Common Workflow Language, v1.0

DOI:
10.6084/m9.figshare.3115156.v2

Document Version
Final published version

Semantic Annotations for Linked Avro Data (SALAD)

Author:

- Peter Amstutz peter.amstutz@curoverse.com (mailto:peter.amstutz@curoverse.com), Curoverse

Contributors:

- The developers of Apache Avro
- The developers of JSON-LD
- Nebojša Tijanić nebojsa.tijanic@sbgenomics.com (mailto:nebojsa.tijanic@sbgenomics.com), Seven Bridges Genomics

Abstract

Salad is a schema language for describing structured linked data documents in JSON or YAML documents. A Salad schema provides rules for preprocessing, structural validation, and link checking for documents described by a Salad schema. Salad builds on JSON-LD and the Apache Avro data serialization system, and extends Avro with features for rich data modeling such as inheritance, template specialization, object identifiers, and object references. Salad was developed to provide a bridge between the record oriented data modeling supported by Apache Avro and the Semantic Web.

Status of This Document

This document is the product of the Common Workflow Language working group (https://groups.google.com/forum/#!forum/common-workflow-language). The latest version of this document is available in the "schema_salad" directory at

https://github.com/common-workflow-language/schema_salad (https://github.com/common-workflow-language/schema_salad)

Table of contents

1. Introduction

The JSON data model is an extremely popular way to represent structured data. It is attractive because of it's relative simplicity and is a natural fit with the standard types of many programming languages. However, this simplicity means that basic JSON lacks expressive features useful for working with complex data structures and document formats, such as schemas, object references, and namespaces.

JSON-LD is a W3C standard providing a way to describe how to interpret a JSON document as Linked Data by means of a "context". JSON-LD provides a powerful solution for representing object references and namespaces in JSON based on standard web URIs, but is not itself a schema language. Without a schema providing a well defined structure, it is difficult to process an arbitrary JSON-LD document as idiomatic JSON because there are many ways to express the same data that are logically equivalent but structurally distinct.

Several schema languages exist for describing and validating JSON data, such as the Apache Avro data serialization system, however none understand linked data. As a result, to fully take advantage of JSON-LD to build the next generation of linked data applications, one must maintain separate JSON schema, JSON-LD context, RDF schema, and human documentation, despite significant overlap of content and obvious need for these documents to stay synchronized.

Schema Salad is designed to address this gap. It provides a schema language and processing rules for describing structured JSON content permitting URI resolution and strict document validation. The schema language supports linked data through annotations that describe the linked data interpretation of the content, enables generation of JSON-LD context and RDF schema, and production of RDF triples by applying the JSON-LD context. The schema language also provides for robust support of inline documentation.

1.1 Introduction to draft 1

This is the first version of Schema Salad. It is developed concurrently with draft 3 of the Common Workflow Language for use in specifying the Common Workflow Language, however Schema Salad is intended to be useful to a broader audience.

1.2 References to Other Specifications

Javascript Object Notation (JSON): http://json.org (http://json.org)

JSON Linked Data (JSON-LD): http://json-ld.org (http://json-ld.org)

YAML: http://yaml.org (http://yaml.org)

Avro: https://avro.apache.org/docs/current/spec.html (https://avro.apache.org/docs/current/spec.html)

Uniform Resource Identifier (URI) Generic Syntax: https://tools.ietf.org/html/rfc3986 (https://tools.ietf.org/html/rfc3986))

Resource Description Framework (RDF): http://www.w3.org/RDF/ (http://www.w3.org/RDF/)

UTF-8: https://www.ietf.org/rfc/rfc2279.txt (https://www.ietf.org/rfc/rfc2279.txt))

1.3 Scope

This document describes the syntax, data model, algorithms, and schema language for working with Salad documents. It is not intended to document a specific implementation of Salad, however it may serve as a reference for the behavior of conforming implementations.

1.4 Terminology

The terminology used to describe Salad documents is defined in the Concepts section of the specification. The terms defined in the following list are used in building those definitions and in describing the actions of an Salad implementation:

may: Conforming Salad documents and Salad implementations are permitted but not required to be interpreted as described.

must: Conforming Salad documents and Salad implementations are required to be interpreted as described; otherwise they are in error.

error: A violation of the rules of this specification; results are undefined. Conforming implementations may detect and report an error and may recover from it.

fatal error: A violation of the rules of this specification; results are undefined. Conforming implementations must not continue to process the document and may report an error.

at user option: Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.

