W3C Recommendation

JSON-LD 1.0 Processing Algorithms and API

W3C Recommendation 16 January 2014

This version:
http://www.w3.org/TR/2014/REC-json-ld-api-20140116/
Latest published version:
http://www.w3.org/TR/json-ld-api/
Test suite:
http://www.w3.org/2013/json-ld-tests/
Previous version:
http://www.w3.org/TR/2013/PR-json-ld-api-20131105/
Editors:
Markus Lanthaler, Graz University of Technology
Gregg Kellogg, Kellogg Associates
Manu Sporny, Digital Bazaar
Authors:
Dave Longley, Digital Bazaar
Gregg Kellogg, Kellogg Associates
Markus Lanthaler, Graz University of Technology
Manu Sporny, Digital Bazaar

Please refer to the **errata** for this document, which may include some normative corrections.

This document is also available in this non-normative format: diff to previous version

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

This specification defines a set of algorithms for programmatic transformations of JSON-LD documents. Restructuring data according to the defined transformations often dramatically simplifies its usage. Furthermore, this document proposes an Application Programming Interface (API) for developers implementing the specified algorithms.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This specification has been developed by the JSON for Linking Data Community Group before it has been transferred to the RDF Working Group for review, improvement, and publication along the Recommendation track. The document contains small editorial changes arising from comments received during the Proposed Recommendation review; see the diff-marked version for details.

There are several independent interoperable implementations of this specification. An implementation report as of October 2013 is available.

This document was published by the RDF Working Group as a Recommendation. If you wish to make comments regarding this document, please send them to public-rdf-comments@w3.org (subscribe, archives). All comments are welcome.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

1. Introduction
2. Features
    2.1 Expansion
    2.2 Compaction
    2.3 Flattening
    2.4 RDF Serialization/Deserialization
3. Conformance
4. General Terminology
5. Algorithm Terms
6. Context Processing Algorithms
    6.1 Context Processing Algorithm
    6.2 Create Term Definition
    6.3 IRI Expansion
7. Expansion Algorithms
    7.1 Expansion Algorithm
    7.2 Value Expansion
8. Compaction Algorithms
    8.1 Compaction Algorithm
    8.2 Inverse Context Creation
    8.3 IRI Compaction
    8.4 Term Selection
    8.5 Value Compaction
9. Flattening Algorithms
    9.1 Flattening Algorithm
    9.2 Node Map Generation
    9.3 Generate Blank Node Identifier
10. RDF Serialization/Deserialization Algorithms
    10.1 Deserialize JSON-LD to RDF algorithm
    10.2 Object to RDF Conversion
    10.3 List to RDF Conversion
    10.4 Serialize RDF as JSON-LD Algorithm
    10.5 RDF to Object Conversion
    10.6 Data Round Tripping
11. The Application Programming Interface
    11.1 The `JsonLdProcessor` Interface
    11.2 The `JsonLdOptions` Type
    11.3 Remote Document and Context Retrieval
    11.4 Error Handling
A. Acknowledgements
B. References
    B.1 Normative references
    B.2 Informative references

1. Introduction

This section is non-normative.

This document is a detailed specification of the JSON-LD processing algorithms. The document is primarily intended for the following audiences:

- Software developers who want to implement the algorithms to transform JSON-LD documents.
- Web authors and developers who want a very detailed view of how a JSON-LD Processor operates.
- Developers who want an overview of the proposed JSON-LD API.

To understand the basics in this specification you must first be familiar with JSON, which is detailed in [RFC4627]. You must also understand the JSON-LD syntax defined in [JSON-LD], which is the base syntax used by all of the algorithms in this document. To understand the API and how it is intended to operate in a programming environment, it is useful to have working knowledge of the JavaScript programming language [ECMA-262] and WebIDL [WEBIDL]. To understand how JSON-LD maps to RDF, it is helpful to be familiar with the basic RDF concepts [RDF11-CONCEPTS].

2. Features

This section is non-normative.

The JSON-LD Syntax specification [JSON-LD] defines a syntax to express Linked Data in JSON. Because there is more than one way to express Linked Data using this syntax, it is often useful to be able to transform JSON-LD documents so that they may be more easily consumed by specific applications.

JSON-LD uses contexts to allow Linked Data to be expressed in a way that is specifically tailored to a particular person or application. By providing a context, JSON data can be expressed in a way that is a natural fit for a particular person or application whilst also indicating how the data should be understood at a global scale. In order for people or applications to share data that was created using a context that is different from their own, a JSON-LD processor must be able to transform a document from one context to another. Instead of requiring JSON-LD processors to write specific code for every imaginable context switching scenario, it is much easier to specify a single algorithm that can remove any context. Similarly, another algorithm can be specified to subsequently apply any context. These two algorithms represent the most basic transformations of JSON-LD documents. They are referred to as expansion and compaction, respectively.

There are four major types of transformation that are discussed in this document: expansion, compaction, flattening, and RDF serialization/deserialization.

2.1 Expansion

This section is non-normative.

The algorithm that removes context is called ***expansion***. Before performing any other transformations on a JSON-LD document, it is easiest to remove any context from it and to make data structures more regular.

To get an idea of how context and data structuring affects the same data, here is an example of JSON-LD that uses only terms and is fairly compact:

EXAMPLE 1: Sample JSON-LD document

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  },
  "@id": "http://me.markus-lanthaler.com/",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/"
}
```

The next input example uses one IRI to express a property and an array to encapsulate another, but leaves the rest of the information untouched.

EXAMPLE 2: Sample JSON-LD document using an IRI instead of a term to express a property

```
{
  "@context": {
    "website": "http://xmlns.com/foaf/0.1/homepage"
  },
  "@id": "http://me.markus-lanthaler.com/",
  "http://xmlns.com/foaf/0.1/name": "Markus Lanthaler",
  "website": { "@id": "http://www.markus-lanthaler.com/" }
}
```

Note that both inputs are valid JSON-LD and both represent the same information. The difference is in their context information and in the data structures used. A JSON-LD processor can remove context and ensure that the data is more regular by employing expansion.

Expansion has two important goals: removing any contextual information from the document, and ensuring all values are represented in a regular form. These goals are accomplished by expanding all properties to absolute IRIs and by expressing all values in arrays in expanded form. Expanded form is the most verbose and regular way of expressing of values in JSON-LD; all contextual information from the document is instead stored locally with each value. Running the Expansion algorithm (`expand` operation) against the above examples results in the following output:

EXAMPLE 3: Expanded sample document

```
[
  {
    "@id": "http://me.markus-lanthaler.com/",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Markus Lanthaler" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://www.markus-lanthaler.com/" }
    ]
  }
]
```

Note that in the output above all context definitions have been removed, all terms and compact IRIs have been expanded to absolute IRIs, and all JSON-LD values are expressed in arrays in expanded form. While the output is more verbose and difficult for a human to read, it establishes a baseline that makes JSON-LD processing easier because of its very regular structure.

2.2 Compaction

This section is non-normative.

While expansion removes context from a given input, compaction's primary function is to perform the opposite operation: to express a given input according to a particular context. ***Compaction*** applies a context that specifically tailors the way information is expressed for a particular person or application. This simplifies applications that consume JSON or JSON-LD by expressing the data in application-specific terms, and it makes the data easier to read by humans.

Compaction uses a developer-supplied context to shorten IRIs to terms or compact IRIs and JSON-LD values expressed in expanded form to simple values such as strings or numbers.

For example, assume the following expanded JSON-LD input document:

EXAMPLE 4: Expanded sample document

```
[
  {
    "@id": "http://me.markus-lanthaler.com/",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Markus Lanthaler" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://www.markus-lanthaler.com/" }
    ]
  }
]
```

Additionally, assume the following developer-supplied JSON-LD context:

EXAMPLE 5: JSON-LD context

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  }
}
```

Running the Compaction Algorithm (`compact` operation) given the context supplied above against the JSON-LD input document provided above would result in the following output:

EXAMPLE 6: Compacted sample document

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  },
  "@id": "http://me.markus-lanthaler.com/",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/"
}
```

Note that all IRIs have been compacted to terms as specified in the context, which has been injected into the output. While compacted output is useful to humans, it is also used to generate structures that are easy to program against. Compaction enables developers to map any expanded document into an application-specific compacted document. While the context provided above mapped `http://xmlns.com/foaf/0.1/name` to `name`, it could also have been mapped to any other term provided by the developer.

2.3 Flattening

This section is non-normative.

While expansion ensures that a document is in a uniform structure, flattening goes a step further to ensure that the shape of the data is deterministic. In expanded documents, the properties of a single node may be spread across a number of different JSON objects. By flattening a document, all properties of a node are collected in a single JSON object and all blank nodes are labeled with a blank node identifier. This may drastically simplify the code required to process JSON-LD data in certain applications.

For example, assume the following JSON-LD input document:

EXAMPLE 7: Sample JSON-LD document

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@id": "http://me.markus-lanthaler.com/",
  "name": "Markus Lanthaler",
  "knows": [
    {
      "name": "Dave Longley"
    }
  ]
}
```

Running the Flattening algorithm (`flatten` operation) with a context set to null to prevent compaction returns the following document:

EXAMPLE 8: Flattened sample document in expanded form

```
[
  {
    "@id": "_:t0",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Dave Longley" }
    ]
  },
  {
    "@id": "http://me.markus-lanthaler.com/",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Markus Lanthaler" }
    ],
    "http://xmlns.com/foaf/0.1/knows": [
      { "@id": "_:t0" }
    ]
  }
]
```

Note how in the output above all properties of a node are collected in a single JSON object and how the blank node representing "Dave Longley" has been assigned the blank node identifier `_:t0`.

To make it easier for humans to read or for certain applications to process it, a flattened document can be compacted by passing a context. Using the same context as the input document, the flattened and compacted document looks as follows:

EXAMPLE 9: Flattened and compacted sample document

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@graph": [
    {
      "@id": "_:t0",
      "name": "Dave Longley"
    },
    {
      "@id": "http://me.markus-lanthaler.com/",
      "name": "Markus Lanthaler",
      "knows": { "@id": "_:t0" }
    }
  ]
}
```

Please note that the result of flattening and compacting a document is always a JSON object which contains an `@graph` member that represents the default graph.

2.4 RDF Serialization/Deserialization

This section is non-normative.

JSON-LD can be used to serialize RDF data as described in [RDF11-CONCEPTS]. This ensures that data can be round-tripped to and from any RDF syntax without any loss in fidelity.

For example, assume the following RDF input serialized in Turtle [TURTLE]:

EXAMPLE 10: Sample Turtle document

```
<http://me.markus-lanthaler.com/> <http://xmlns.com/foaf/0.1/name> "Markus Lanthaler" .
<http://me.markus-lanthaler.com/> <http://xmlns.com/foaf/0.1/homepage> <http://www.markus-lanthaler.com/> .
```

Using the Serialize RDF as JSON-LD algorithm a developer could transform this document into expanded JSON-LD:

EXAMPLE 11: Sample Turtle document converted to JSON-LD

