JSON-LD 1.1

A JSON-based Serialization for Linked Data

W3C Candidate Recommendation 12 December 2019

This version:
https://www.w3.org/TR/2019/CR-json-ld11-20191212/

Latest published version:
https://www.w3.org/TR/json-ld11/

Latest editor's draft:
https://w3c.github.io/json-ld-syntax/

Test suite:
https://w3c.github.io/json-ld-api/tests/

Implementation report:
https://w3c.github.io/json-ld-api/reports/

Previous version:
https://www.w3.org/TR/2019/WD-json-ld11-20191112/

Latest Recommendation:
https://www.w3.org/TR/2014/REC-json-ld-20140116/

Editors:
Gregg Kellogg (v1.0 and v1.1)
Pierre-Antoine Champin (LIRIS - Université de Lyon) (v1.1)
Dave Longley (Digital Bazaar) (v1.1)

Former editors:
Manu Sporny (Digital Bazaar) (v1.0)
Markus Lanthaler (Graz University of Technology) (v1.0)

Authors:
Manu Sporny (Digital Bazaar) (v1.0)
Dave Longley (Digital Bazaar) (v1.0 and v1.1)
Gregg Kellogg (v1.0 and v1.1)
Markus Lanthaler (Graz University of Technology) (v1.0)
Pierre-Antoine Champin (LIRIS - Université de Lyon) (v1.1)
Niklas Lindström (v1.0)

Participate:
GitHub w3c/json-ld-syntax
File a bug
Commit history

Pull requests

Copyright © 2010-2019 W3C® (MIT, ERCIM, Keio, Beihang). W3C liability, trademark and permissive document license rules apply.

Abstract

JSON is a useful data serialization and messaging format. This specification defines JSON-LD 1.1, a JSON-based format to serialize Linked Data. The syntax is designed to easily integrate into deployed systems that already use JSON, and provides a smooth upgrade path from JSON to JSON-LD. It is primarily intended to be a way to use Linked Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines.

This specification describes a superset of the features defined in *JSON-LD 1.0* [JSON-LD10] and, except where noted, documents created using the 1.0 version of this specification remain compatible with JSON-LD 1.1.

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 https://www.w3.org/TR/.

This document has been developed by the JSON-LD Working Group and was derived from the JSON-LD Community Group's Final Report.

There is a live JSON-LD playground that is capable of demonstrating the features described in this document.

This document was published by the JSON-LD Working Group as a Candidate Recommendation. This document is intended to become a W3C Recommendation.

GitHub Issues are preferred for discussion of this specification. Alternatively, you can send comments to our mailing list. Please send them to public-json-ld-wg@w3.org (archives).

W3C publishes a Candidate Recommendation to indicate that the document is believed to be stable and to encourage implementation by the developer

community. This Candidate Recommendation is expected to advance to Proposed Recommendation no earlier than 17 February 2020.

Please see the Working Group's implementation report.

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 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.

This document is governed by the 1 March 2019 W3C Process Document.

§ Set of Documents

This document is one of three JSON-LD 1.1 Recommendations produced by the JSON-LD Working Group:

- JSON-LD 1.1
- *JSON-LD 1.1 Processing Algorithms and API*
- *JSON-LD 1.1 Framing*

Table of Contents

1. Introduction
1.1 How to Read this Document
1.2 Contributing
1.3 Typographical conventions
1.4 Terminology
1.5 Design Goals and Rationale
1.6 Data Model Overview
1.7 Syntax Tokens and Keywords

2. Conformance

3. Basic Concepts
3.1 The Context
3.2 IRIs
3.3 Node Identifiers
3.4 Uses of JSON Objects
3.5 Specifying the Type

4. Advanced Concepts
4.1 Advanced Context Usage
4.1.1 JSON-LD 1.1 Processing Mode
4.1.2 Default Vocabulary
4.1.3 Base IRI
4.1.4 Using the Document Base for the Default Vocabulary
4.1.5 Compact IRIs
4.1.6 Aliasing Keywords
4.1.7 IRI Expansion within a Context
4.1.8 Scoped Contexts
4.1.9 Context Propagation
4.1.10 Imported Contexts
4.1.11 Protected Term Definitions
4.2 Describing Values
4.2.1 Typed Values
4.2.2 JSON Literals
4.2.3 Type Coercion
4.2.4 String Internationalization
4.2.4.1 Base Direction
4.3 Value Ordering
4.3.1 Lists
4.3.2 Sets
4.3.3 Using `@set` with `@type`
4.4 Nested Properties
4.5 Embedding
4.5.1 Identifying Blank Nodes
4.6 Indexed Values
4.6.1 Data Indexing
4.6.1.1 Property-based data indexing
4.6.2 Language Indexing
4.6.3 Node Identifier Indexing
4.6.4 Node Type Indexing
4.7 Included Nodes