2. Document model

2.1 Data concepts

An **object** is a data structure equivalent to the "object" type in JSON, consisting of a unordered set of name/value pairs (referred to here as **fields**) and where the name is a string and the value is a string, number, boolean, array, or object.

A **document** is a file containing a serialized object, or an array of objects.

A **document type** is a class of files that share a common structure and semantics.

A **document schema** is a formal description of the grammar of a document type.

A **base URI** is a context-dependent URI used to resolve relative references.

An **identifier** is a URI that designates a single document or single object within a document.

A **vocabulary** is the set of symbolic field names and enumerated symbols defined by a document schema, where each term maps to absolute URI.

2.2 Syntax

Conforming Salad documents are serialized and loaded using YAML syntax and UTF-8 text encoding. Salad documents are written using the JSON-compatible subset of YAML. Features of YAML such as headers and type tags that are not found in the standard JSON data model must not be used in conforming Salad documents. It is a fatal error if the document is not valid YAML.

A Salad document must consist only of either a single root object or an array of objects.

2.3 Document context

2.3.1 Implied context

The implicit context consists of the vocabulary defined by the schema and the base URI. By default, the base URI must be the URI that was used to load the document. It may be overridden by an explicit context.

2.3.2 Explicit context

If a document consists of a root object, this object may contain the fields `$base`, `$namespaces`, `$schemas`, and `$graph`:

- `$base`: Must be a string. Set the base URI for the document used to resolve relative references.
- `$namespaces`: Must be an object with strings as values. The keys of the object are namespace prefixes used in the document; the values of the object are the prefix expansions.
- `$schemas`: Must be an array of strings. This field may list URI references to documents in RDF-XML format which will be queried for RDF schema data. The subjects and predicates described by the RDF schema may provide additional semantic context for the document, and may be used for validation of prefixed extension fields found in the document.

Other directives beginning with `$` must be ignored.

2.4 Document graph

If a document consists of a single root object, this object may contain the field `$graph`. This field must be an array of objects. If present, this field holds the primary content of the document. A document that consists of array of objects at the root is an implicit graph.

2.5 Document metadata

If a document consists of a single root object, metadata about the document, such as authorship, may be declared in the root object.

2.6 Document schema

Document preprocessing, link validation and schema validation require a document schema. A schema may consist of:

- At least one record definition object which defines valid fields that make up a record type. Record field definitions include the valid types that may be assigned to each field and annotations to indicate fields that represent identifiers and links, described below in "Semantic Annotations".
- Any number of enumerated type objects which define a set of finite set of symbols that are valid value of the type.
- Any number of documentation objects which allow in-line documentation of the schema.

The schema for defining a salad schema (the metaschema) is described in detail in "Schema validation".

2.6.1 Record field annotations

In a document schema, record field definitions may include the field `jsonldPredicate`, which may be either a string or object. Implementations must use the following document preprocessing of fields by the following rules:

- If the value of `jsonldPredicate` is `@id`, the field is an identifier field.
- If the value of `jsonldPredicate` is an object, and contains that object contains the field `_type` with the value `@id`, the field is a link field.
- If the value of `jsonldPredicate` is an object, and contains that object contains the field `_type` with the value `@vocab`, the field is a vocabulary field, which is a subtype of link field.

2.7 Document traversal

To perform document document preprocessing, link validation and schema validation, the document must be traversed starting from the fields or array items of the root object or array and recursively visiting each child item which contains an object or arrays.

3. Document preprocessing

After processing the explicit context (if any), document preprocessing begins. Starting from the document root, object fields values or array items which contain objects or arrays are recursively traversed depth-first. For each visited object, field names, identifier fields, link fields, vocabulary fields, and `$import` and `$include` directives must be processed as described in this section. The order of traversal of child nodes within a parent node is undefined.

3.1 Field name resolution

The document schema declares the vocabulary of known field names. During preprocessing traversal, field name in the document which are not part of the schema vocabulary must be resolved to absolute URIs. Under "strict" validation, it is an error for a document to include fields which are not part of the vocabulary and not resolvable to absolute URIs. Fields names which are not part of the vocabulary are resolved using the following rules:

- If an field name URI begins with a namespace prefix declared in the document context (`@context`) followed by a colon `:`, the prefix and colon must be replaced by the namespace declared in `@context`.
- If there is a vocabulary term which maps to the URI of a resolved field, the field name must be replace with the vocabulary term.
- If a field name URI is an absolute URI consisting of a scheme and path and is not part of the vocabulary, no processing occurs.