```
[
  {
    "@id": "http://me.markus-lanthaler.com/",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Markus Lanthaler" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://www.markus-lanthaler.com/" }
    ]
  }
]
```

Note that the output above could easily be compacted using the technique outlined in the previous section. It is also possible to deserialize the JSON-LD document back to RDF using the Deserialize JSON-LD to RDF algorithm.

3. Conformance

All examples and notes as well as sections marked as non-normative in this specification are non-normative. Everything else in this specification is normative.

The keywords MUST, MUST NOT, REQUIRED, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this specification are to be interpreted as described in [RFC2119].

There are two classes of products that can claim conformance to this specification: JSON-LD Processors, and RDF Serializers/Deserializers.

A conforming ***JSON-LD Processor*** is a system which can perform the Expansion, Compaction, and Flattening operations defined in this specification.

JSON-LD Processors MUST NOT attempt to correct malformed IRIs or language tags; however, they MAY issue validation warnings. IRIs are not modified other than conversion between relative and absolute IRIs.

A conforming ***RDF Serializer/Deserializer*** is a system that can deserialize JSON-LD to RDF and serialize RDF as JSON-LD as defined in this specification.

The algorithms in this specification are generally written with more concern for clarity than efficiency. Thus, JSON-LD Processors may implement the algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms.

NOTE

Implementers can partially check their level of conformance to this specification by successfully passing the test cases of the JSON-LD test suite [JSON-LD-TESTS]. Note, however, that passing all the tests in the test suite does not imply complete conformance to this specification. It only implies that the implementation conforms to aspects tested by the test suite.

4. General Terminology

This document uses the following terms as defined in JSON [RFC4627]. Refer to the *JSON Grammar* section in [RFC4627] for formal definitions.

JSON object
: An object structure is represented as a pair of curly brackets surrounding zero or more key-value pairs. A key is a string. A single colon comes after each key, separating the key from the value. A single comma separates a value from a following key. In contrast to JSON, in JSON-LD the keys in an object must be unique.

array
: An array structure is represented as square brackets surrounding zero or more values. Values are separated by commas. In JSON, an array is an *ordered* sequence of zero or more values. While JSON-LD uses the same array representation as JSON, the collection is *unordered* by default. While order is preserved in regular JSON arrays, it is not in regular JSON-LD arrays unless specifically defined (see "*Sets and Lists*" in the JSON-LD specification [JSON-LD]).

string
: A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes (if necessary). A character is represented as a single character string.

number
: A number is similar to that used in most programming languages, except that the octal and hexadecimal formats are not used and that leading zeros are not allowed.

true and ***false***

Values that are used to express one of two possible boolean states.

null
: The null value. A key-value pair in the `@context` where the value, or the `@id` of the value, is null explicitly decouples a term's association with an IRI. A key-value pair in the body of a JSON-LD document whose value is null has the same meaning as if the key-value pair was not defined. If `@value`, `@list`, or `@set` is set to null in expanded form, then the entire JSON object is ignored.

Furthermore, the following terminology is used throughout this document:

keyword
: A JSON key that is specific to JSON-LD, specified in the section *Syntax Tokens and Keywords* of the JSON-LD specification [JSON-LD].

context
: A set of rules for interpreting a JSON-LD document as specified in the section *The Context* of the JSON-LD specification [JSON-LD].

JSON-LD document
: A JSON-LD document is a serialization of a collection of graphs and comprises exactly one default graph and zero or more named graphs.

named graph
: A named graph is a pair consisting of an IRI or blank node (the ***graph name***) and a graph.

default graph
: The default graph is the only graph in a JSON-LD document which has no graph name.

Graph
: A labeled directed graph, i.e., a set of nodes connected by edges, as specified in the *Data Model* section of the JSON-LD specification [JSON-LD].

edge
: Every edge has a direction associated with it and is labeled with an IRI or a blank node identifier. Within the JSON-LD syntax these edge labels are called ***properties***. Whenever possible, an edge should be labeled with an IRI.

node
: Every node is an IRI, a blank node, a JSON-LD value, or a list.

IRI
: An IRI (Internationalized Resource Identifier) is a string that conforms to the syntax defined in [RFC3987].

absolute IRI
: An absolute IRI is defined in [RFC3987] containing a *scheme* along with a *path* and optional *query* and fragment segments.

relative IRI
: A relative IRI is an IRI that is relative to some other absolute IRI.

blank node
: A node in a graph that is neither an IRI, nor a JSON-LD value, nor a list.

blank node identifier
: A blank node identifier is a string that can be used as an identifier for a blank node within the scope of a JSON-LD document. Blank node identifiers begin with `_:`.

JSON-LD value
: A JSON-LD value is a string, a number, true or false, a typed value, or a language-tagged string.

typed value
: A typed value consists of a value, which is a string, and a type, which is an IRI.

language-tagged string
: A language-tagged string consists of a string and a non-empty language tag as defined by [BCP47]. The language tag must be well-formed according to section 2.2.9 Classes of Conformance of [BCP47], and is normalized to lowercase.

list
: A list is an ordered sequence of IRIs, blank nodes, and JSON-LD values.

5. Algorithm Terms

active graph
: The name of the currently active graph that the processor should use when processing.

active subject
: The currently active subject that the processor should use when processing.

active property
: The currently active property or keyword that the processor should use when processing.

active context
: A context that is used to resolve terms while the processing algorithm is running.

local context
: A context that is specified within a JSON object, specified via the `@context` keyword.

JSON-LD input
: The JSON-LD data structure that is provided as input to the algorithm.

term
: A term is a short word defined in a context that may be expanded to an IRI

compact IRI
: A compact IRI has the form of ***prefix***:*suffix* and is used as a way of expressing an IRI without needing to define separate term definitions for each IRI contained within a common vocabulary identified by prefix.

node object

A node object represents zero or more properties of a node in the graph serialized by the JSON-LD document. A JSON object is a node object if it exists outside of the JSON-LD context and:

- it does not contain the `@value`, `@list`, or `@set` keywords, or
- it is not the top-most JSON object in the JSON-LD document consisting of no other members than `@graph` and `@context`.

value object
: A value object is a JSON object that has an `@value` member.

list object
: A list object is a JSON object that has an `@list` member.

set object
: A set object is a JSON object that has an `@set` member.

scalar
: A scalar is either a JSON string, number, true, or false.

RDF subject
: A subject as specified by [RDF11-CONCEPTS].

RDF predicate
: A predicate as specified by [RDF11-CONCEPTS].

RDF object
: An object as specified by [RDF11-CONCEPTS].

RDF triple
: A triple as specified by [RDF11-CONCEPTS].

RDF dataset
: A dataset as specified by [RDF11-CONCEPTS] representing a collection of RDF graphs.

6. Context Processing Algorithms

6.1 Context Processing Algorithm

When processing a JSON-LD data structure, each processing rule is applied using information provided by the active context. This section describes how to produce an active context.

The active context contains the active ***term definitions*** which specify how properties and values have to be interpreted as well as the current ***base IRI***, the ***vocabulary mapping*** and the ***default language***. Each term definition consists of an ***IRI mapping***, a boolean flag ***reverse property***, an optional ***type mapping*** or ***language mapping***, and an optional ***container mapping***. A term definition can not only be used to map a term to an IRI, but also to map a term to a keyword, in which case it is referred to as a ***keyword alias***.

When processing, the active context is initialized without any term definitions, vocabulary mapping, or default language. If a local context is encountered during processing, a new active context is created by cloning the existing active context. Then the information from the local context is merged into the new active context. Given that local contexts may contain references to remote contexts, this includes their retrieval.

Overview

This section is non-normative.

First we prepare a new active context *result* by cloning the current active context. Then we normalize the form of the passed local context to an array. Local contexts may be in the form of a JSON object, a string, or an array containing a combination of the two. Finally we process each context contained in the local context array as follows.

If context is a string, it represents a reference to a remote context. We dereference the remote context and replace context with the value of the `@context` key of the top-level object in the retrieved JSON-LD document. If there's no such key, an invalid remote context has been detected. Otherwise, we process context by recursively using this algorithm ensuring that there is no cyclical reference.

If context is a JSON object, we first update the base IRI, the vocabulary mapping, and the default language by processing three specific keywords: `@base`, `@vocab`, and `@language`. These are handled before any other keys in the local context because they affect how the other keys are processed. Please note that `@base` is ignored when processing remote contexts.

Then, for every other key in local context, we update the term definition in *result*. Since term definitions in a local context may themselves contain terms or compact IRIs, we may need to recurse. When doing so, we must ensure that there is no cyclical dependency, which is an error. After we have processed any term definition dependencies, we update the current term definition, which may be a keyword alias.

Finally, we return *result* as the new active context.

Algorithm

This algorithm specifies how a new active context is updated with a local context. The algorithm takes three input variables: an active context, a local context, and an array *remote contexts* which is used to detect cyclical context inclusions. If *remote contexts* is not passed, it is initialized to an empty array.

1) Initialize *result* to the result of cloning active context.

2) If local context is not an array, set it to an array containing only local context.

3) For each item *context* in local context:

3.1) If *context* is null, set *result* to a newly-initialized active context and continue with the next *context*. The base IRI of the active context is set to the IRI of the currently being processed document (which might be different from the currently being processed context), if available; otherwise to null. If set, the `base` option of a JSON-LD API Implementation overrides the base IRI.

3.2) If *context* is a string,

3.2.1) Set *context* to the result of resolving *value* against the base IRI which is established as specified in *section 5.1 Establishing a Base URI* of [RFC3986]. Only the basic algorithm in *section 5.2* of [RFC3986] is used; neither *Syntax-Based Normalization* nor *Scheme-Based Normalization* are performed. Characters additionally allowed in IRI references are treated in the same way that unreserved characters are treated in URI references, per *section 6.5* of [RFC3987].

3.2.2) If *context* is in the *remote contexts* array, a `recursive context inclusion` error has been detected and processing is aborted; otherwise, add *context* to *remote contexts*.

3.2.3) Dereference *context*. If *context* cannot be dereferenced, a `loading remote context failed` error has been detected and processing is aborted. If the dereferenced document has no top-level JSON object with an `@context` member, an `invalid remote context` has been detected and processing is aborted; otherwise, set *context* to the value of that member.

3.2.4) Set *result* to the result of recursively calling this algorithm, passing *result* for active context, *context* for local context, and *remote contexts*.

3.2.5) Continue with the next *context*.

3.3) If *context* is not a JSON object, an `invalid local context` error has been detected and processing is aborted.

3.4) If *context* has an `@base` key and *remote contexts* is empty, i.e., the currently being processed context is not a remote context:

3.4.1) Initialize *value* to the value associated with the `@base` key.

3.4.2) If *value* is null, remove the base IRI of *result*.

3.4.3) Otherwise, if *value* is an absolute IRI, the base IRI of *result* is set to *value*.

3.4.4) Otherwise, if *value* is a relative IRI and the base IRI of *result* is not null, set the base IRI of *result* to the result of resolving *value* against the current base IRI of *result*.

3.4.5) Otherwise, an `invalid base IRI` error has been detected and processing is aborted.

3.5) If *context* has an `@vocab` key:

3.5.1) Initialize *value* to the value associated with the `@vocab` key.

3.5.2) If *value* is null, remove any vocabulary mapping from *result*.

3.5.3) Otherwise, if *value* is an absolute IRI or blank node identifier, the vocabulary mapping of *result* is set to *value*. If it is not an absolute IRI or blank node identifier, an `invalid vocab mapping` error has been detected and processing is aborted.

3.6) If *context* has an `@language` key:

3.6.1) Initialize *value* to the value associated with the `@language` key.

3.6.2) If *value* is null, remove any default language from *result*.

3.6.3) Otherwise, if *value* is string, the default language of *result* is set to lowercased *value*. If it is not a string, an `invalid default language` error has been detected and processing is aborted.

3.7) Create a JSON object *defined* to use to keep track of whether or not a term has already been defined or currently being defined during recursion.

3.8) For each *key-value* pair in *context* where *key* is not `@base`, `@vocab`, or `@language`, invoke the Create Term Definition algorithm, passing *result* for active context, *context* for local context, *key*, and *defined*.

4) Return *result*.

6.2 Create Term Definition

This algorithm is called from the Context Processing algorithm to create a term definition in the active context for a term being processed in a local context.

Overview

This section is non-normative.

Term definitions are created by parsing the information in the given local context for the given term. If the given term is a compact IRI, it may omit an IRI mapping by depending on its prefix having its own term definition. If the prefix is a key in the local context, then its term definition must first be created, through recursion, before continuing. Because a term definition can depend on other term definitions, a mechanism must be used to detect cyclical dependencies. The solution employed here uses a map, *defined*, that keeps track of whether or not a term has been defined or is currently in the process of being defined. This map is checked before any recursion is attempted.

After all dependencies for a term have been defined, the rest of the information in the local context for the given term is taken into account, creating the appropriate IRI mapping, container mapping, and type mapping or language mapping for the term.

Algorithm

The algorithm has four required inputs which are: an active context, a local context, a *term*, and a map *defined*.

1) If *defined* contains the key *term* and the associated value is true (indicating that the term definition has already been created), return. Otherwise, if the value is false, a `cyclic IRI mapping` error has been detected and processing is aborted.

2) Set the value associated with *defined*'s *term* key to false. This indicates that the term definition is now being created but is not yet complete.

3) Since keywords cannot be overridden, *term* must not be a keyword. Otherwise, a `keyword redefinition` error has been detected and processing is aborted.

4) Remove any existing term definition for *term* in active context.

5) Initialize *value* to a copy of the value associated with the key *term* in local context.

6) If *value* is null or *value* is a JSON object containing the key-value pair `@id`-null, set the term definition in active context to null, set the value associated with *defined*'s key *term* to true, and return.

7) Otherwise, if *value* is a string, convert it to a JSON object consisting of a single member whose key is `@id` and whose value is *value*.

8) Otherwise, *value* must be a JSON object, if not, an `invalid term definition` error has been detected and processing is aborted.

9) Create a new term definition, *definition*.

10) If *value* contains the key `@type`:

> **10.1)** Initialize *type* to the value associated with the `@type` key, which must be a string. Otherwise, an `invalid type mapping` error has been detected and processing is aborted.
>
> **10.2)** Set *type* to the result of using the IRI Expansion algorithm, passing active context, *type* for *value*, true for *vocab*, false for *document relative*, local context, and *defined*. If the expanded *type* is neither `@id`, nor `@vocab`, nor an absolute IRI, an `invalid type mapping` error has been detected and processing is aborted.
>
> **10.3)** Set the type mapping for *definition* to *type*.

11) If *value* contains the key `@reverse`:

> **11.1)** If *value* contains an `@id`, member, an `invalid reverse property` error has been detected and processing is aborted.
>
> **11.2)** If the value associated with the `@reverse` key is not a string, an `invalid IRI mapping` error has been detected and processing is aborted.
>
> **11.3)** Otherwise, set the IRI mapping of *definition* to the result of using the IRI Expansion algorithm, passing active context, the value associated with the `@reverse` key for *value*, true for *vocab*, false for *document relative*, local context, and *defined*. If the result is neither an absolute IRI nor a blank node identifier, i.e., it contains no colon (`:`), an `invalid IRI mapping` error has been detected and processing is aborted.
>
> **11.4)** If *value* contains an `@container` member, set the container mapping of *definition* to its value; if its value is neither `@set`, nor `@index`, nor null, an `invalid reverse property` error has been detected (reverse properties only support set- and index-containers) and processing is aborted.
>
> **11.5)** Set the reverse property flag of *definition* to true.
>
> **11.6)** Set the term definition of *term* in active context to *definition* and the value associated with *defined*'s key *term* to true and return.

12) Set the reverse property flag of *definition* to false.

13) If *value* contains the key `@id` and its value does not equal *term*:

> **13.1)** If the value associated with the `@id` key is not a string, an `invalid IRI mapping` error has been detected and processing is aborted.
>
> **13.2)** Otherwise, set the IRI mapping of *definition* to the result of using the IRI Expansion algorithm, passing active context, the value associated with the `@id` key for *value*, true for *vocab*, false for *document relative*, local context, and *defined*. If the resulting IRI mapping is neither a keyword, nor an absolute IRI, nor a blank node identifier, an `invalid IRI mapping` error has been detected and processing is aborted; if it equals `@context`, an `invalid keyword alias` error has been detected and processing is aborted.

14) Otherwise if the *term* contains a colon (`:`):

> **14.1)** If *term* is a compact IRI with a prefix that is a key in local context a dependency has been found. Use this

algorithm recursively passing active context, local context, the prefix as *term*, and *defined*.

14.2) If *term*'s prefix has a term definition in active context, set the IRI mapping of *definition* to the result of concatenating the value associated with the prefix's IRI mapping and the *term*'s *suffix*.

14.3) Otherwise, *term* is an absolute IRI or blank node identifier. Set the IRI mapping of *definition* to *term*.

15) Otherwise, if active context has a vocabulary mapping, the IRI mapping of *definition* is set to the result of concatenating the value associated with the vocabulary mapping and *term*. If it does not have a vocabulary mapping, an `invalid IRI mapping` error been detected and processing is aborted.

16) If *value* contains the key `@container`:

16.1) Initialize *container* to the value associated with the `@container` key, which must be either `@list`, `@set`, `@index`, or `@language`. Otherwise, an `invalid container mapping` error has been detected and processing is aborted.

16.2) Set the container mapping of *definition* to *container*.

17) If *value* contains the key `@language` and does not contain the key `@type`:

17.1) Initialize *language* to the value associated with the `@language` key, which must be either null or a string. Otherwise, an `invalid language mapping` error has been detected and processing is aborted.

17.2) If *language* is a string set it to lowercased *language*. Set the language mapping of *definition* to *language*.

18) Set the term definition of *term* in active context to *definition* and set the value associated with *defined*'s key *term* to true.

6.3 IRI Expansion

In JSON-LD documents, some keys and values may represent IRIs. This section defines an algorithm for transforming a string that represents an IRI into an absolute IRI or blank node identifier. It also covers transforming keyword aliases into keywords.

IRI expansion may occur during context processing or during any of the other JSON-LD algorithms. If IRI expansion occurs during context processing, then the local context and its related *defined* map from the Context Processing algorithm are passed to this algorithm. This allows for term definition dependencies to be processed via the Create Term Definition algorithm.

Overview

This section is non-normative.

In order to expand *value* to an absolute IRI, we must first determine if it is null, a term, a keyword alias, or some form of IRI. Based on what we find, we handle the specific kind of expansion; for example, we expand a keyword alias to a keyword and a term to an absolute IRI according to its IRI mapping in the active context. While inspecting *value* we may also find that we need to create term definition dependencies because we're running this algorithm during context processing. We can tell whether or not we're running during context processing by checking local context against null. We know we need to create a term definition in the active context when *value* is a key in the local context and the *defined* map does not have a key for *value* with an associated value of true. The *defined* map is used during Context Processing to keep track of which terms have already been defined or are in the process of being defined. We create a term definition by using the Create Term Definition algorithm.

Algorithm

The algorithm takes two required and four optional input variables. The required inputs are an active context and a *value* to be expanded. The optional inputs are two flags, *document relative* and *vocab*, that specifying whether *value* can be interpreted as a relative IRI against the document's base IRI or the active context's vocabulary mapping, respectively, and a local context and a map *defined* to be used when this algorithm is used during Context Processing. If not passed, the two flags are set to `false` and local context and *defined* are initialized to null.

1) If *value* is a keyword or null, return *value* as is.

2) If local context is not null, it contains a key that equals *value*, and the value associated with the key that equals *value* in *defined* is not true, invoke the Create Term Definition algorithm, passing active context, local context, *value* as *term*, and *defined*. This will ensure that a term definition is created for *value* in active context during Context Processing.

3) If *vocab* is true and the active context has a term definition for *value*, return the associated IRI mapping.

4) If *value* contains a colon (`:`), it is either an absolute IRI, a compact IRI, or a blank node identifier:

4.1) Split *value* into a prefix and *suffix* at the first occurrence of a colon (`:`).

4.2) If prefix is underscore (_) or *suffix* begins with double-forward-slash (`//`), return *value* as it is already an absolute IRI or a blank node identifier.

4.3) If local context is not null, it contains a key that equals prefix, and the value associated with the key that equals *prefix* in *defined* is not true, invoke the Create Term Definition algorithm, passing active context, local context, *prefix* as *term*, and *defined*. This will ensure that a term definition is created for prefix in active context during Context Processing.

4.4) If active context contains a term definition for prefix, return the result of concatenating the IRI mapping associated with prefix and *suffix*.

4.5) Return *value* as it is already an absolute IRI.

5) If *vocab* is true, and active context has a vocabulary mapping, return the result of concatenating the vocabulary mapping with *value*.

6) Otherwise, if *document relative* is true, set *value* to the result of resolving *value* against the base IRI. Only the basic algorithm in *section 5.2* of [RFC3986] is used; neither *Syntax-Based Normalization* nor *Scheme-Based Normalization* are performed. Characters additionally allowed in IRI references are treated in the same way that unreserved characters are treated in URI references, per *section 6.5* of [RFC3987].

7) Return *value* as is.

7. Expansion Algorithms

7.1 Expansion Algorithm

This algorithm expands a JSON-LD document, such that all context definitions are removed, all terms and compact IRIs are expanded to absolute IRIs, blank node identifiers, or keywords and all JSON-LD values are expressed in arrays in expanded form.

Overview

This section is non-normative.

Starting with its root *element*, we can process the JSON-LD document recursively, until we have a fully expanded *result*. When expanding an *element*, we can treat each one differently according to its type, in order to break down the problem:

1. If the *element* is null, there is nothing to expand.
2. Otherwise, if *element* is a scalar, we expand it according to the Value Expansion algorithm.
3. Otherwise, if the *element* is an array, then we expand each of its items recursively and return them in a new array.
4. Otherwise, *element* is a JSON object. We expand each of its keys, adding them to our *result*, and then we expand each value for each key recursively. Some of the keys will be terms or compact IRIs and others will be keywords or simply ignored because they do not have definitions in the context. Any IRIs will be expanded using the IRI Expansion algorithm.

Finally, after ensuring *result* is in an array, we return *result*.

Algorithm

The algorithm takes three input variables: an active context, an active property, and an *element* to be expanded. To begin, the active property is set to null, and *element* is set to the JSON-LD input.

1) If *element* is null, return null.

2) If *element* is a scalar,

2.1) If active property is null or `@graph`, drop the free-floating scalar by returning null.

2.2) Return the result of the Value Expansion algorithm, passing the active context, active property, and *element* as *value*.

3) If *element* is an array,

3.1) Initialize an empty array, *result*.

3.2) For each *item* in *element*:

3.2.1) Initialize *expanded item* to the result of using this algorithm recursively, passing active context, active property, and *item* as *element*.

3.2.2) If the active property is `@list` or its container mapping is set to `@list`, the *expanded item* must not be an array or a list object, otherwise a `list of lists` error has been detected and processing is aborted.

3.2.3) If *expanded item* is an array, append each of its items to *result*. Otherwise, if *expanded item* is not null, append it to *result*.

3.3) Return *result*.

4) Otherwise *element* is a JSON object.

5) If *element* contains the key `@context`, set active context to the result of the Context Processing algorithm, passing active context and the value of the `@context` key as local context.

6) Initialize an empty JSON object, *result*.

7) For each *key* and *value* in *element*, ordered lexicographically by *key*:

7.1) If *key* is `@context`, continue to the next *key*.

7.2) Set *expanded property* to the result of using the IRI Expansion algorithm, passing active context, *key* for *value*, and true for *vocab*.

7.3) If *expanded property* is null or it neither contains a colon (`:`) nor it is a keyword, drop *key* by continuing to the next *key*.

7.4) If *expanded property* is a keyword:

7.4.1) If active property equals `@reverse`, an `invalid reverse property map` error has been detected and processing is aborted.

7.4.2) If *result* has already an *expanded property* member, an `colliding keywords` error has been detected and processing is aborted.

7.4.3) If *expanded property* is `@id` and *value* is not a string, an `invalid @id value` error has been detected and processing is aborted. Otherwise, set *expanded value* to the result of using the IRI Expansion algorithm, passing active context, *value*, and true for *document relative*.

7.4.4) If *expanded property* is `@type` and *value* is neither a string nor an array of strings, an `invalid type value` error has been detected and processing is aborted. Otherwise, set *expanded value* to the result of using the IRI Expansion algorithm, passing active context, true for *vocab*, and true for *document relative* to expand the *value* or each of its items.

7.4.5) If *expanded property* is `@graph`, set *expanded value* to the result of using this algorithm recursively passing active context, `@graph` for active property, and *value* for *element*.

7.4.6) If *expanded property* is `@value` and *value* is not a scalar or null, an `invalid value object value` error has been detected and processing is aborted. Otherwise, set *expanded value* to *value*. If *expanded value* is null, set the `@value` member of *result* to null and continue with the next *key* from *element*. Null values need to be preserved in this case as the meaning of an `@type` member depends on the existence of an `@value` member.

7.4.7) If *expanded property* is `@language` and *value* is not a string, an `invalid language-tagged string` error has been detected and processing is aborted. Otherwise, set *expanded value* to lowercased *value*.

7.4.8) If *expanded property* is `@index` and *value* is not a string, an `invalid @index value` error has been detected and processing is aborted. Otherwise, set *expanded value* to *value*.

7.4.9) If *expanded property* is `@list`:

7.4.9.1) If active property is null or `@graph`, continue with the next *key* from *element* to remove the free-floating list.

7.4.9.2) Otherwise, initialize *expanded value* to the result of using this algorithm recursively passing active context, active property, and *value* for *element*.

7.4.9.3) If *expanded value* is a list object, a `list of lists` error has been detected and processing is aborted.

7.4.10) If *expanded property* is `@set`, set *expanded value* to the result of using this algorithm recursively, passing active context, active property, and *value* for *element*.

7.4.11) If *expanded property* is `@reverse` and *value* is not a JSON object, an `invalid @reverse value` error has been detected and processing is aborted. Otherwise

7.4.11.1) Initialize *expanded value* to the result of using this algorithm recursively, passing active context, `@reverse` as active property, and *value* as *element*.

7.4.11.2) If *expanded value* contains an `@reverse` member, i.e., properties that are reversed twice, execute for each of its *property* and *item* the following steps:

7.4.11.2.1) If *result* does not have a *property* member, create one and set its value to an empty array.

7.4.11.2.2) Append *item* to the value of the *property* member of *result*.

7.4.11.3) If *expanded value* contains members other than `@reverse`:

7.4.11.3.1) If *result* does not have an `@reverse` member, create one and set its value to an empty JSON object.

7.4.11.3.2) Reference the value of the `@reverse` member in *result* using the variable *reverse map*.

7.4.11.3.3) For each *property* and *items* in *expanded value* other than `@reverse`:

7.4.11.3.3.1) For each *item* in *items*:

7.4.11.3.3.1.1) If *item* is a value object or list object, an `invalid reverse property value` has been detected and processing is aborted.

7.4.11.3.3.1.2) If *reverse map* has no *property* member, create one and initialize its value to an empty array.

7.4.11.3.3.1.3) Append *item* to the value of the *property* member in *reverse map*.