4.8 Reverse Properties
4.9 Named Graphs
4.9.1 Graph Containers
4.9.2 Named Graph Data Indexing
4.9.3 Named Graph Indexing
4.10 Loading Documents

5. Forms of JSON-LD
5.1 Expanded Document Form
5.2 Compacted Document Form
5.2.1 Shortening IRIs
5.2.2 Representing Values as Strings
5.2.3 Representing Lists as Arrays
5.2.4 Reversing Node Relationships
5.2.5 Indexing Values
5.2.6 Normalizing Values as Objects
5.2.7 Representing Singular Values as Arrays
5.2.8 Term Selection
5.3 Flattened Document Form
5.4 Framed Document Form

6. Modifying Behavior with Link Relationships
6.1 Interpreting JSON as JSON-LD
6.2 Alternate Document Location

7. Embedding JSON-LD in HTML Documents
7.1 Inheriting base IRI from HTML's `base` element
7.2 Restrictions for contents of JSON-LD `script` elements
7.3 Locating a Specific JSON-LD Script Element

8. Data Model

9. JSON-LD Grammar
9.1 Terms
9.2 Node Objects
9.3 Frame Objects
9.4 Graph Objects
9.5 Value Objects
9.6 Value Patterns
9.7 Lists and Sets
9.8 Language Maps

9.9 Index Maps
9.10 Property-based Index Maps
9.11 Id Maps
9.12 Type Maps
9.13 Included Blocks
9.14 Property Nesting
9.15 Context Definitions
9.15.1 Expanded term definition
9.16 Keywords

10. Relationship to RDF
10.1 Serializing/Deserializing RDF
10.2 The `rdf:JSON` Datatype
10.3 The `i18n` Namespace
10.4 The `rdf:CompoundLiteral` class and the `rdf:language` and `rdf:direction` properties

11. Security Considerations

12. Privacy Considerations

13. Internationalization Considerations

A. Image Descriptions
A.1 Linked Data Dataset

B. Relationship to Other Linked Data Formats
B.1 Turtle
B.1.1 Prefix definitions
B.1.2 Embedding
B.1.3 Conversion of native data types
B.1.4 Lists
B.2 RDFa
B.3 Microdata

C. IANA Considerations
C.1 Examples

D. Open Issues

E. Changes since 1.0 Recommendation of 16 January 2014

F. Changes since JSON-LD Community Group Final Report

G. Acknowledgements

H. References
H.1 Normative references
H.2 Informative references

§ 1. Introduction

This section is non-normative.

Linked Data [LINKED-DATA] is a way to create a network of standards-based machine interpretable data across different documents and Web sites. It allows an application to start at one piece of Linked Data, and follow embedded links to other pieces of Linked Data that are hosted on different sites across the Web.

JSON-LD is a lightweight syntax to serialize Linked Data in JSON [RFC8259]. Its design allows existing JSON to be interpreted as Linked Data with minimal changes. JSON-LD is primarily intended to be a way to use Linked Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines. Since JSON-LD is 100% compatible with JSON, the large number of JSON parsers and libraries available today can be reused. In addition to all the features JSON provides, JSON-LD introduces:

- a universal identifier mechanism for JSON objects via the use of IRIs,
- a way to disambiguate keys shared among different JSON documents by mapping them to IRIs via a context,
- a mechanism in which a value in a JSON object may refer to a resource on a different site on the Web,
- the ability to annotate strings with their language,
- a way to associate datatypes with values such as dates and times,
- and a facility to express one or more directed graphs, such as a social network, in a single document.

JSON-LD is designed to be usable directly as JSON, with no knowledge of RDF [RDF11-CONCEPTS]. It is also designed to be usable as RDF in conjunction with other Linked Data technologies like SPARQL [SPARQL11-OVERVIEW]. Developers who require any of the facilities listed above or need

to serialize an RDF graph or Dataset in a JSON-based syntax will find JSON-LD of interest. People intending to use JSON-LD with RDF tools will find it can be used as another RDF syntax, as with [Turtle] and [TriG]. Complete details of how JSON-LD relates to RDF are in section § 10. Relationship to RDF.