Field name resolution is not relative. It must not be affected by the base URI.

3.1.1 Field name resolution example

Given the following schema:

```
{
  "$namespaces": {
    "acid": "http://example.com/acid#"
  },
  "$graph": [{
    "name": "ExampleType",
    "type": "record",
    "fields": [{
      "name": "base",
      "type": "string",
      "jsonldPredicate": "http://example.com/base"
    }]
  }]
}
```

Process the following example:

```
    {
      "base": "one",
      "form": {
        "http://example.com/base": "two",
        "http://example.com/three": "three",
      },
      "acid:four": "four"
    }
```

This becomes:

```
    {
      "base": "one",
      "form": {
        "base": "two",
        "http://example.com/three": "three",
      },
      "http://example.com/acid#four": "four"
    }
```

3.2 Identifier resolution

The schema may designate one or more fields as identifier fields to identify specific objects. Processing must resolve relative identifiers to absolute identifiers using the following rules:

- If an identifier URI is prefixed with `#` it is a URI relative fragment identifier. It is resolved relative to the base URI by setting or replacing the fragment portion of the base URI.
- If an identifier URI does not contain a scheme and is not prefixed `#` it is a parent relative fragment identifier. It is resolved relative to the base URI by the following rule: if the base URI does not contain a document fragment, set the fragment portion of the base URI. If the base URI does contain a document fragment, append a slash `/` followed by the identifier field to the fragment portion of the base URI.
- If an identifier URI begins with a namespace prefix declared in `$namespaces` followed by a colon `:`, the prefix and colon must be replaced by the namespace declared in `$namespaces`.
- If an identifier URI is an absolute URI consisting of a scheme and path, no processing occurs.

When preprocessing visits a node containing an identifier, that identifier must be used as the base URI to process child nodes.

It is an error for more than one object in a document to have the same absolute URI.

3.2.1 Identifier resolution example

Given the following schema:

```
{
  "$namespaces": {
    "acid": "http://example.com/acid#"
  },
  "$graph": [{
    "name": "ExampleType",
    "type": "record",
    "fields": [{
      "name": "id",
      "type": "string",
      "jsonldPredicate": "@id"
    }]
  }]
}
```

Process the following example:

```
  {
    "id": "http://example.com/base",
    "form": {
      "id": "one",
      "things": [
        {
          "id": "two"
        },
        {
          "id": "#three",
        },
        {
          "id": "four#five",
        },
        {
          "id": "acid:six",
        }
      ]
    }
  }
```

This becomes:

```
{
  "id": "http://example.com/base",
  "form": {
    "id": "http://example.com/base#one",
    "things": [
      {
        "id": "http://example.com/base#one/two"
      },
      {
        "id": "http://example.com/base#three"
      },
      {
        "id": "http://example.com/four#five",
      },
      {
        "id": "http://example.com/acid#six",
      }
    ]
  }
}
```

3.3 Link resolution

The schema may designate one or more fields as link fields reference other objects. Processing must resolve links to either absolute URIs using the following rules:

- If a reference URI is prefixed with `#` it is a relative fragment identifier. It is resolved relative to the base URI by setting or replacing the fragment portion of the base URI.
- If a reference URI does not contain a scheme and is not prefixed with `#` it is a path relative reference. If the reference URI contains `#` in any position other than the first character, the reference URI must be divided into a path portion and a fragment portion split on the first instance of `#`. The path portion is resolved relative to the base URI by the following rule: if the path portion of the base URI ends in a slash `/`, append the path portion of the reference URI to the path portion of the base URI. If the path portion of the base URI does not end in a slash, replace the final path segment with the path portion of the reference URI. Replace the fragment portion of the base URI with the fragment portion of the reference URI.
- If a reference URI begins with a namespace prefix declared in `$namespaces` followed by a colon `:`, the prefix and colon must be replaced by the namespace declared in `$namespaces`.
- If a reference URI is an absolute URI consisting of a scheme and path, no processing occurs.

Link resolution must not affect the base URI used to resolve identifiers and other links.