7.4.11.4) Continue with the next *key* from *element*.

7.4.12) Unless *expanded value* is null, set the *expanded property* member of *result* to *expanded value*.

7.4.13) Continue with the next *key* from *element*.

7.5) Otherwise, if *key*'s container mapping in active context is `@language` and *value* is a JSON object then *value* is expanded from a language map as follows:

7.5.1) Initialize *expanded value* to an empty array.

7.5.2) For each key-value pair *language*-*language value* in *value*, ordered lexicographically by *language*:

7.5.2.1) If *language value* is not an array set it to an array containing only *language value*.

7.5.2.2) For each *item* in *language value*:

7.5.2.2.1) *item* must be a string, otherwise an `invalid language map value` error has been detected and processing is aborted.

7.5.2.2.2) Append a JSON object to *expanded value* that consists of two key-value pairs: (`@value`-*item*) and (`@language`-lowercased *language*).

7.6) Otherwise, if *key*'s container mapping in active context is `@index` and *value* is a JSON object then *value* is expanded from an index map as follows:

7.6.1) Initialize *expanded value* to an empty array.

7.6.2) For each key-value pair *index*-*index value* in *value*, ordered lexicographically by *index*:

7.6.2.1) If *index value* is not an array set it to an array containing only *index value*.

7.6.2.2) Initialize *index value* to the result of using this algorithm recursively, passing active context, *key* as active property, and *index value* as *element*.

7.6.2.3) For each *item* in *index value*:

7.6.2.3.1) If *item* does not have the key `@index`, add the key-value pair (`@index`-*index*) to *item*.

7.6.2.3.2) Append *item* to *expanded value*.

7.7) Otherwise, initialize *expanded value* to the result of using this algorithm recursively, passing active context, *key* for active property, and *value* for *element*.

7.8) If *expanded value* is null, ignore *key* by continuing to the next *key* from *element*.

7.9) If the container mapping associated to *key* in active context is `@list` and *expanded value* is not already a list object, convert *expanded value* to a list object by first setting it to an array containing only *expanded value* if it is not already an array, and then by setting it to a JSON object containing the key-value pair `@list`-*expanded value*.

7.10) Otherwise, if the term definition associated to *key* indicates that it is a reverse property

7.10.1) If *result* has no `@reverse` member, create one and initialize its value to an empty JSON object.

7.10.2) Reference the value of the `@reverse` member in *result* using the variable *reverse map*.

7.10.3) If *expanded value* is not an array, set it to an array containing *expanded value*.

7.10.4) For each *item* in *expanded value*

7.10.4.1) If *item* is a value object or list object, an `invalid reverse property value` has been detected and processing is aborted.

7.10.4.2) If *reverse map* has no *expanded property* member, create one and initialize its value to an empty array.

7.10.4.3) Append *item* to the value of the *expanded property* member of *reverse map*.

7.11) Otherwise, if *key* is not a reverse property:

7.11.1) If *result* does not have an *expanded property* member, create one and initialize its value to an empty array.

7.11.2) Append *expanded value* to value of the *expanded property* member of *result*.

8) If *result* contains the key `@value`:

8.1) The *result* must not contain any keys other than `@value`, `@language`, `@type`, and `@index`. It must not contain both the `@language` key and the `@type` key. Otherwise, an `invalid value object` error has been detected and processing is aborted.

8.2) If the value of *result*'s `@value` key is null, then set *result* to null.

8.3) Otherwise, if the value of *result*'s `@value` member is not a string and *result* contains the key `@language`, an `invalid language-tagged value` error has been detected (only strings can be language-tagged) and processing is aborted.

8.4) Otherwise, if the *result* has an `@type` member and its value is not an IRI, an `invalid typed value` error has been detected and processing is aborted.

9) Otherwise, if *result* contains the key `@type` and its associated value is not an array, set it to an array containing only the associated value.

10) Otherwise, if *result* contains the key `@set` or `@list`:

10.1) The *result* must contain at most one other key and that key must be `@index`. Otherwise, an `invalid set or list object` error has been detected and processing is aborted.

10.2) If *result* contains the key `@set`, then set *result* to the key's associated value.

11) If *result* contains only the key `@language`, set *result* to null.

12) If active property is null or `@graph`, drop free-floating values as follows:

12.1) If *result* is an empty JSON object or contains the keys `@value` or `@list`, set *result* to null.

12.2) Otherwise, if *result* is a JSON object whose only key is `@id`, set *result* to null.

13) Return *result*.

If, after the above algorithm is run, the result is a JSON object that contains only an `@graph` key, set the result to the value of `@graph`'s value. Otherwise, if the result is null, set it to an empty array. Finally, if the result is not an array, then set the result to an array containing only the result.

7.2 Value Expansion

Some values in JSON-LD can be expressed in a compact form. These values are required to be expanded at times when processing JSON-LD documents. A value is said to be in ***expanded form*** after the application of this algorithm.

Overview

This section is non-normative.

If active property has a type mapping in the active context set to `@id` or `@vocab`, a JSON object with a single member `@id` whose value is the result of using the IRI Expansion algorithm on *value* is returned.

Otherwise, the result will be a JSON object containing an `@value` member whose value is the passed *value*. Additionally, an `@type` member will be included if there is a type mapping associated with the active property or an `@language` member if *value* is a string and there is language mapping associated with the active property.

Algorithm

The algorithm takes three required inputs: an active context, an active property, and a *value* to expand.

1) If the active property has a type mapping in active context that is `@id`, return a new JSON object containing a single key-value pair where the key is `@id` and the value is the result of using the IRI Expansion algorithm, passing active context, *value*, and true for *document relative*.

2) If active property has a type mapping in active context that is `@vocab`, return a new JSON object containing a single key-value pair where the key is `@id` and the value is the result of using the IRI Expansion algorithm, passing active context, *value*, true for *vocab*, and true for *document relative*.

3) Otherwise, initialize *result* to a JSON object with an `@value` member whose value is set to *value*.

4) If active property has a type mapping in active context, add an `@type` member to *result* and set its value to the value associated with the type mapping.

5) Otherwise, if *value* is a string:

5.1) If a language mapping is associated with active property in active context, add an `@language` to *result* and set its value to the language code associated with the language mapping; unless the language mapping is set to null in which case no member is added.

5.2) Otherwise, if the active context has a default language, add an `@language` to *result* and set its value to the default language.

6) Return *result*.

8. Compaction Algorithms

8.1 Compaction Algorithm

This algorithm compacts a JSON-LD document, such that the given context is applied. This must result in shortening any applicable IRIs to terms or compact IRIs, any applicable keywords to keyword aliases, and any applicable JSON-LD values expressed in expanded form to simple values such as strings or numbers.

Overview

This section is non-normative.

Starting with its root *element*, we can process the JSON-LD document recursively, until we have a fully compacted *result*. When compacting an *element*, we can treat each one differently according to its type, in order to break down the problem:

1. If the *element* is a scalar, it is already in compacted form, so we simply return it.
2. If the *element* is an array, we compact each of its items recursively and return them in a new array.
3. Otherwise *element* is a JSON object. The value of each key in element is compacted recursively. Some of the keys will be compacted, using the IRI Compaction algorithm, to terms or compact IRIs and others will be compacted from keywords to

keyword aliases or simply left unchanged because they do not have definitions in the context. Values will be converted to compacted form via the Value Compaction algorithm. Some data will be reshaped based on container mappings specified in the context such as `@index` or `@language` maps.

The final output is a JSON object with an `@context` key, if a non-empty context was given, where the JSON object is either *result* or a wrapper for it where *result* appears as the value of an (aliased) `@graph` key because *result* contained two or more items in an array.

Algorithm

The algorithm takes five required input variables: an active context, an inverse context, an active property, an *element* to be compacted, and a flag `compactArrays`. To begin, the active context is set to the result of performing Context Processing on the passed context, the inverse context is set to the result of performing the Inverse Context Creation algorithm on active context, the active property is set to null, *element* is set to the result of performing the Expansion algorithm on the JSON-LD input, and, if not passed, `compactArrays` is set to true.

1) If *element* is a scalar, it is already in its most compact form, so simply return *element*.

2) If *element* is an array:

2.1) Initialize *result* to an empty array.

2.2) For each *item* in *element*:

2.2.1) Initialize *compacted item* to the result of using this algorithm recursively, passing active context, inverse context, active property, and *item* for *element*.

2.2.2) If *compacted item* is not null, then append it to *result*.

2.3) If *result* contains only one item (it has a length of `1`), active property has no container mapping in active context, and `compactArrays` is true, set *result* to its only item.

2.4) Return *result*.

3) Otherwise *element* is a JSON object.

4) If *element* has an `@value` or `@id` member and the result of using the Value Compaction algorithm, passing active context, inverse context, active property,and *element* as *value* is a scalar, return that result.

5) Initialize *inside reverse* to true if active property equals `@reverse`, otherwise to false.

6) Initialize *result* to an empty JSON object.

7) For each key *expanded property* and value *expanded value* in *element*, ordered lexicographically by *expanded property*:

7.1) If *expanded property* is `@id` or `@type`:

7.1.1) If *expanded value* is a string, then initialize *compacted value* to the result of using the IRI Compaction algorithm, passing active context, inverse context, *expanded value* for *iri*, and true for *vocab* if *expanded property* is `@type`, false otherwise.

7.1.2) Otherwise, *expanded value* must be a `@type` array:

7.1.2.1) Initialize *compacted value* to an empty array.

7.1.2.2) For each item *expanded type* in *expanded value*, append the result of of using the IRI Compaction algorithm, passing active context, inverse context, *expanded type* for *iri*, and true for *vocab*, to *compacted value*.

7.1.2.3) If *compacted value* contains only one item (it has a length of `1`), then set *compacted value* to its only item.

7.1.3) Initialize *alias* to the result of using the IRI Compaction algorithm, passing active context, inverse context, *expanded property* for *iri*, and true for *vocab*.

7.1.4) Add a member *alias* to *result* whose value is set to *compacted value* and continue to the next *expanded property*.

7.2) If *expanded property* is `@reverse`:

7.2.1) Initialize *compacted value* to the result of using this algorithm recursively, passing active context, inverse context, `@reverse` for active property, and *expanded value* for *element*.

7.2.2) For each *property* and *value* in *compacted value*:

7.2.2.1) If the term definition for *property* in the active context indicates that *property* is a reverse property

7.2.2.1.1) If the term definition for *property* in the active context has a container mapping of `@set` or `compactArrays` is false, and *value* is not an array, set *value* to a new array containing only *value*.

7.2.2.1.2) If *property* is not a member of *result*, add one and set its value to *value*.

7.2.2.1.3) Otherwise, if the value of the *property* member of *result* is not an array, set it to a new array containing only the value. Then append *value* to its value if *value* is not an array, otherwise append each of its items.

7.2.2.1.4) Remove the *property* member from *compacted value*.

7.2.3) If *compacted value* has some remaining members, i.e., it is not an empty JSON object:

7.2.3.1) Initialize *alias* to the result of using the IRI Compaction algorithm, passing active context, inverse context, `@reverse` for *iri*, and true for *vocab*.

7.2.3.2) Set the value of the *alias* member of *result* to *compacted value*.

7.2.4) Continue with the next *expanded property* from *element*.

7.3) If *expanded property* is `@index` and active property has a container mapping in active context that is `@index`, then the compacted result will be inside of an `@index` container, drop the `@index` property by continuing to the next *expanded property*.

7.4) Otherwise, if *expanded property* is `@index`, `@value`, or `@language`:

7.4.1) Initialize *alias* to the result of using the IRI Compaction algorithm, passing active context, inverse context, *expanded property* for *iri*, and true for *vocab*.

7.4.2) Add a member *alias* to *result* whose value is set to *expanded value* and continue with the next *expanded property*.

7.5) If *expanded value* is an empty array:

7.5.1) Initialize *item active property* to the result of using the IRI Compaction algorithm, passing active context, inverse context, *expanded property* for *iri*, *expanded value* for *value*, true for *vocab*, and *inside reverse*.

7.5.2) If *result* does not have the key that equals *item active property*, set this key's value in *result* to an empty array. Otherwise, if the key's value is not an array, then set it to one containing only the value.

7.6) At this point, *expanded value* must be an array due to the Expansion algorithm. For each item *expanded item* in *expanded value*:

7.6.1) Initialize *item active property* to the result of using the IRI Compaction algorithm, passing active context, inverse context, *expanded property* for *iri*, *expanded item* for *value*, true for *vocab*, and *inside reverse*.

7.6.2) Initialize *container* to null. If there is a container mapping for *item active property* in active context, set *container* to its value.

7.6.3) Initialize *compacted item* to the result of using this algorithm recursively, passing active context, inverse context, *item active property* for active property, *expanded item* for *element* if it does not contain the key `@list`, otherwise pass the key's associated value for *element*.

7.6.4) If *expanded item* is a list object:

7.6.4.1) If *compacted item* is not an array, then set it to an array containing only *compacted item*.

7.6.4.2) If *container* is not `@list`:

7.6.4.2.1) Convert *compacted item* to a list object by setting it to a JSON object containing key-value pair where the key is the result of the IRI Compaction algorithm, passing active context, inverse context, `@list` for *iri*, and *compacted item* for *value*.

7.6.4.2.2) If *expanded item* contains the key `@index`, then add a key-value pair to *compacted item* where the key is the result of the IRI Compaction algorithm, passing active context, inverse context, `@index` as *iri*, and the value associated with the `@index` key in *expanded item* as *value*.

7.6.4.3) Otherwise, *item active property* must not be a key in *result* because there cannot be two list objects associated with an active property that has a container mapping; a `compaction to list of lists` error has been detected and processing is aborted.

7.6.5) If *container* is `@language` or `@index`:

7.6.5.1) If *item active property* is not a key in *result*, initialize it to an empty JSON object. Initialize *map object* to the value of *item active property* in *result*.

7.6.5.2) If *container* is `@language` and *compacted item* contains the key `@value`, then set *compacted item* to the value associated with its `@value` key.

7.6.5.3) Initialize *map key* to the value associated with with the key that equals *container* in *expanded item*.

7.6.5.4) If *map key* is not a key in *map object*, then set this key's value in *map object* to *compacted item*. Otherwise, if the value is not an array, then set it to one containing only the value and then append *compacted item* to it.

7.6.6) Otherwise,

7.6.6.1) If `compactArrays` is false, *container* is `@set` or `@list`, or *expanded property* is `@list` or `@graph` and *compacted item* is not an array, set it to a new array containing only *compacted item*.

7.6.6.2) If *item active property* is not a key in *result* then add the key-value pair, (*item active property*-*compacted item*), to *result*.

7.6.6.3) Otherwise, if the value associated with the key that equals *item active property* in *result* is not an array, set it to a new array containing only the value. Then append *compacted item* to the value if *compacted item* is not an array, otherwise, concatenate it.

8) Return *result*.

If, after the algorithm outlined above is run, the result *result* is an array, replace it with a new JSON object with a single member whose key is the result of using the IRI Compaction algorithm, passing active context, inverse context, and `@graph` as *iri* and whose value is the array *result*. Finally, if a non-empty *context* has been passed, add an `@context` member to *result* and set its value to the passed *context*.

8.2 Inverse Context Creation

When there is more than one term that could be chosen to compact an IRI, it has to be ensured that the term selection is both deterministic and represents the most context-appropriate choice whilst taking into consideration algorithmic complexity.

In order to make term selections, the concept of an inverse context is introduced. An ***inverse context*** is essentially a reverse lookup table that maps container mappings, type mappings, and language mappings to a simple term for a given active context. A inverse context only needs to be generated for an active context if it is being used for compaction.

To make use of an inverse context, a list of preferred container mappings and the type mapping or language mapping are gathered for a particular value associated with an IRI. These parameters are then fed to the Term Selection algorithm, which will find the term that most appropriately matches the value's mappings.

Overview

This section is non-normative.

To create an inverse context for a given active context, each term in the active context is visited, ordered by length, shortest first (ties are broken by choosing the lexicographically least term). For each term, an entry is added to the inverse context for each possible combination of container mapping and type mapping or language mapping that would legally match the term. Illegal matches include differences between a value's type mapping or language mapping and that of the term. If a term has no container mapping, type mapping, or language mapping (or some combination of these), then it will have an entry in the inverse context using the special key `@none`. This allows the Term Selection algorithm to fall back to choosing more generic terms when a more specifically-matching term is not available for a particular IRI and value combination.

Algorithm

The algorithm takes one required input: the active context that the inverse context is being created for.

1) Initialize *result* to an empty JSON object.

2) Initialize *default language* to `@none`. If the active context has a default language, set *default language* to it.

3) For each key term and value term definition in the active context, ordered by shortest term first (breaking ties by choosing the lexicographically least term):

3.1) If the term definition is null, term cannot be selected during compaction, so continue to the next term.

3.2) Initialize *container* to `@none`. If there is a container mapping in term definition, set *container* to its associated value.

3.3) Initialize *iri* to the value of the IRI mapping for the term definition.

3.4) If *iri* is not a key in *result*, add a key-value pair where the key is *iri* and the value is an empty JSON object to *result*.

3.5) Reference the value associated with the *iri* member in *result* using the variable *container map*.

3.6) If *container map* has no *container* member, create one and set its value to a new JSON object with two members. The first member is `@language` and its value is a new empty JSON object, the second member is `@type` and its value is a new empty JSON object.

3.7) Reference the value associated with the *container* member in *container map* using the variable *type/language map*.

3.8) If the term definition indicates that the term represents a reverse property:

3.8.1) Reference the value associated with the `@type` member in *type/language map* using the variable *type map*.

3.8.2) If *type map* does not have an `@reverse` member, create one and set its value to the term being processed.

3.9) Otherwise, if term definition has a type mapping:

3.9.1) Reference the value associated with the `@type` member in *type/language map* using the variable *type map*.

3.9.2) If *type map* does not have a member corresponding to the type mapping in term definition, create one and set its value to the term being processed.

3.10) Otherwise, if term definition has a language mapping (might be null):

3.10.1) Reference the value associated with the `@language` member in *type/language map* using the variable *language map*.

3.10.2) If the language mapping equals null, set *language* to `@null`; otherwise set it to the language code in language mapping.

3.10.3) If *language map* does not have a *language* member, create one and set its value to the term being processed.

3.11) Otherwise:

3.11.1) Reference the value associated with the `@language` member in *type/language map* using the variable *language map*.

3.11.2) If *language map* does not have a *default language* member, create one and set its value to the term being processed.

3.11.3) If *language map* does not have an `@none` member, create one and set its value to the term being processed.

3.11.4) Reference the value associated with the `@type` member in *type/language map* using the variable *type map*.

3.11.5) If *type map* does not have an `@none` member, create one and set its value to the term being processed.

4) Return *result*.

8.3 IRI Compaction

This algorithm compacts an IRI to a term or compact IRI, or a keyword to a keyword alias. A value that is associated with the IRI may be passed in order to assist in selecting the most context-appropriate term.

Overview

This section is non-normative.

If the passed IRI is null, we simply return null. Otherwise, we first try to find a term that the IRI or keyword can be compacted to if it is relative to active context's vocabulary mapping. In order to select the most appropriate term, we may have to collect information about the passed *value*. This information includes which container mappings would be preferred for expressing the *value*, and what its type mapping or language mapping is. For JSON-LD lists, the type mapping or language mapping will be chosen based on the most specific values that work for all items in the list. Once this information is gathered, it is passed to the Term Selection algorithm, which will return the most appropriate term to use.

If no term was found that could be used to compact the IRI, an attempt is made to compact the IRI using the active context's vocabulary mapping, if there is one. If the IRI could not be compacted, an attempt is made to find a compact IRI. If there is no appropriate compact IRI, the IRI is transformed to a relative IRI using the document's base IRI. Finally, if the IRI or keyword still could not be compacted, it is returned as is.

Algorithm

This algorithm takes three required inputs and three optional inputs. The required inputs are an active context, an inverse context, and the *iri* to be compacted. The optional inputs are a *value* associated with the *iri*, a *vocab* flag which specifies whether the passed *iri* should be compacted using the active context's vocabulary mapping, and a *reverse* flag which specifies whether a reverse property is being compacted. If not passed, *value* is set to null and *vocab* and *reverse* are both set to `false`.

1) If *iri* is null, return null.

2) If *vocab* is true and *iri* is a key in inverse context:

2.1) Initialize *default language* to active context's default language, if it has one, otherwise to `@none`.

2.2) Initialize *containers* to an empty array. This array will be used to keep track of an ordered list of preferred container mappings for a term, based on what is compatible with *value*.

2.3) Initialize *type/language* to `@language`, and *type/language value* to `@null`. These two variables will keep track of the preferred type mapping or language mapping for a term, based on what is compatible with *value*.

2.4) If *value* is a JSON object that contains the key `@index`, then append the value `@index` to *containers*.

2.5) If *reverse* is true, set *type/language* to `@type`, *type/language value* to `@reverse`, and append `@set` to *containers*.

2.6) Otherwise, if *value* is a list object, then set *type/language* and *type/language value* to the most specific values that work for all items in the list as follows:

2.6.1) If `@index` is a not key in *value*, then append `@list` to *containers*.

2.6.2) Initialize *list* to the array associated with the key `@list` in *value*.

2.6.3) Initialize *common type* and *common language* to null. If *list* is empty, set *common language* to *default language*.

2.6.4) For each *item* in *list*:

2.6.4.1) Initialize *item language* to `@none` and *item type* to `@none`.

2.6.4.2) If *item* contains the key `@value`:

2.6.4.2.1) If *item* contains the key `@language`, then set *item language* to its associated value.

2.6.4.2.2) Otherwise, if *item* contains the key `@type`, set *item type* to its associated value.

2.6.4.2.3) Otherwise, set *item language* to `@null`.

2.6.4.3) Otherwise, set *item type* to `@id`.

2.6.4.4) If *common language* is null, set it to *item language*.

2.6.4.5) Otherwise, if *item language* does not equal *common language* and *item* contains the key `@value`, then set *common language* to `@none` because list items have conflicting languages.

2.6.4.6) If *common type* is null, set it to *item type*.

2.6.4.7) Otherwise, if *item type* does not equal *common type*, then set *common type* to `@none` because list items have conflicting types.

2.6.4.8) If *common language* is `@none` and *common type* is `@none`, then stop processing items in the list because it has been detected that there is no common language or type amongst the items.

2.6.5) If *common language* is null, set it to `@none`.

2.6.6) If *common type* is null, set it to `@none`.

2.6.7) If *common type* is not `@none` then set *type/language* to `@type` and *type/language value* to *common type*.

2.6.8) Otherwise, set *type/language value* to *common language*.

2.7) Otherwise:

2.7.1) If *value* is a value object:

2.7.1.1) If *value* contains the key `@language` and does not contain the key `@index`, then set *type/language value* to its associated value and append `@language` to *containers*.

2.7.1.2) Otherwise, if *value* contains the key `@type`, then set *type/language value* to its associated value and set *type/language* to `@type`.

2.7.2) Otherwise, set *type/language* to `@type` and set *type/language value* to `@id`.

2.7.3) Append `@set` to *containers*.

2.8) Append `@none` to *containers*. This represents the non-existence of a container mapping, and it will be the last container mapping value to be checked as it is the most generic.

2.9) If *type/language value* is null, set it to `@null`. This is the key under which null values are stored in the inverse context *entry*.

2.10) Initialize *preferred values* to an empty array. This array will indicate, in order, the preferred values for a term's type mapping or language mapping.

2.11) If *type/language value* is `@reverse`, append `@reverse` to *preferred values*.

2.12) If *type/language value* is `@id` or `@reverse` and *value* has an `@id` member:

2.12.1) If the result of using the IRI compaction algorithm, passing active context, inverse context, the value associated with the `@id` key in *value* for *iri*, true for *vocab*, and true for *document relative* has a term definition in the active context with an IRI mapping that equals the value associated with the `@id` key in *value*, then append `@vocab`, `@id`, and `@none`, in that order, to *preferred values*.

2.12.2) Otherwise, append `@id`, `@vocab`, and `@none`, in that order, to *preferred values*.

2.13) Otherwise, append *type/language value* and `@none`, in that order, to *preferred values*.

2.14) Initialize *term* to the result of the Term Selection algorithm, passing inverse context, *iri*, *containers*, *type/language*, and *preferred values*.

2.15) If *term* is not null, return *term*.

3) At this point, there is no simple term that *iri* can be compacted to. If *vocab* is true and active context has a vocabulary mapping:

3.1) If *iri* begins with the vocabulary mapping's value but is longer, then initialize *suffix* to the substring of *iri* that does not match. If *suffix* does not have a term definition in active context, then return *suffix*.

4) The *iri* could not be compacted using the active context's vocabulary mapping. Try to create a compact IRI, starting by initializing *compact IRI* to null. This variable will be used to tore the created compact IRI, if any.

5) For each key term and value term definition in the active context:

5.1) If the term contains a colon (`:`), then continue to the next term because terms with colons can't be used as prefixes.

5.2) If the term definition is null, its IRI mapping equals *iri*, or its IRI mapping is not a substring at the beginning of *iri*, the term cannot be used as a prefix because it is not a partial match with *iri*. Continue with the next term.

5.3) Initialize *candidate* by concatenating term, a colon (`:`), and the substring of *iri* that follows after the value of the term definition's IRI mapping.

5.4) If either *compact IRI* is null or *candidate* is shorter or the same length but lexicographically less than *compact IRI* and *candidate* does not have a term definition in active context or if the term definition has an IRI mapping that equals *iri* and *value* is null, set *compact IRI* to *candidate*.

6) If *compact IRI* is not null, return *compact IRI*.

7) If *vocab* is false then transform *iri* to a relative IRI using the document's base IRI.

8) Finally, return *iri* as is.

8.4 Term Selection

This algorithm, invoked via the IRI Compaction algorithm, makes use of an active context's inverse context to find the term that is best used to compact an IRI. Other information about a value associated with the IRI is given, including which container mappings and which type mapping or language mapping would be best used to express the value.

Overview

This section is non-normative.

The inverse context's entry for the IRI will be first searched according to the preferred container mappings, in the order that they are given. Amongst terms with a matching container mapping, preference will be given to those with a matching type mapping or language mapping, over those without a type mapping or language mapping. If there is no term with a matching container mapping then the term without a container mapping that matches the given type mapping or language mapping is selected. If there is still no selected term, then a term with no type mapping or language mapping will be selected if available. No term will be selected that has a conflicting type mapping or language mapping. Ties between terms that have the same mappings are resolved by first choosing the shortest terms, and then by choosing the lexicographically least term. Note that these ties are resolved automatically because they were previously resolved when the Inverse Context Creation algorithm was used to create the inverse context.

Algorithm

This algorithm has five required inputs. They are: an inverse context, a keyword or IRI *iri*, an array *containers* that represents an ordered list of preferred container mappings, a string *type/language* that indicates whether to look for a term with a matching type mapping or language mapping, and an array representing an ordered list of *preferred values* for the type mapping or language mapping to look for.

1) Initialize *container map* to the value associated with *iri* in the inverse context.

2) For each item *container* in *containers*:

2.1) If *container* is not a key in *container map*, then there is no term with a matching container mapping for it, so continue to the next *container*.