The syntax is designed to not disturb already deployed systems running on JSON, but provide a smooth upgrade path from JSON to JSON-LD. Since the shape of such data varies wildly, JSON-LD features mechanisms to reshape documents into a deterministic structure which simplifies their processing.

§ 1.1 How to Read this Document

This section is non-normative.

This document is a detailed specification for a serialization of Linked Data in JSON. The document is primarily intended for the following audiences:

- Software developers who want to encode Linked Data in a variety of programming languages that can use JSON
- Software developers who want to convert existing JSON to JSON-LD
- Software developers who want to understand the design decisions and language syntax for JSON-LD
- Software developers who want to implement processors and APIs for JSON-LD
- Software developers who want to generate or consume Linked Data, an RDF graph, or an RDF Dataset in a JSON syntax

A companion document, the JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API], specifies how to work with JSON-LD at a higher level by providing a standard library interface for common JSON-LD operations.

To understand the basics in this specification you must first be familiar with *JSON*, which is detailed in [RFC8259].

This document almost exclusively uses the term IRI (Internationalized Resource Indicator) when discussing hyperlinks. Many Web developers are more familiar with the URL (Uniform Resource Locator) terminology. The document also uses, albeit rarely, the URI (Uniform Resource Indicator) terminology. While these terms are often used interchangeably among

technical communities, they do have important distinctions from one another and the specification goes to great lengths to try and use the proper terminology at all times.

This document can highlight changes since the *JSON-LD 1.0* version. Select to highlight changes.

§ 1.2 Contributing

This section is non-normative.

There are a number of ways that one may participate in the development of this specification:

- Technical discussion typically occurs on the working group mailing list: public-json-ld-wg@w3.org
- The working group uses #json-ld IRC channel is available for real-time discussion on irc.w3.org.
- The #json-ld IRC channel is also available for real-time discussion on irc.freenode.net.

§ 1.3 Typographical conventions

This section is non-normative.

The following typographic conventions are used in this specification:

`markup`
: Markup (elements, attributes, properties), machine processable values (string, characters, media types), property name, or a file name is in red-orange monospace font.

variable
: A variable in pseudo-code or in an algorithm description is in italics.

definition
: A definition of a term, to be used elsewhere in this or other specifications, is in bold and italics.

definition reference
: A reference to a definition *in this document* is underlined and is also an active link to the definition itself.

`markup definition reference`
: A references to a definition *in this document*, when the reference itself is

also a markup, is underlined, red-orange monospace font, and is also an active link to the definition itself.

external definition reference
A reference to a definition *in another document* is underlined, in italics, and is also an active link to the definition itself.

`markup external definition reference`
A reference to a definition *in another document*, when the reference itself is also a markup, is underlined, in italics red-orange monospace font, and is also an active link to the definition itself.

hyperlink
A hyperlink is underlined and in blue.

[reference]
A document reference (normative or informative) is enclosed in square brackets and links to the references section.

Changes from Recommendation
Sections or phrases changed from the previous Recommendation may be highlighted using a control in § 1.1 How to Read this Document.

NOTE

Notes are in light green boxes with a green left border and with a "Note" header in green. Notes are always informative.

EXAMPLE 1

```
Examples are in light khaki boxes, with khaki left border,
and with a numbered "Example" header in khaki.
Examples are always informative. The content of the example is in mor

Examples may have tabbed navigation buttons
to show the results of transforming an example into other representat
```

§ 1.4 Terminology

This section is non-normative.

This document uses the following terms as defined in external specifications and defines terms specific to JSON-LD.

§ Terms imported from Other Specifications

Terms imported from *ECMAScript Language Specification* [ECMASCRIPT], *The JavaScript Object Notation (JSON) Data Interchange Format* [RFC8259], *Infra Standard* [INFRA], and *Web IDL* [WEBIDL]

array

In the JSON serialization, an array structure is represented as square brackets surrounding zero or more values. Values are separated by commas. In the internal representation, a list (also called an array) is an *ordered* collection 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 the Sets and Lists section of JSON-LD 1.1.

boolean

The values `true` and `false` that are used to express one of two possible states.