3.3.1 Link resolution example

Given the following schema:

```
{
  "$namespaces": {
    "acid": "http://example.com/acid#"
  },
  "$graph": [{
    "name": "ExampleType",
    "type": "record",
    "fields": [{
      "name": "link",
      "type": "string",
      "jsonldPredicate": {
        "_type": "@id"
      }
    }]
  }]
}
```

Process the following example:

```
{
  "$base": "http://example.com/base",
  "link": "http://example.com/base/zero",
  "form": {
    "link": "one",
    "things": [
      {
        "link": "two"
      },
      {
        "link": "#three",
      },
      {
        "link": "four#five",
      },
      {
        "link": "acid:six",
      }
    ]
  }
}
```

This becomes:

```
{
  "$base": "http://example.com/base",
  "link": "http://example.com/base/zero",
  "form": {
    "link": "http://example.com/one",
    "things": [
      {
        "link": "http://example.com/two"
      },
      {
        "link": "http://example.com/base#three"
      },
      {
        "link": "http://example.com/four#five",
      },
      {
        "link": "http://example.com/acid#six",
      }
    ]
  }
}
```

3.4 Vocabulary resolution

The schema may designate one or more vocabulary fields which use terms defined in the vocabulary. Processing must resolve vocabulary fields to either vocabulary terms or absolute URIs by first applying the link resolution rules defined above, then applying the following additional rule:

```
* If a reference URI is a vocabulary field, and there is a vocabulary
term which maps to the resolved URI, the reference must be replace with
the vocabulary term.
```

3.4.1 Vocabulary resolution example

Given the following schema:

```
{
  "$namespaces": {
    "acid": "http://example.com/acid#"
  },
  "$graph": [{
    "name": "Colors",
    "type": "enum",
    "symbols": ["acid:red"]
  },
  {
    "name": "ExampleType",
    "type": "record",
    "fields": [{
      "name": "voc",
      "type": "string",
      "jsonldPredicate": {
        "_type": "@vocab"
      }
    }]
  }]
}
```

Process the following example:

```
  {
    "form": {
      "things": [
        {
          "voc": "red",
        },
        {
          "voc": "http://example.com/acid#red",
        },
        {
          "voc": "http://example.com/acid#blue",
        }
      ]
    }
  }
```

This becomes:

```
  {
    "form": {
      "things": [
        {
          "voc": "red",
        },
        {
          "voc": "red",
        },
        {
          "voc": "http://example.com/acid#blue",
        }
      ]
    }
  }
```

3.5 Import

During preprocessing traversal, an implementation must resolve `$import` directives. An `$import` directive is an object consisting of exactly one field `$import` specifying resource by URI string. It is an error if there are additional fields in the `$import` object, such additional fields must be ignored.

The URI string must be resolved to an absolute URI using the link resolution rules described previously. Implementations must support loading from `file`, `http` and `https` resources. The URI referenced by `$import` must be loaded and recursively preprocessed as a Salad document. The external imported document does not inherit the context of the importing document, and the default base URI for processing the imported document must be the URI used to retrieve the imported document. If the `$import` URI includes a document fragment, the fragment must be excluded from the base URI used to preprocess the imported document.

Once loaded and processed, the `$import` node is replaced in the document structure by the object or array yielded from the import operation.

URIs may reference document fragments which refer to specific an object in the target document. This indicates that the `$import` node must be replaced by only the object with the appropriate fragment identifier.

It is a fatal error if an import directive refers to an external resource or resource fragment which does not exist or is not accessible.

3.5.1 Import example

import.yml:

```
{
  "hello": "world"
}
```

parent.yml:

```
{
  "form": {
    "bar": {
      "$import": "import.yml"
      }
  }
}
```

This becomes:

```
{
  "form": {
    "bar": {
      "hello": "world"
    }
  }
}
```

3.6 Include

During preprocessing traversal, an implementation must resolve `$include` directives. An `$include` directive is an object consisting of exactly one field `$include` specifying a URI string. It is an error if there are additional fields in the `$include` object, such additional fields must be ignored.

The URI string must be resolved to an absolute URI using the link resolution rules described previously. The URI referenced by `$include` must be loaded as a text data. Implementations must support loading from `file`, `http` and `https` resources. Implementations may transcode the character encoding of the text data to match that of the parent document, but must not interpret or parse the text document in any other way.

Once loaded, the `$include` node is replaced in the document structure by a string containing the text data loaded from the resource.

It is a fatal error if an import directive refers to an external resource which does not exist or is not accessible.

3.6.1 Include example

parent.yml:

```
{
  "form": {
    "bar": {
      "$include": "include.txt"
      }
  }
}
```

include.txt:

```
hello world
```

This becomes:

```
{
  "form": {
    "bar": "hello world"
  }
}
```

3.7 Mixin

During preprocessing traversal, an implementation must resolve `$mixin` directives. An `$mixin` directive is an object consisting of the field `$mixin` specifying resource by URI string. If there are additional fields in the `$mixin` object, these fields override fields in the object which is loaded from the `$mixin` URI.