2.2) Initialize *type/language map* to the value associated with the *container* member in *container map*.

2.3) Initialize *value map* to the value associated with *type/language* member in *type/language map*.

2.4) For each *item* in *preferred values*:

2.4.1) If *item* is not a key in *value map*, then there is no term with a matching type mapping or language mapping, so continue to the next *item*.

2.4.2) Otherwise, a matching term has been found, return the value associated with the *item* member in *value map*.

3) No matching term has been found. Return null.

8.5 Value Compaction

Expansion transforms all values into expanded form in JSON-LD. This algorithm performs the opposite operation, transforming a value into ***compacted form***. This algorithm compacts a value according to the term definition in the given active context that is associated with the value's associated active property.

Overview

This section is non-normative.

The *value* to compact has either an `@id` or an `@value` member.

For the former case, if the type mapping of active property is set to `@id` or `@vocab` and *value* consists of only an `@id` member and, if the container mapping of active property is set to `@index`, an `@index` member, *value* can be compacted to a string by returning the result of using the IRI Compaction algorithm to compact the value associated with the `@id` member. Otherwise, *value* cannot be compacted and is returned as is.

For the latter case, it might be possible to compact *value* just into the value associated with the `@value` member. This can be done if the active property has a matching type mapping or language mapping and there is either no `@index` member or the container mapping of active property is set to `@index`. It can also be done if `@value` is the only member in *value* (apart an `@index` member in case the container mapping of active property is set to `@index`) and either its associated value is not a string, there is no default language, or there is an explicit null language mapping for the active property.

Algorithm

This algorithm has four required inputs: an active context, an inverse context, an active property, and a *value* to be compacted.

1) Initialize *number members* to the number of members *value* contains.

2) If *value* has an `@index` member and the container mapping associated to active property is set to `@index`, decrease *number members* by `1`.

3) If *number members* is greater than `2`, return *value* as it cannot be compacted.

4) If *value* has an `@id` member:

- **4.1)** If *number members* is `1` and the type mapping of active property is set to `@id`, return the result of using the IRI compaction algorithm, passing active context, inverse context, and the value of the `@id` member for *iri*.
- **4.2)** Otherwise, if *number members* is `1` and the type mapping of active property is set to `@vocab`, return the result of using the IRI compaction algorithm, passing active context, inverse context, the value of the `@id` member for *iri*, and true for *vocab*.
- **4.3)** Otherwise, return *value* as is.

5) Otherwise, if *value* has an `@type` member whose value matches the type mapping of active property, return the value associated with the `@value` member of *value*.

6) Otherwise, if *value* has an `@language` member whose value matches the language mapping of active property, return the value associated with the `@value` member of *value*.

7) Otherwise, if *number members* equals `1` and either the value of the `@value` member is not a string, or the active context has no default language, or the language mapping of active property is set to null,, return the value associated with the `@value` member.

8) Otherwise, return *value* as is.

9. Flattening Algorithms

9.1 Flattening Algorithm

This algorithm flattens an expanded JSON-LD document by collecting all properties of a node in a single JSON object and labeling all blank nodes with blank node identifiers. This resulting uniform shape of the document, may drastically simplify the code required to process JSON-LD data in certain applications.

Overview

This section is non-normative.

First, a *node map* is generated using the Node Map Generation algorithm which collects all properties of a node in a single JSON object. In the next step, the *node map* is converted to a JSON-LD document in flattened document form. Finally, if a context has been passed, the flattened document is compacted using the Compaction algorithm before being returned.

Algorithm

The algorithm takes two input variables, an *element* to flatten and an optional *context* used to compact the flattened document. If not passed, *context* is set to null.

This algorithm generates new blank node identifiers and relabels existing blank node identifiers. The used Generate Blank Node Identifier algorithm keeps an *identifier map* and a *counter* to ensure consistent relabeling and avoid collisions. Thus, before this algorithm is run, the *identifier map* is reset and the *counter* is initialized to `0`.

1) Initialize *node map* to a JSON object consisting of a single member whose key is `@default` and whose value is an empty JSON object.

2) Perform the Node Map Generation algorithm, passing *element* and *node map*.

3) Initialize *default graph* to the value of the `@default` member of *node map*, which is a JSON object representing the default graph.

4) For each key-value pair *graph name*-*graph* in *node map* where *graph name* is not `@default`, perform the following steps:

- **4.1)** If *default graph* does not have a *graph name* member, create one and initialize its value to a JSON object consisting of an `@id` member whose value is set to *graph name*.

4.2) Reference the value associated with the *graph name* member in *default graph* using the variable *entry*.

4.3) Add an `@graph` member to *entry* and set it to an empty array.

4.4) For each *id-node* pair in *graph* ordered by *id*, add *node* to the `@graph` member of *entry*, unless the only member of *node* is `@id`.

5) Initialize an empty array *flattened*.

6) For each *id-node* pair in *default graph* ordered by *id*, add *node* to *flattened*, unless the only member of *node* is `@id`.

7) If *context* is null, return *flattened*.

8) Otherwise, return the result of compacting *flattened* according the Compaction algorithm passing *context* ensuring that the compaction result has only the `@graph` keyword (or its alias) at the top-level other than `@context`, even if the context is empty or if there is only one element to put in the `@graph` array. This ensures that the returned document has a deterministic structure.

9.2 Node Map Generation

This algorithm creates a JSON object *node map* holding an indexed representation of the graphs and nodes represented in the passed expanded document. All nodes that are not uniquely identified by an IRI get assigned a (new) blank node identifier. The resulting *node map* will have a member for every graph in the document whose value is another object with a member for every node represented in the document. The default graph is stored under the `@default` member, all other graphs are stored under their graph name.

Overview

This section is non-normative.

The algorithm recursively runs over an expanded JSON-LD document to collect all properties of a node in a single JSON object. The algorithm constructs a JSON object *node map* whose keys represent the graph names used in the document (the default graph is stored under the key `@default`) and whose associated values are JSON objects which index the nodes in the graph. If a property's value is a node object, it is replaced by a node object consisting of only an `@id` member. If a node object has no `@id` member or it is identified by a blank node identifier, a new blank node identifier is generated. This relabeling of blank node identifiers is also done for properties and values of `@type`.

Algorithm

The algorithm takes as input an expanded JSON-LD document *element* and a reference to a JSON object *node map*. Furthermore it has the optional parameters active graph (which defaults to `@default`), an active subject, active property, and a reference to a JSON object *list*. If not passed, active subject, active property, and *list* are set to null.

1) If *element* is an array, process each *item* in *element* as follows and then return:

1.1) Run this algorithm recursively by passing *item* for *element*, *node map*, active graph, active subject, active property, and *list*.

2) Otherwise *element* is a JSON object. Reference the JSON object which is the value of the active graph member of *node map* using the variable *graph*. If the active subject is null, set *node* to null otherwise reference the active subject member of *graph* using the variable *node*.

3) If *element* has an `@type` member, perform for each *item* the following steps:

3.1) If *item* is a blank node identifier, replace it with a newly generated blank node identifier passing *item* for *identifier*.

4) If *element* has an `@value` member, perform the following steps:

4.1) If *list* is null:

4.1.1) If *node* does not have an active property member, create one and initialize its value to an array containing *element*.

4.1.2) Otherwise, compare *element* against every item in the array associated with the active property member of *node*. If there is no item equivalent to *element*, append *element* to the array. Two JSON objects are considered equal if they have equivalent key-value pairs.

4.2) Otherwise, append *element* to the `@list` member of *list*.

5) Otherwise, if *element* has an `@list` member, perform the following steps:

5.1) Initialize a new JSON object *result* consisting of a single member `@list` whose value is initialized to an empty array.

5.2) Recursively call this algorithm passing the value of *element's* `@list` member for *element*, active graph, active subject, active property, and *result* for *list*.

5.3) Append *result* to the value of the active property member of *node*.

6) Otherwise *element* is a node object, perform the following steps:

6.1) If *element* has an `@id` member, set *id* to its value and remove the member from *element*. If *id* is a blank node identifier, replace it with a newly generated blank node identifier passing *id* for *identifier*.

6.2) Otherwise, set *id* to the result of the Generate Blank Node Identifier algorithm passing null for *identifier*.

6.3) If *graph* does not contain a member *id*, create one and initialize its value to a JSON object consisting of a single member `@id` whose value is *id*.

6.4) Reference the value of the *id* member of *graph* using the variable *node*.

6.5) If active subject is a JSON object, a reverse property relationship is being processed. Perform the following steps:

6.5.1) If *node* does not have an active property member, create one and initialize its value to an array containing active subject.

6.5.2) Otherwise, compare active subject against every item in the array associated with the active property member of *node*. If there is no item equivalent to active subject, append active subject to the array. Two JSON objects are considered equal if they have equivalent key-value pairs.

6.6) Otherwise, if active property is not null, perform the following steps:

6.6.1) Create a new JSON object *reference* consisting of a single member `@id` whose value is *id*.

6.6.2) If *list* is null:

6.6.2.1) If *node* does not have an active property member, create one and initialize its value to an array containing *reference*.

6.6.2.2) Otherwise, compare *reference* against every item in the array associated with the active property member of *node*. If there is no item equivalent to *reference*, append *reference* to the array. Two JSON objects are considered equal if they have equivalent key-value pairs.

6.6.3) Otherwise, append *element* to the `@list` member of *list*.

6.7) If *element* has an `@type` key, append each item of its associated array to the array associated with the `@type` key of *node* unless it is already in that array. Finally remove the `@type` member from *element*.

6.8) If *element* has an `@index` member, set the `@index` member of *node* to its value. If node has already an `@index` member with a different value, a `conflicting indexes` error has been detected and processing is aborted. Otherwise, continue by removing the `@index` member from *element*.

6.9) If *element* has an `@reverse` member:

6.9.1) Create a JSON object *referenced node* with a single member `@id` whose value is *id*.

6.9.2) Set *reverse map* to the value of the `@reverse` member of *element*.

6.9.3) For each key-value pair *property-values* in *reverse map*:

6.9.3.1) For each *value* of *values*:

6.9.3.1.1) Recursively invoke this algorithm passing *value* for *element*, *node map*, active graph, *referenced node* for active subject, and *property* for active property. Passing a JSON object for active subject indicates to the algorithm that a reverse property relationship is being processed.

6.9.4) Remove the `@reverse` member from *element*.

6.10) If *element* has an `@graph` member, recursively invoke this algorithm passing the value of the `@graph` member for *element*, *node map*, and *id* for active graph before removing the `@graph` member from *element*.

6.11) Finally, for each key-value pair *property-value* in *element* ordered by *property* perform the following steps:

6.11.1) If *property* is a blank node identifier, replace it with a newly generated blank node identifier passing *property* for *identifier*.

6.11.2) If *node* does not have a *property* member, create one and initialize its value to an empty array.

6.11.3) Recursively invoke this algorithm passing *value* for *element*, *node map*, active graph, *id* for active subject, and *property* for active property.

9.3 Generate Blank Node Identifier

This algorithm is used to generate new blank node identifiers or to relabel an existing blank node identifier to avoid collision by the introduction of new ones.

Overview

This section is non-normative.

The simplest case is if there exists already a blank node identifier in the *identifier map* for the passed *identifier*, in which case it is simply returned. Otherwise, a new blank node identifier is generated by concatenating the string `_:b` and the *counter*. If the passed *identifier* is not null, an entry is created in the *identifier map* associating the *identifier* with the blank node identifier.

Finally, the *counter* is increased by one and the new blank node identifier is returned.

Algorithm

The algorithm takes a single input variable *identifier* which may be null. Between its executions, the algorithm needs to keep an *identifier map* to relabel existing blank node identifiers consistently and a *counter* to generate new blank node identifiers. The *counter* is initialized to `0` by default.

1) If *identifier* is not null and has an entry in the *identifier map*, return the mapped identifier.

2) Otherwise, generate a new blank node identifier by concatenating the string `_:b` and *counter*.

3) Increment *counter* by `1`.

4) If *identifier* is not null, create a new entry for *identifier* in *identifier map* and set its value to the new blank node identifier.

5) Return the new blank node identifier.

10. RDF Serialization/Deserialization Algorithms

This section describes algorithms to deserialize a JSON-LD document to an RDF dataset and vice versa. The algorithms are designed for in-memory implementations with random access to JSON object elements.

Throughout this section, the following vocabulary prefixes are used in compact IRIs:

Prefix	IRI
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
xsd	http://www.w3.org/2001/XMLSchema#

10.1 Deserialize JSON-LD to RDF algorithm

This algorithm deserializes a JSON-LD document to an RDF dataset. Please note that RDF does not allow a blank node to be used as a property, while JSON-LD does. Therefore, by default RDF triples that would have contained blank nodes as properties are discarded when interpreting JSON-LD as RDF.

Overview

This section is non-normative.

The JSON-LD document is expanded and converted to a *node map* using the Node Map Generation algorithm. This allows each graph represented within the document to be extracted and flattened, making it easier to process each node object. Each graph from the *node map* is processed to extract RDF triples, to which any (non-default) graph name is applied to create an RDF dataset. Each node object in the *node map* has an `@id` member which corresponds to the RDF subject, the other members represent RDF predicates. Each member value is either an IRI or blank node identifier or can be transformed to an RDF literal to generate an RDF triple. Lists are transformed into an RDF Collection using the List to RDF Conversion algorithm.

Algorithm

The algorithm takes a JSON-LD document *element* and returns an RDF dataset. Unless the *produce generalized RDF* flag is set to true, RDF triples containing a blank node predicate are excluded from output.

This algorithm generates new blank node identifiers and relabels existing blank node identifiers. The used Generate Blank Node Identifier algorithm keeps an *identifier map* and a *counter* to ensure consistent relabeling and avoid collisions. Thus, before this algorithm is run, the *identifier map* is reset and the *counter* is initialized to `0`.

1) Expand *element* according to the Expansion algorithm.

2) Generate a *node map* according to the Node Map Generation algorithm.

3) Initialize an empty RDF dataset *dataset*.

4) For each *graph name* and *graph* in *node map* ordered by *graph name*:

4.1) If *graph name* is a relative IRI, continue with the next *graph name*-*graph* pair.

4.2) Initialize *triples* as an empty array.

4.3) For each *subject* and *node* in *graph* ordered by *subject*:

4.3.1) If *subject* is a relative IRI, continue with the next *subject*-*node* pair.

4.3.2) For each *property* and *values* in *node* ordered by *property*:

4.3.2.1) If *property* is `@type`, then for each *type* in *values*, append a triple composed of *subject*, `rdf:type`, and *type* to *triples*.

4.3.2.2) Otherwise, if *property* is a keyword continue with the next *property-values* pair.

4.3.2.3) Otherwise, if *property* is a blank node identifier and the *produce generalized RDF* flag is not true, continue with the next *property-values* pair.

4.3.2.4) Otherwise, if *property* is a relative IRI, continue with the next *property-values* pair.

4.3.2.5) Otherwise, *property* is an absolute IRI or blank node identifier. For each *item* in *values*:

4.3.2.5.1) If *item* is a list object, initialize *list triples* as an empty array and *list head* to the result of the List Conversion algorithm, passing the value associated with the `@list` key from *item* and *list triples*. Append first a triple composed of *subject*, *property*, and *list head* to *triples* and finally append all triples from *list triples* to *triples*.

4.3.2.5.2) Otherwise, *item* is a value object or a node object. Append a triple composed of *subject*, *property*, and the result of using the Object to RDF Conversion algorithm passing *item* to *triples*, unless the result is null, indicating a relative IRI that has to be ignored.

4.4) If *graph name* is `@default`, add *triples* to the default graph in *dataset*.

4.5) Otherwise, create a named graph in *dataset* composed of *graph name* and add *triples*.

5) Return *dataset*.

10.2 Object to RDF Conversion

This algorithm takes a node object or value object and transforms it into an RDF resource to be used as the object of an RDF triple. If a node object containing a relative IRI is passed to the algorithm, null is returned which then causes the resulting RDF triple to be ignored.

Overview

This section is non-normative.

Value objects are transformed to RDF literals as described in section 10.6 Data Round Tripping whereas node objects are transformed to IRIs, blank node identifiers, or null.

Algorithm

The algorithm takes as its sole argument *item* which must be either a value object or node object.

1) If *item* is a node object and the value of its `@id` member is a relative IRI, return null.

2) If *item* is a node object, return the IRI or blank node identifier associated with its `@id` member.

3) Otherwise, *item* is a value object. Initialize *value* to the value associated with the `@value` member in *item*.

4) Initialize *datatype* to the value associated with the `@type` member of *item* or null if *item* does not have such a member.

5) If *value* is true or false, set *value* to the string `true` or `false` which is the canonical lexical form as described in section 10.6 Data Round Tripping If *datatype* is null, set it to `xsd:boolean`.

6) Otherwise, if *value* is a number with a non-zero fractional part (the result of a modulo-1 operation) or *value* is a number and *datatype* equals `xsd:double`, convert *value* to a string in canonical lexical form of an `xsd:double` as defined in [XMLSCHEMA11-2] and described in section 10.6 Data Round Tripping. If *datatype* is null, set it to `xsd:double`.

7) Otherwise, if *value* is a number with no non-zero fractional part (the result of a modulo-1 operation) or *value* is a number and *datatype* equals `xsd:integer`, convert *value* to a string in canonical lexical form of an `xsd:integer` as defined in [XMLSCHEMA11-2] and described in section 10.6 Data Round Tripping. If *datatype* is null, set it to `xsd:integer`.

8) Otherwise, if *datatype* is null, set it to `xsd:string` or `rdf:langString`, depending on if item has an `@language` member.

9) Initialize *literal* as an RDF literal using *value* and *datatype*. If *item* has an `@language` member, add the value associated with the `@language` key as the language tag of *literal*.

10) Return *literal*.

10.3 List to RDF Conversion

List Conversion is the process of taking a list object and transforming it into an RDF Collection as defined in RDF Semantics [RDF11-MT].

Overview

This section is non-normative.

For each element of the list a new blank node identifier is allocated which is used to generate `rdf:first` and `rdf:rest` triples. The algorithm returns the list head, which is either the first allocated blank node identifier or `rdf:nil` if the list is empty. If a list element represents a relative IRI, the corresponding `rdf:first` triple is omitted.

Algorithm

The algorithm takes two inputs: an array *list* and an empty array *list triples* used for returning the generated triples.

1) If *list* is empty, return `rdf:nil`.

2) Otherwise, create an array *bnodes* composed of a newly generated blank node identifier for each entry in *list*.

3) Initialize an empty array *list triples*.

4) For each pair of *subject* from *bnodes* and *item* from *list*:

> **4.1)** Initialize *object* to the result of using the Object to RDF Conversion algorithm passing *item* to *list triples*.
>
> **4.2)** Unless *object* is null, append a triple composed of *subject*, `rdf:first`, and *object*.
>
> **4.3)** Set *rest* as the next entry in *bnodes*, or if that does not exist, `rdf:nil`. Append a triple composed of *subject*, `rdf:rest`, and *rest* to *list triples*.

5) Return the first blank node from *bnodes* or `rdf:nil` if *bnodes* is empty.

10.4 Serialize RDF as JSON-LD Algorithm

This algorithm serializes an RDF dataset consisting of a default graph and zero or more named graphs into a JSON-LD document.

Overview

This section is non-normative.

Iterate through each graph in the dataset, converting each RDF Collection into a list and generating a JSON-LD document in expanded form for all RDF literals, IRIs and blank node identifiers. If the *use native types* flag is set to true, RDF literals with a datatype IRI that equals `xsd:integer` or `xsd:double` are converted to a JSON numbers and RDF literals with a datatype IRI that equals `xsd:boolean` are converted to true or false based on their lexical form as described in section 10.6 Data Round Tripping. Unless the *use `rdf:type`* flag is set to true, `rdf:type` predicates will be serialized as `@type` as long as the associated object is either an IRI or blank node identifier.

Algorithm

The algorithm takes one required and two optional inputs: an RDF dataset and the two flags *use native types* and *use `rdf:type`* that both default to false.

1) Initialize *default graph* to an empty JSON object.

2) Initialize *graph map* to a JSON object consisting of a single member `@default` whose value references *default graph*.

3) For each *graph* in RDF dataset:

> **3.1)** If *graph* is the default graph, set *name* to `@default`, otherwise to the graph name associated with *graph*.
>
> **3.2)** If *graph map* has no *name* member, create one and set its value to an empty JSON object.
>
> **3.3)** If *graph* is not the default graph and *default graph* does not have a *name* member, create such a member and initialize its value to a new JSON object with a single member `@id` whose value is *name*.
>
> **3.4)** Reference the value of the *name* member in *graph map* using the variable *node map*.
>
> **3.5)** For each RDF triple in *graph* consisting of *subject*, *predicate*, and *object*:
>
> > **3.5.1)** If *node map* does not have a *subject* member, create one and initialize its value to a new JSON object consisting of a single member `@id` whose value is set to *subject*.
> >
> > **3.5.2)** Reference the value of the *subject* member in *node map* using the variable *node*.
> >
> > **3.5.3)** If *object* is an IRI or blank node identifier, and *node map* does not have an *object* member, create one and initialize its value to a new JSON object consisting of a single member `@id` whose value is set to *object*.
> >
> > **3.5.4)** If *predicate* equals `rdf:type`, the *use `rdf:type`* flag is not true, and *object* is an IRI or blank node identifier, append *object* to the value of the `@type` member of *node*; unless such an item already exists. If no such member exists, create one and initialize it to an array whose only item is *object*. Finally, continue to the next RDF triple.
> >
> > **3.5.5)** Set *value* to the result of using the RDF to Object Conversion algorithm, passing *object* and *use native types*.
> >
> > **3.5.6)** If *node* does not have an *predicate* member, create one and initialize its value to an empty array.
> >
> > **3.5.7)** If there is no item equivalent to *value* in the array associated with the *predicate* member of *node*, append a reference to *value* to the array. Two JSON objects are considered equal if they have equivalent key-value pairs.
> >
> > **3.5.8)** If *object* is a blank node identifier or IRI, it might represent the list node:

3.5.8.1) If the *object* member of *node map* has no `usages` member, create one and initialize it to an empty array.

3.5.8.2) Reference the `usages` member of the *object* member of *node map* using the variable *usages*.

3.5.8.3) Append a new JSON object consisting of three members, `node`, `property`, and `value` to the *usages* array. The `node` member is set to a reference to *node*, `property` to *predicate*, and `value` to a reference to *value*.

4) For each *name* and *graph object* in *graph map*:

4.1) If *graph object* has no `rdf:nil` member, continue with the next *name*-*graph object* pair as the graph does not contain any lists that need to be converted.

4.2) Initialize *nil* to the value of the `rdf:nil` member of *graph object*.

4.3) For each item *usage* in the `usages` member of *nil*, perform the following steps:

4.3.1) Initialize *node* to the value of the value of the `node` member of *usage*, *property* to the value of the `property` member of *usage*, and *head* to the value of the `value` member of *usage*.

4.3.2) Initialize two empty arrays *list* and *list nodes*.

4.3.3) While *property* equals `rdf:rest`, the value associated to the `usages` member of *node* has exactly 1 entry, *node* has a `rdf:first` and `rdf:rest` property, both of which have as value an array consisting of a single element, and *node* has no other members apart from an optional `@type` member whose value is an array with a single item equal to `rdf:List`, *node* represents a well-formed list node. Perform the following steps to traverse the list backwards towards its head:

4.3.3.1) Append the only item of `rdf:first` member of *node* to the *list* array.

4.3.3.2) Append the value of the `@id` member of *node* to the *list nodes* array.

4.3.3.3) Initialize *node usage* to the only item of the `usages` member of *node*.

4.3.3.4) Set *node* to the value of the `node` member of *node usage*, *property* to the value of the `property` member of *node usage*, and *head* to the value of the `value` member of *node usage*.

4.3.3.5) If the `@id` member of *node* is an IRI instead of a blank node identifier, exit the while loop.

4.3.4) If *property* equals `rdf:first`, i.e., the detected list is nested inside another list

4.3.4.1) and the value of the `@id` of *node* equals `rdf:nil`, i.e., the detected list is empty, continue with the next *usage* item. The `rdf:nil` node cannot be converted to a list object as it would result in a list of lists, which isn't supported.

4.3.4.2) Otherwise, the list consists of at least one item. We preserve the head node and transform the rest of the linked list to a list object.

4.3.4.3) Set *head id* to the value of the `@id` member of *head*.

4.3.4.4) Set *head* to the value of the *head id* member of *graph object* so that all it's properties can be accessed.

4.3.4.5) Then, set *head* to the only item in the value of the `rdf:rest` member of *head*.

4.3.4.6) Finally, remove the last item of the *list* array and the last item of the *list nodes* array.

4.3.5) Remove the `@id` member from *head*.

4.3.6) Reverse the order of the *list* array.

4.3.7) Add an `@list` member to *head* and initialize its value to the *list* array.

4.3.8) For each item *node id* in *list nodes*, remove the *node id* member from *graph object*.

5) Initialize an empty array *result*.

6) For each *subject* and *node* in *default graph* ordered by *subject*:

6.1) If *graph map* has a *subject* member:

6.1.1) Add an `@graph` member to *node* and initialize its value to an empty array.

6.1.2) For each key-value pair *s*-*n* in the *subject* member of *graph map* ordered by *s*, append *n* to the `@graph` member of *node* after removing its `usages` member, unless the only remaining member of *n* is `@id`.

6.2) Append *node* to *result* after removing its `usages` member, unless the only remaining member of *node* is `@id`.

7) Return *result*.

10.5 RDF to Object Conversion

This algorithm transforms an RDF literal to a JSON-LD value object and a RDF blank node or IRI to an JSON-LD node object.

Overview

This section is non-normative.

RDF literals are transformed to value objects whereas IRIs and blank node identifiers are transformed to node objects. If the *use native types* flag is set to true, RDF literals with a datatype IRI that equals `xsd:integer` or `xsd:double` are converted to a JSON numbers and RDF literals with a datatype IRI that equals `xsd:boolean` are converted to true or false based on their lexical form as described in section 10.6 Data Round Tripping.

Algorithm

This algorithm takes two required inputs: a *value* to be converted to a JSON object and a flag *use native types*.

1) If *value* is an IRI or a blank node identifier, return a new JSON object consisting of a single member `@id` whose value is set to *value*.

2) Otherwise *value* is an RDF literal:

2.1) Initialize a new empty JSON object result.

2.2) Initialize *converted value* to *value*.

2.3) Initialize *type* to null

2.4) If *use native types* is true

2.4.1) If the datatype IRI of *value* equals `xsd:string`, set *converted value* to the lexical form of *value*.

2.4.2) Otherwise, if the datatype IRI of *value* equals `xsd:boolean`, set *converted value* to true if the lexical form of *value* matches `true`, or false if it matches `false`. If it matches neither, set *type* to `xsd:boolean`.

2.4.3) Otherwise, if the datatype IRI of *value* equals `xsd:integer` or `xsd:double` and its lexical form is a valid `xsd:integer` or `xsd:double` according [XMLSCHEMA11-2], set *converted value* to the result of converting the lexical form to a JSON number.

2.5) Otherwise, if *value* is a language-tagged string add a member `@language` to *result* and set its value to the language tag of *value*.

2.6) Otherwise, set *type* to the datatype IRI of *value*, unless it equals `xsd:string` which is ignored.

2.7) Add a member `@value` to *result* whose value is set to *converted value*.

2.8) If *type* is not null, add a member `@type` to *result* whose value is set to *type*.

2.9) Return *result*.

10.6 Data Round Tripping

When deserializing JSON-LD to RDF JSON-native numbers are automatically type-coerced to `xsd:integer` or `xsd:double` depending on whether the number has a non-zero fractional part or not (the result of a modulo-1 operation), the boolean values true and false are coerced to `xsd:boolean`, and strings are coerced to `xsd:string`. The numeric or boolean values themselves are converted to ***canonical lexical form***, i.e., a deterministic string representation as defined in [XMLSCHEMA11-2].

The canonical lexical form of an *integer*, i.e., a number with no non-zero fractional part or a number coerced to `xsd:integer`, is a finite-length sequence of decimal digits (`0-9`) with an optional leading minus sign; leading zeros are prohibited. In JavaScript, implementers can use the following snippet of code to convert an integer to canonical lexical form:

EXAMPLE 12: Sample integer serialization implementation in JavaScript