JSON object

In the JSON serialization, an object structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members). A name is a string. A single colon comes after each name, separating the name from the value. A single comma separates a value from a following name. In JSON-LD the names in an object must be unique.

In the internal representation a JSON object is described as a ***map*** (see [INFRA]), composed of ***entries*** with key/value pairs.

In the Application Programming Interface, a map is described using a [WEBIDL] dictionary.

null

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

number

In the JSON serialization, 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. In the internal

representation, a number is equivalent to either a ***long*** or ***double***, depending on if the number has a non-zero fractional part (see [WEBIDL]).

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

string
A string is a sequence of zero or more Unicode (UTF-8) characters, wrapped in double quotes, using backslash escapes (if necessary). A character is represented as a single character string.

Terms imported from *Internationalized Resource Identifiers (IRIs)* [RFC3987]

IRI
The absolute form of an IRI containing a *scheme* along with a *path* and optional *query* and *fragment* segments.

IRI reference
Denotes the common usage of an Internationalized Resource Identifier. An IRI reference may be absolute or relative. However, the "IRI" that results from such a reference only includes absolute IRIs; any relative IRI references are resolved to their absolute form.

relative IRI reference
A relative IRI reference is an IRI reference that is relative to some other IRI, typically the base IRI of the document. Note that properties, values of `@type`, and values of terms defined to be *vocabulary relative* are resolved relative to the vocabulary mapping, not the base IRI.

Terms imported from *RDF 1.1 Concepts and Abstract Syntax* [RDF11-CONCEPTS], *RDF Schema 1.1* [RDF-SCHEMA], and *Linked Data Design Issues* [LINKED-DATA]

base IRI
The base IRI is an IRI established in the context, or is based on the JSON-LD document location. The base IRI is used to turn relative IRI references into IRIs.

blank node
A node in a graph that is neither an IRI, nor a literal. A blank node does not contain a de-referenceable identifier because it is either ephemeral in nature or does not contain information that needs to be linked to from outside of the linked data graph. In JSON-LD, a blank node is assigned an identifier starting with the prefix `_:`.

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 `_:`.

dataset

A dataset representing a collection of RDF graphs including exactly one default graph and zero or more named graphs.

datatype IRI

A datatype IRI is an IRI identifying a datatype that determines how the lexical form maps to a literal value.

default graph

The default graph of a dataset is an RDF graph having no name, which may be empty.

graph name

The IRI or blank node identifying a named graph.

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].

Processors may normalize language tags to lowercase.

Linked Data

A set of documents, each containing a representation of a linked data graph or dataset.

list

A list is an ordered sequence of IRIs, blank nodes, and literals.

literal

An object expressed as a value such as a string or number. Implicitly or explicitly includes a datatype IRI and, if the datatype is `rdf:langString`, an optional language tag.

named graph

A named graph is a linked data graph that is identified by an IRI or blank node.

node

A node in an RDF graph, either the subject and object of at least one triple. Note that a node can play both roles (subject and object) in a graph, even in the same triple.

object

An object is a node in a linked data graph with at least one incoming edge.

property

The name of a directed-arc in a linked data graph. Every property is directional and is labeled with an IRI or a blank node identifier.

Whenever possible, a property should be labeled with an IRI.

> NOTE
> The use of blank node identifiers to label properties is obsolete, and may be removed in a future version of JSON-LD.

Also, see ***predicate*** in [RDF11-CONCEPTS].

RDF graph
A labeled directed graph, i.e., a set of nodes connected by directed-arcs. Also called linked data graph.

resource
A resource denoted by an IRI, a blank node or literal representing something in the world (the "universe of discourse").

subject
A subject is a node in a linked data graph with at least one outgoing edge, related to an object node through a property.

triple
A component of an RDF graph including a subject, predicate, and object, which represents a node-arc-node segment of an RDF graph.

§ JSON-LD Specific Term Definitions

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

base direction
The base direction is the direction used when a string does not have a direction associated with it directly. It can be set in the context using the `@direction` key whose value must be one of the strings `"ltr"`, `"rtl"`, or `null`. See the Context Definitions section of JSON-LD 1.1 for a normative description.