The URI string must be resolved to an absolute URI using the link resolution rules described previously. Implementations must support loading from `file`, `http` and `https` resources. The URI referenced by `$mixin` must be loaded and recursively preprocessed as a Salad document. The external imported document must inherit the context of the importing document, however the file URI for processing the imported document must be the URI used to retrieve the imported document. The `$mixin` URI must not include a document fragment.

Once loaded and processed, the `$mixin` node is replaced in the document structure by the object or array yielded from the import operation.

URIs may reference document fragments which refer to specific an object in the target document. This indicates that the `$mixin` node must be replaced by only the object with the appropriate fragment identifier.

It is a fatal error if an import directive refers to an external resource or resource fragment which does not exist or is not accessible.

3.7.1 Mixin example

mixin.yml:

```
{
  "hello": "world",
  "carrot": "orange"
}
```

parent.yml:

```
{
  "form": {
    "bar": {
      "$mixin": "mixin.yml"
      "carrot": "cake"
      }
  }
}
```

This becomes:

```
{
  "form": {
    "bar": {
      "hello": "world",
      "carrot": "cake"
    }
  }
}
```

4. SaladRecordSchema

Fields

field	type	required	description
`name`	string	True	The identifier for this type
`type`	Record_symbol	True	Must be `record`

`fields`	array<SaladRecordField>	False	Defines the fields of the record.
`doc`	string | array<string>	False	A documentation string for this type, or an array of strings which should be concatenated.
`docParent`	string	False	Hint to indicate that during documentation generation, documentation for this type should appear in a subsection under `docParent` .
`docChild`	string | array<string>	False	Hint to indicate that during documentation generation, documentation for `docChild` should appear in a subsection under this type.
`docAfter`	string	False	Hint to indicate that during documentation generation, documentation for this type should appear after the `docAfter` section at the same level.
`jsonldPredicate`	string | JsonldPredicate	False	Annotate this type with linked data context.
`documentRoot`	boolean	False	If true, indicates that the type is a valid at the document root. At least one type in a schema must be tagged with `documentRoot: true` .
`abstract`	boolean	False	If true, this record is abstract and may be used as a base for other records, but is not valid on its own.
`extends`	string | array<string>	False	Indicates that this record inherits fields from one or more base records.
`specialize`	array<SpecializeDef>	False	Only applies if `extends` is declared. Apply type specialization using the base record as a template. For each field inherited from the base record, replace any instance of the type `specializeFrom` with `specializeTo` .

4.1 SaladRecordField

A field of a record.

Fields

field	type	required	description
`name`	string	True	The name of the field
`type`	PrimitiveType | RecordSchema | EnumSchema | ArraySchema | string | array<PrimitiveType | RecordSchema | EnumSchema | ArraySchema | string>	True	The field type
`doc`	string	False	A documentation string for this field
`jsonldPredicate`	string | JsonldPredicate	False	Annotate this type with linked data context.

4.1.1 PrimitiveType

Salad data types are based on Avro schema declarations. Refer to the Avro schema declaration documentation (https://avro.apache.org/docs/current/spec.html#schemas) for detailed information.

Symbols

symbol	description
`null`	no value
`boolean`	a binary value
`int`	32-bit signed integer
`long`	64-bit signed integer
`float`	single precision (32-bit) IEEE 754 floating-point number
`double`	double precision (64-bit) IEEE 754 floating-point number
`string`	Unicode character sequence

4.1.2 RecordSchema

Fields

field	type	required	description

`type`	Record_symbol	True	Must be `record`
`fields`	array<RecordField>	False	Defines the fields of the record.

4.1.3 RecordField

A field of a record.

Fields

field	type	required	description
`name`	string	True	The name of the field
`type`	PrimitiveType | RecordSchema | EnumSchema | ArraySchema | string | array<PrimitiveType | RecordSchema | EnumSchema | ArraySchema | string>	True	The field type
`doc`	string	False	A documentation string for this field

4.1.3.1 EnumSchema

Define an enumerated type.

Fields

field	type	required	description
`symbols`	array<string>	True	Defines the set of valid symbols.
`type`	Enum_symbol	True	Must be `enum`

4.1.3.2 ArraySchema

Fields

field	type	required	description
`items`	PrimitiveType | RecordSchema | EnumSchema | ArraySchema | string | array<PrimitiveType | RecordSchema | EnumSchema | ArraySchema | string>	True	Defines the type of the array elements.
`type`	Array_symbol	True	Must be `array`

4.1.4 JsonldPredicate

Attached to a record field to define how the parent record field is handled for URI resolution and JSON-LD context generation.