```
(value).toFixed(0).toString()
```

The canonical lexical form of a *double*, i.e., a number with a non-zero fractional part or a number coerced to `xsd:double`, consists of a mantissa followed by the character `E`, followed by an exponent. The mantissa is a decimal number and the exponent is an integer. Leading zeros and a preceding plus sign (`+`) are prohibited in the exponent. If the exponent is zero, it is indicated by `E0`. For the mantissa, the preceding optional plus sign is prohibited and the decimal point is required. Leading and trailing zeros are prohibited subject to the following: number representations must be normalized such that there is a single digit which is non-zero to the left of the decimal point and at least a single digit to the right of the decimal point unless the value being represented is zero. The canonical representation for zero is `0.0E0`. `xsd:double`'s value space is defined by the IEEE double-precision 64-bit floating point type [IEEE-754-2008] whereas the value space of JSON numbers is not specified; when deserializing JSON-LD to RDF the mantissa is rounded to 15 digits after the decimal point. In JavaScript, implementers can use the following snippet of code to convert a double to canonical lexical form:

EXAMPLE 13: Sample floating point number serialization implementation in JavaScript

```
(value).toExponential(15).replace(/(\d)0*e\+?/,'$1E')
```

The canonical lexical form of the *boolean* values true and false are the strings `true` and `false`.

When JSON-native numbers are deserialized to RDF, lossless data round-tripping cannot be guaranteed, as rounding errors might occur. When serializing RDF as JSON-LD, similar rounding errors might occur. Furthermore, the datatype or the lexical representation might be lost. An `xsd:double` with a value of `2.0` will, e.g., result in an `xsd:integer` with a value of `2` in canonical lexical form when converted from RDF to JSON-LD and back to RDF. It is important to highlight that in practice it might be impossible to losslessly convert an `xsd:integer` to a number because its value space is not limited. While the JSON specification [RFC4627] does not limit the value space of numbers either, concrete implementations typically do have a limited value space.

To ensure lossless round-tripping the Serialize RDF as JSON-LD algorithm specifies a *use native types* flag which controls whether RDF literals with a datatype IRI equal to `xsd:integer`, `xsd:double`, or `xsd:boolean` are converted to their JSON-native counterparts. If the *use native types* flag is set to false, all literals remain in their original string representation.

Some JSON serializers, such as PHP's native implementation in some versions, backslash-escape the forward slash character. For example, the value `http://example.com/` would be serialized as `http:\/\/example.com\/`. This is problematic as other JSON parsers might not understand those escaping characters. There is no need to backslash-escape forward slashes in JSON-LD. To aid interoperability between JSON-LD processors, forward slashes MUST NOT be backslash-escaped.

11. The Application Programming Interface

This section is non-normative.

This API provides a clean mechanism that enables developers to convert JSON-LD data into a variety of output formats that are often easier to work with.

The JSON-LD API uses Promises to represent the result of the various asynchronous operations. ***Promises*** are temporarily being drafted on *GitHub* [PROMISES] but are expected to be standardized as part of ECMAScript 6.

11.1 The `JsonLdProcessor` Interface

This section is non-normative.

The `JsonLdProcessor` interface is the high-level programming structure that developers use to access the JSON-LD transformation methods.

It is important to highlight that implementations do not modify the input parameters. If an error is detected, the Promise is rejected passing a `JsonLdError` with the corresponding error `code` and processing is stopped.

If the `documentLoader` option is specified, it is used to dereference remote documents and contexts. The `documentUrl` in the returned `RemoteDocument` is used as base IRI and the `contextUrl` is used instead of looking at the HTTP Link Header directly. For the sake of simplicity, none of the algorithms in this document mention this directly.

WebIDL

