems": 1,
                "items": {
                  "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                },
                "uniqueItems": true
              }
            },
            "required": [
              "area",
              "owner"
            ]
          }
        },
        "required": [
          "properties"
        ]
      }
    ]
  },
  "Person": {
    "$anchor": "Person",
    "allOf": [
      {
        "$ref": "https://beta.schemas.opengis.net/json-fg/feature.json"
      },
      {
        "type": "object",
        "properties": {
          "properties": {
```

```
              "type": "object",
              "properties": {
                "firstName": {
                  "type": "string"
                },
                "lastName": {
                  "type": "string"
                },
                "owns": {
                  "type": "array",
                  "items": {
                    "$ref":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json#/$
defs/LinkObject"
                  },
                  "uniqueItems": true
                }
              },
              "required": [
                "firstName",
                "lastName"
              ]
            }
          },
          "required": [
            "properties"
          ]
        }
      ]
    }
  }
}
```

Annex C: JSON Schema definitions

The following JSON Schema defines the schema for link object and measure, which are used in this specification.

JSON Schema for link object and measure

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id":
"https://register.geostandaarden.nl/jsonschema/uml2json/0.1/schema_definitions.json",
  "$defs": {
    "LinkObject": {
      "$anchor": "LinkObject",
      "title": "link object",
      "description": "definition of a link object",
      "type": "object",
```

```
      "required": ["href"],
      "properties": {
        "href": {
          "type": "string",
          "description": "Supplies the URI to a remote resource (or resource
fragment).",
          "example": "http://data.example.com/buildings/123"
        },
        "rel": {
          "type": "string",
          "description": "The type or semantics of the relation.",
          "example": "related"
        },
        "type": {
          "type": "string",
          "description": "A hint indicating what the media type of the result of
dereferencing the link should be.",
          "example": "application/geo+json"
        },
        "hreflang": {
          "type": "string",
          "description": "A hint indicating what the language of the result of
dereferencing the link should be.",
          "example": "en"
        },
        "title": {
          "type": "string",
          "description": "Used to label the destination of a link such that it can be
used as a human-readable identifier.",
          "example": "Trierer Strasse 70, 53115 Bonn"
        },
        "length": {"type": "integer"}
      }
    },
    "Measure": {
      "$anchor": "Measure",
      "title": "measure object",
      "description": "definition of a measure object",
      "type": "object",
      "required": [
        "value",
        "uom"
      ],
      "properties": {
        "value": {"type": "number"},
        "uom": {"type": "string"}
      }
    }
  }
}
```

Annex D: JSON Schema definitions for collections

This specification defines requirements for the creation of schema definitions with the JSON Schema constraints to validate a JSON object that represents a type with identity. The specification does not define requirements (classes) with which to define the schema for collections of such types. There are different aspects to consider, when defining such collections. The following sections document considerations and ideas regarding these aspects, to be used as input for future discussion and possibly standardization work.

Scope and naming of collection definitions

In practice, uniform as well as mixed collections are possible. A uniform collection contains features of a single specific type, whereas a mixed collection can have objects of different type. JSON Schema definitions for uniform collections could be defined with name being `{type name} + Collection`, whereas a single definition with name FeatureCollection could cover the case of all mixed collections (at least for all types with identity defined by the application schema).

Structure of collection definitions

The definition for a uniform collection would ensure that the collection members are all valid against a single type definition.

The definition for the general FeatureCollection would use either an *anyOf* or *oneOf*, with a choice of schema definitions for all types with identity. Cases in which an object is valid against more than one of these definitions can be problematic:

- In case of *anyOf*, maybe the schema definition for a more general type matches, causing the validator to stop the validation process, thus potentially not checking constraints for the specific schema definition for a given type.
- In case of *oneOf*, the validator does not stop at the first matching schema definition, but continues validation against all options, until it has ensured that either all other definitions do not match, or one more does match. The latter case would result in the overall validation to fail. However, that can easily happen in case of similar class structures, or in case of inheritance hierarchies, where an object matches the schema defined by a supertype as well as that of a subtype.

Performance

It is unclear how performant the validation against the general FeatureCollection would be in practice, in case of a large or complex application schema. Then again, validation time may not be that critical to an application that wants to ensure that data within a collection is valid.

Abstract types

It is unclear if collection definitions should be created for or include abstract types. It may be useful to create such definitions for "uniform" collections of abstract types, even if that implies that actual collections may have objects of different subtypes of the abstract type. Definitions for abstract types should not be included in the general FeatureCollection.

Base schemas

Another aspect is which base schema to use for the collections. GeoJSON and JSON-FG both define schemas for feature collections (see https://geojson.org/schema/FeatureCollection.json and https://github.com/opengeospatial/ogc-feat-geo-json/blob/main/core/schemas/featurecollection.json). These schemas should be used as base in according encodings. For the plain JSON encoding, a simple JSON array could be used.

Requirements classes

Requirements for the generation of collections could be defined in the three encoding requirements classes. However, it may be better to create additional requirements classes that depend on them, since communities may want to create different schema definitions for collections. It would even be possible to have separate requirements classes for uniform collections and the general collection, since communities may or may not want to use both types of collections.

Annex E: Title ({Normative/Informative})

NOTE	Place other Annex material in sequential annexes beginning with "B" and leave final two annexes for the Revision History and Bibliography

Annex F: Revision History

Date	Release	Editor	Primary clauses modified	Description
2022-12-13	0.1	Johannes Echterhoff	all	initial version for internal review
2024-04-25	0.2	Johannes Echterhoff	all	solving a number of issues

Bibliography

- [OGC 07-036r1]Open Geospatial Consortium (OGC). OGC 07-036r1: OpenGIS Geography Markup

Language (GML) Encoding Standard, Version 3.2.2

- [OGC 20-012]Open Geospatial Consortium (OGC). OGC 20-012: UML-to-GML Application Schema Pilot (UGAS-2020) Engineering Report
- [OGC 23-058r1]Open Geospatial Consortium (OGC). OGC 23-058r1: OGC API - Features - Part 5: Schemas Implementation Specification