Fields

field	type	required	description
`_id`	string	False	The predicate URI that this field corresponds to. Corresponds to JSON-LD `@id` directive.
`_type`	string	False	The context type hint, corresponds to JSON-LD `@type` directive. • If the value of this field is `@id` and `identity` is false or unspecified, the parent field must be resolved using the link resolution rules. If `identity` is true, the parent field must be resolved using the identifier expansion rules. • If the value of this field is `@vocab` , the parent field must be resolved using the vocabulary resolution rules.
`_container`	string	False	Structure hint, corresponds to JSON-LD `@container` directive.
`identity`	boolean	False	If true and `_type` is `@id` this indicates that the parent field must be resolved according to identity resolution rules instead of link resolution rules. In addition, the field value is considered an assertion that the linked value exists; absence of an object in the loaded document with the URI is not an error.
`noLinkCheck`	boolean	False	If true, this indicates that link validation traversal must stop at this field. This field (it is is a URI) or any fields under it (if it is an object or array) are not subject to link checking.
`mapSubject`	string	False	If the value of the field is a JSON object, it must be transformed into an array of JSON objects, where each key-value pair from the source JSON object is a list item, the list items must be JSON objects, and the key is assigned to the field specified by `mapSubject` .

`mapPredicate`	string	False	Only applies if `mapSubject` is also provided. If the value of the field is a JSON object, it is transformed as described in `mapSubject`, with the addition that when the value of a map item is not an object, the item is transformed to a JSON object with the key assigned to the field specified by `mapSubject` and the value assigned to the field specified by `mapPredicate`.
`refScope`	int	False	If the field contains a relative reference, it must be resolved by searching for valid document references in each successive parent scope in the document fragment. For example, a reference of `foo` in the context `#foo/bar/baz` will first check for the existence of `#foo/bar/baz/foo`, followed by `#foo/bar/foo`, then `#foo/foo` and then finally `#foo`. The first valid URI in the search order shall be used as the fully resolved value of the identifier. The value of the refScope field is the specified number of levels from the containing identifer scope before starting the search, so if `refScope: 2` then "baz" and "bar" must be stripped to get the base `#foo` and search `#foo/foo` and the `#foo`. The last scope searched must be the top level scope before determining if the identifier cannot be resolved.
`typeDSL`	boolean	False	Field must be expanded based on the the Schema Salad type DSL.

4.2 SpecializeDef

Fields

field	type	required	description
`specializeFrom`	string	True	The data type to be replaced
`specializeTo`	string	True	The new data type to replace with

5. SaladEnumSchema

Define an enumerated type.

Fields

field	type	required	description
`symbols`	array<string>	True	Defines the set of valid symbols.
`type`	Enum_symbol	True	Must be `enum`
`doc`	string | array<string>	False	A documentation string for this type, or an array of strings which should be concatenated.
`docParent`	string	False	Hint to indicate that during documentation generation, documentation for this type should appear in a subsection under `docParent`.
`docChild`	string | array<string>	False	Hint to indicate that during documentation generation, documentation for `docChild` should appear in a subsection under this type.
`docAfter`	string	False	Hint to indicate that during documentation generation, documentation for this type should appear after the `docAfter` section at the same level.
`jsonldPredicate`	string | JsonldPredicate	False	Annotate this type with linked data context.
`documentRoot`	boolean	False	If true, indicates that the type is a valid at the document root. At least one type in a schema must be tagged with `documentRoot: true`.
`extends`	string | array<string>	False	Indicates that this enum inherits symbols from a base enum.

6. Documentation

A documentation section. This type exists to facilitate self-documenting schemas but has no role in formal validation.

Fields

field	type	required	description
`name`	string	True	The identifier for this type
`type`	Documentation_symbol	True	Must be `documentation`
`doc`	string | array<string>	False	A documentation string for this type, or an array of strings which should be concatenated.

`docParent`	string	False	Hint to indicate that during documentation generation, documentation for this type should appear in a subsection under `docParent` .
`docChild`	string | array<string>	False	Hint to indicate that during documentation generation, documentation for `docChild` should appear in a subsection under this type.
`docAfter`	string	False	Hint to indicate that during documentation generation, documentation for this type should appear after the `docAfter` section at the same level.