```
[Constructor]
interface JsonLdProcessor {
    Promise compact (any input, JsonLdContext context, optional JsonLdOptions options);
    Promise expand (any input, optional JsonLdOptions options);
    Promise flatten (any input, optional JsonLdContext? context, optional JsonLdOptions options);
};
```

Methods

This section is non-normative.

`compact`

Compacts the given *input* using the *context* according to the steps in the Compaction algorithm:

1) Create a new Promise *promise* and return it. The following steps are then executed asynchronously.

2) If the passed *input* is a DOMString representing the IRI of a remote document, dereference it. If the retrieved document's content type is neither `application/json`, nor `application/ld+json`, nor any other media type using a `+json` suffix as defined in [RFC6839] or if the document cannot be parsed as JSON, reject the *promise* passing an `loading document failed` error.

3) Initialize a new empty active context. The base IRI of the active context is set to the IRI of the currently being processed document, if available; otherwise to null. If set, the `base` option overrides the base IRI.

4) If an `expandContext` has been passed, update the active context using the Context Processing algorithm, passing the `expandContext` as local context. If `expandContext` is a JSON object having an `@context` member, pass that member's value instead.

5) If the *input* has been retrieved, the response has an HTTP Link Header [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation and a content type of `application/json` or any media type with a `+json` suffix as defined in [RFC6839] except `application/ld+json`, update the active context using the Context Processing algorithm, passing the context referenced in the HTTP Link Header as local context. The HTTP Link Header is ignored for documents served as `application/ld+json` If multiple HTTP Link Headers using the `http://www.w3.org/ns/json-ld#context` link relation are found, the *promise* is rejected with a `JsonLdError` whose code is set to `multiple context link headers` and processing is terminated.

6) Set *expanded* to the result of using the Expansion algorithm, passing the active context and *input* as *element*.

7) If *context* is a JSON object having an `@context` member, set *context* to that member's value.

8) Set *compacted* to the result of using the Compaction algorithm, passing *context*, *expanded* as *element*, and if passed, the `compactArrays` flag in *options*.

9) Fulfill the *promise* passing *compacted*.

Parameter	Type	Nullable	Optional	Description
input	`any`	✘	✘	The JSON-LD object or array of JSON-LD objects to perform the compaction upon or an IRI referencing the JSON-LD document to compact.
context	`JsonLdContext`	✘	✘	The context to use when compacting the `input`; it can be specified by using a JSON object, an IRI, or an array consisting of JSON objects and IRIs.
options	`JsonLdOptions`	✘	✔	A set of options to configure the algorithms. This allows, e.g., to set the input document's base IRI.

Return type: `Promise`

`expand`

Expands the given *input* according to the steps in the Expansion algorithm:

1) Create a new Promise *promise* and return it. The following steps are then executed asynchronously.

2) If the passed *input* is a DOMString representing the IRI of a remote document, dereference it. If the retrieved document's content type is neither `application/json`, nor `application/ld+json`, nor any other media type using a `+json` suffix as defined in [RFC6839], reject the *promise* passing an `loading document failed` error.

3) Initialize a new empty active context. The base IRI of the active context is set to the IRI of the currently being processed document, if available; otherwise to null. If set, the `base` option overrides the base IRI.

4) If an `expandContext` has been passed, update the active context using the Context Processing algorithm, passing the `expandContext` as local context. If `expandContext` is a JSON object having an `@context` member, pass that member's value instead.

5) If the *input* has been retrieved, the response has an HTTP Link Header [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation and a content type of `application/json` or any media type with a `+json` suffix as defined in [RFC6839] except `application/ld+json`, update the active context using the Context Processing algorithm, passing the context referenced in the HTTP Link Header as local context. The HTTP Link Header is ignored for documents served as `application/ld+json` If multiple HTTP Link Headers using the `http://www.w3.org/ns/json-ld#context` link relation are found, the *promise* is rejected with a `JsonLdError` whose code is set to `multiple context link headers` and processing is terminated.

6) Set *expanded* to the result of using the Expansion algorithm, passing the active context and *input* as *element*.

7) Fulfill the *promise* passing *expanded*.

Parameter	Type	Nullable	Optional	Description
input	`any`	✘	✘	The JSON-LD object or array of JSON-LD objects to perform the expansion upon or an IRI referencing the JSON-LD document to expand.
options	`JsonLdOptions`	✘	✔	A set of options to configure the used algorithms such. This allows, e.g., to set the input document's base IRI.

Return type: `Promise`

`flatten`

Flattens the given *input* and compacts it using the passed *context* according to the steps in the Flattening algorithm:

1) Create a new Promise *promise* and return it. The following steps are then executed asynchronously.

2) If the passed *input* is a DOMString representing the IRI of a remote document, dereference it. If the retrieved document's content type is neither `application/json`, nor `application/ld+json`, nor any other media type using a `+json` suffix as defined in [RFC6839], reject the *promise* passing an `loading document failed` error.

3) Initialize a new empty active context. The base IRI of the active context is set to the IRI of the currently being processed document, if available; otherwise to null. If set, the `base` option overrides the base IRI.

4) If an `expandContext` has been passed, update the active context using the Context Processing algorithm, passing the `expandContext` as local context. If `expandContext` is a JSON object having an `@context` member, pass that member's value instead.

5) If the *input* has been retrieved, the response has an HTTP Link Header [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation and a content type of `application/json` or any media type with a `+json` suffix as defined in [RFC6839] except `application/ld+json`, update the active context using the Context Processing algorithm, passing the context referenced in the HTTP Link Header as local context. The HTTP Link Header is ignored for documents served as `application/ld+json` If multiple HTTP Link Headers using the `http://www.w3.org/ns/json-ld#context` link relation are found, the *promise* is rejected with a `JsonLdError` whose code is set to `multiple context link headers` and processing is terminated.

6) Set *expanded* to the result of using the Expansion algorithm, passing the active context and *input* as *element*.

7) If *context* is a JSON object having an `@context` member, set *context* to that member's value.

8) Initialize an empty *identifier map* and a *counter* (set to `0`) to be used by the Generate Blank Node Identifier algorithm.

9) Set *flattened* to the result of using the Flattening algorithm, passing *expanded* as *element*, *context*, and if passed, the `compactArrays` flag in *options* (which is internally passed to the Compaction algorithm).

10) Fulfill the *promise* passing *flattened*.

Parameter	Type	Nullable	Optional	Description
input	`any`	✘	✘	The JSON-LD object or array of JSON-LD objects or an IRI referencing the JSON-LD document to flatten.
context	`JsonLdContext`	✔	✔	The context to use when compacting the flattened `input`; it can be specified by using a JSON object, an IRI, or an array consisting of JSON objects and IRIs. If not passed or null is passed, the result will not be compacted but kept in expanded form.
options	`JsonLdOptions`	✘	✔	A set of options to configure the used algorithms such. This allows, e.g., to set the input document's base IRI.

Return type: `Promise`

WebIDL

```
typedef (object or DOMString or (object or DOMString[])) JsonLdContext;
```

The `JsonLdContext` type is used to refer to a value that that may be a JSON object, a string representing an IRI, or an array of JSON objects and strings.

11.2 The `JsonLdOptions` Type

This section is non-normative.

The `JsonLdOptions` type is used to pass various options to the `JsonLdProcessor` methods.

WebIDL

```
dictionary JsonLdOptions {
    DOMString?             base;
    boolean                compactArrays = true;
    LoadDocumentCallback   documentLoader = null;
    (object? or DOMString) expandContext = null;
    DOMString              processingMode = "json-ld-1.0";
};
```

Dictionary `JsonLdOptions` Members

This section is non-normative.

`base` of type DOMString, nullable
: The base IRI to use when expanding or compacting the document. If set, this overrides the input document's IRI.

`compactArrays` of type boolean, defaulting to `true`
: If set to `true`, the JSON-LD processor replaces arrays with just one element with that element during compaction. If set to `false`, all arrays will remain arrays even if they have just one element.

`documentLoader` of type *LoadDocumentCallback*, defaulting to `null`
: The callback of the loader to be used to retrieve remote documents and contexts. If specified, it is used to retrieve remote documents and contexts; otherwise, if not specified, the processor's built-in loader is used.

`expandContext` of type (object? or DOMString), defaulting to `null`
: A context that is used to initialize the active context when expanding a document.

`processingMode` of type DOMString, defaulting to `"json-ld-1.0"`
: If set to `json-ld-1.0`, the implementation has to produce exactly the same results as the algorithms defined in this specification. If set to another value, the JSON-LD processor is allowed to extend or modify the algorithms defined in this specification to enable application-specific optimizations. The definition of such optimizations is beyond the scope of this specification and thus not defined. Consequently, different implementations may implement different optimizations. Developers must not define modes beginning with `json-ld` as they are reserved for future versions of this specification.

11.3 Remote Document and Context Retrieval

This section is non-normative.

Users of an API implementation can utilize a callback to control how remote documents and contexts are retrieved. This section details the parameters of that callback and the data structure used to return the retrieved context.

LoadDocumentCallback

This section is non-normative.

The `LoadDocumentCallback` defines a callback that custom document loaders have to implement to be used to retrieve remote documents and contexts.

WebIDL

```
callback LoadDocumentCallback = Promise (DOMString url);
```

Callback `LoadDocumentCallback` Parameters

This section is non-normative.

`url` of type DOMString
: The URL of the remote document or context to load.

All errors result in the Promise being rejected with a `JsonLdError` whose code is set to `loading document failed` or `multiple context link headers` as described in the next section.

RemoteDocument

This section is non-normative.

The `RemoteDocument` type is used by a `LoadDocumentCallback` to return information about a remote document or context.

WebIDL

```
dictionary RemoteDocument {
    DOMString contextUrl = null;
    DOMString documentUrl;
    any       document;
};
```

Dictionary `RemoteDocument` Members

This section is non-normative.

`contextUrl` of type DOMString, defaulting to `null`
: If available, the value of the HTTP Link Header [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation in the response. If the response's content type is `application/ld+json`, the HTTP Link Header is ignored. If multiple HTTP Link Headers using the `http://www.w3.org/ns/json-ld#context` link relation are found, the Promise of the `LoadDocumentCallback` is rejected with a `JsonLdError` whose code is set to `multiple context link headers`.

`document` of type any
: The retrieved document. This can either be the raw payload or the already parsed document.

`documentUrl` of type DOMString
: The final URL of the loaded document. This is important to handle HTTP redirects properly.

11.4 Error Handling

This section is non-normative.

This section describes the datatype definitions used within the JSON-LD API for error handling.

JsonLdError

This section is non-normative.

The `JsonLdError` type is used to report processing errors.

WebIDL

```
dictionary JsonLdError {
    JsonLdErrorCode code;
    DOMString?      message = null;
};
```

Dictionary `JsonLdError` Members

This section is non-normative.

`code` of type *JsonLdErrorCode*
: a string representing the particular error type, as described in the various algorithms in this document.

`message` of type DOMString, nullable, defaulting to **`null`**
: an optional error message containing additional debugging information. The specific contents of error messages are outside the scope of this specification.

JsonLdErrorCode

This section is non-normative.

The `JsonLdErrorCode` represents the collection of valid JSON-LD error codes.

WebIDL

```
enum JsonLdErrorCode {
    "loading document failed",
    "list of lists",
    "invalid @index value",
    "conflicting indexes",
    "invalid @id value",
    "invalid local context",
    "multiple context link headers",
    "loading remote context failed",
    "invalid remote context",
    "recursive context inclusion",
    "invalid base IRI",
    "invalid vocab mapping",
    "invalid default language",
    "keyword redefinition",
    "invalid term definition",
    "invalid reverse property",
    "invalid IRI mapping",
    "cyclic IRI mapping",
    "invalid keyword alias",
    "invalid type mapping",
    "invalid language mapping",
    "colliding keywords",
    "invalid container mapping",
    "invalid type value",
    "invalid value object",
    "invalid value object value",
    "invalid language-tagged string",
    "invalid language-tagged value",
    "invalid typed value",
    "invalid set or list object",
    "invalid language map value",
    "compaction to list of lists",
    "invalid reverse property map",
    "invalid @reverse value",
    "invalid reverse property value"
};
```

Enumeration description	
`loading document failed`	The document could not be loaded or parsed as JSON.
`list of lists`	A list of lists was detected. List of lists are not supported in this version of JSON-LD due to the algorithmic complexity.
`invalid @index value`	An `@index` member was encountered whose value was not a string.
`conflicting indexes`	Multiple conflicting indexes have been found for the same node.
`invalid @id value`	An `@id` member was encountered whose value was not a string.
`invalid local context`	In invalid local context was detected.
`multiple context link headers`	Multiple HTTP Link Headers [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation have been detected.
`loading remote context failed`	There was a problem encountered loading a remote context.

invalid remote context	No valid context document has been found for a referenced, remote context.
recursive context inclusion	A cycle in remote context inclusions has been detected.
invalid base IRI	An invalid base IRI has been detected, i.e., it is neither an absolute IRI nor null.
invalid vocab mapping	An invalid vocabulary mapping has been detected, i.e., it is neither an absolute IRI nor null.
invalid default language	The value of the default language is not a string or null and thus invalid.
keyword redefinition	A keyword redefinition has been detected.
invalid term definition	An invalid term definition has been detected.
invalid reverse property	An invalid reverse property definition has been detected.
invalid IRI mapping	A local context contains a term that has an invalid or missing IRI mapping.
cyclic IRI mapping	A cycle in IRI mappings has been detected.
invalid keyword alias	An invalid keyword alias definition has been encountered.
invalid type mapping	An `@type` member in a term definition was encountered whose value could not be expanded to an absolute IRI.
invalid language mapping	An `@language` member in a term definition was encountered whose value was neither a string nor null and thus invalid.
colliding keywords	Two properties which expand to the same keyword have been detected. This might occur if a keyword and an alias thereof are used at the same time.
invalid container mapping	An `@container` member was encountered whose value was not one of the following strings: `@list`, `@set`, or `@index`.
invalid type value	An invalid value for an `@type` member has been detected, i.e., the value was neither a string nor an array of strings.
invalid value object	A value object with disallowed members has been detected.
invalid value object value	An invalid value for the `@value` member of a value object has been detected, i.e., it is neither a scalar nor null.
invalid language-tagged string	A language-tagged string with an invalid language value was detected.
invalid language-tagged value	A number, true, or false with an associated language tag was detected.
invalid typed value	A typed value with an invalid type was detected.
invalid set or list object	A set object or list object with disallowed members has been detected.
invalid language map value	An invalid value in a language map has been detected. It has to be a string or an array of strings.
compaction to list of lists	The compacted document contains a list of lists as multiple lists have been compacted to the same term.
invalid reverse property map	An invalid reverse property map has been detected. No keywords apart from `@context` are allowed in reverse property maps.
invalid @reverse value	An invalid value for an `@reverse` member has been detected, i.e., the value was not a JSON object.
invalid reverse property value	An invalid value for a reverse property has been detected. The value of an inverse property must be a node object.

A. Acknowledgements

This section is non-normative.

A large amount of thanks goes out to the JSON-LD Community Group participants who worked through many of the technical issues on the mailing list and the weekly telecons - of special mention are Niklas Lindström, François Daoust, Lin Clark, and Zdenko 'Denny' Vrandečić. The editors would like to thank Mark Birbeck, who provided a great deal of the initial push behind the JSON-LD work via his work on RDFj. The work of Dave Lehn and Mike Johnson are appreciated for reviewing, and performing several implementations of the specification. Ian Davis is thanked for his work on RDF/JSON. Thanks also to Nathan Rixham, Bradley P. Allen, Kingsley Idehen, Glenn McDonald, Alexandre Passant, Danny Ayers, Ted Thibodeau Jr., Olivier Grisel, Josh Mandel, Eric Prud'hommeaux, David Wood, Guus Schreiber, Pat Hayes, Sandro Hawke, and Richard Cyganiak for their input on the specification.

B. References

B.1 Normative references

[IEEE-754-2008]
IEEE 754-2008 Standard for Floating-Point Arithmetic. URL: http://standards.ieee.org/findstds/standard/754-2008.html

[JSON-LD]
Manu Sporny, Gregg Kellogg, Markus Lanthaler, Editors. *JSON-LD 1.0*. 16 January 2014. W3C Recommendation. URL: http://www.w3.org/TR/json-ld/

[RDF11-MT]
Patrick J. Hayes, Peter F. Patel-Schneider, Editors. *RDF 1.1 Semantics*. 9 January 2014. W3C Proposed Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PR-rdf11-mt-20140109/. The latest edition is available at http://www.w3.org/TR/rdf11-mt/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt

[RFC3986]
T. Berners-Lee; R. Fielding; L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax (RFC 3986)*. January 2005. RFC. URL: http://www.ietf.org/rfc/rfc3986.txt

[RFC3987]
M. Dürst; M. Suignard. *Internationalized Resource Identifiers (IRIs)*. January 2005. RFC. URL: http://www.ietf.org/rfc/rfc3987.txt

[RFC4627]
D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON) (RFC 4627)*. July 2006. RFC. URL: http://www.ietf.org/rfc/rfc4627.txt

[RFC5988]
M. Nottingham. *Web Linking*. October 2010. Internet RFC 5988. URL: http://www.ietf.org/rfc/rfc5988.txt

[XMLSCHEMA11-2]
David Peterson; Sandy Gao; Ashok Malhotra; Michael Sperberg-McQueen; Henry Thompson; Paul V. Biron et al. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. 5 April 2012. W3C Recommendation. URL: http://www.w3.org/TR/xmlschema11-2/

B.2 Informative references

[BCP47]
A. Phillips; M. Davis. *Tags for Identifying Languages*. September 2009. IETF Best Current Practice. URL: http://tools.ietf.org/html/bcp47

[ECMA-262]
ECMAScript Language Specification, Edition 5.1. June 2011. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[JSON-LD-TESTS]
JSON-LD 1.0 Test Suite. W3C Test Suite. URL: http://www.w3.org/2013/json-ld-tests/

[PROMISES]
Domenic Denicola. *Promise Objects*. January 2014 (work in progress). URL: http://www.w3.org/2013/10/json-ld-api/snapshot-promises-draft. The latest draft is available at https://github.com/domenic/promises-unwrapping

[RDF11-CONCEPTS]
Richard Cyganiak, David Wood, Markus Lanthaler, Editors. *RDF 1.1 Concepts and Abstract Syntax*. 9 January 2014. W3C Proposed Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PR-rdf11-concepts-20140109/. The latest edition is available at http://www.w3.org/TR/rdf11-concepts/

[RFC6839]
Tony Hansen, Alexey Melnikov. *Additional Media Type Structured Syntax Suffixes*. January 2013. Internet RFC 6839. URL: http://www.ietf.org/rfc/rfc6839.txt

[TURTLE]
Eric Prud'hommeaux, Gavin Carothers, Editors. *RDF 1.1 Turtle: Terse RDF Triple Language*. 9 January 2014. W3C Proposed Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PR-turtle-20140109/. The latest edition is available at http://www.w3.org/TR/turtle/

[WEBIDL]
Cameron McCormack, Editor. *Web IDL*. 19 April 2012. W3C Candidate Recommendation (work in progress). URL: http://www.w3.org/TR/2012/CR-WebIDL-20120419/. The latest edition is available at http://www.w3.org/TR/WebIDL/