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.

context
A set of rules for interpreting a JSON-LD document as described in the The Context section of JSON-LD 1.1, and normatively specified in the Context Definitions section of JSON-LD 1.1.

default language

The default language is the language used when a string does not have a language associated with it directly. It can be set in the context using the `@language` key whose value must be a string representing a [BCP47] language code or `null`. See the Context Definitions section of JSON-LD 1.1 for a normative description.

default object

A default object is a map that has a `@default` key.

embedded context

An embedded context is a context which appears as the `@context` entry of one of the following: a node object, a value object, a graph object, a list object, a set object, the value of a nested properties, or the value of an expanded term definition. Its value may be a map for a context definition, as an IRI, or as an array combining either of the above.

expanded term definition

An expanded term definition is a term definition where the value is a map containing one or more keyword keys to define the associated IRI, if this is a reverse property, the type associated with string values, and a container mapping. See the Expanded Term Definition section of JSON-LD 1.1 for a normative description.

frame

A JSON-LD document, which describes the form for transforming another JSON-LD document using matching and embedding rules. A frame document allows additional keywords and certain map entries to describe the matching and transforming process.

frame object

A frame object is a map element within a frame which represents a specific portion of the frame matching either a node object or a value object in the input. See the Frame Objects section of JSON-LD 1.1 for a normative description.

graph object

A graph object represents a named graph as the value of a map entry within a node object. When expanded, a graph object must have an `@graph` entry, and may also have `@id`, and `@index` entries. A ***simple graph object*** is a graph object which does not have an `@id` entry. Note that node objects may have a `@graph` entry, but are not considered graph objects if they include any other entries. A top-level object consisting of `@graph` is also not a graph object. Note that a node object may also represent a named graph it it includes other properties. See the Graph Objects section of JSON-LD 1.1 for a normative description.

id map

An id map is a map value of a term defined with `@container` set to `@id`. The

values of the id map must be node objects, and its keys are interpreted as IRIs representing the `@id` of the associated node object. If a value in the id map contains a key expanding to `@id`, its value must be equivalent to the referencing key in the id map. See the Id Maps section of JSON-LD 1.1 for a normative description.

implicitly named graph

A named graph created from the value of a map entry having an expanded term definition where `@container` is set to `@graph`.

included block

An included block is an entry in a node object where the key is either `@included` or an alias of `@included` and the value is one or more node objects. See the Included Blocks section of JSON-LD 1.1 for a normative description.

index map

An index map is a map value of a term defined with `@container` set to `@index`, whose values must be any of the following types: string, number, `true`, `false`, null, node object, value object, list object, set object, or an array of zero or more of the above possibilities. See the Index Maps section in JSON-LD 1.1 for a formal description.

JSON literal

A JSON literal is a literal where the associated datatype IRI is `rdf:JSON`. In the value object representation, the value of `@type` is `@json`. JSON literals represent values which are valid JSON [RFC8259]. See the The `rdf:JSON` Datatype section in JSON-LD 1.1 for a normative description.

JSON-LD document

A JSON-LD document is a serialization of an RDF dataset. See the JSON-LD Grammar section in JSON-LD 1.1 for a formal description.

JSON-LD internal representation

The JSON-LD internal representation is the result of transforming a JSON syntactic structure into the core data structures suitable for direct processing: arrays, maps, strings, numbers, booleans, and null.

JSON-LD Processor

A JSON-LD Processor is a system which can perform the algorithms defined in JSON-LD 1.1 Processing Algorithms and API. See the Conformance section in JSON-LD 1.1 API for a formal description.

JSON-LD value

A JSON-LD value is a string, a number, `true` or `false`, a typed value, or a language-tagged string. It represents an RDF literal.

keyword

A string that is specific to JSON-LD, described in the Syntax Tokens and

Keywords section of JSON-LD 1.1, and normatively specified in the Keywords section of JSON-LD 1.1,

language map

An language map is a map value of a term defined with `@container` set to `@language`, whose keys must be strings representing [BCP47] language codes and the values must be any of the following types: null, string, or an array of zero or more of the above possibilities. See the Language Maps section of JSON-LD 1.1 for a normative description.

list object

A list object is a map that has a `@list` key. It may also have an `@index` key, but no other entries. See the Lists and Sets section of JSON-LD 1.1 for a normative description.

local context

A context that is specified with a map, specified via the `@context` keyword.

nested property

A nested property is a key in a node object whose value is a map containing entries which are treated as if they were values of the node object. The nested property itself is semantically meaningless and used only to create a sub-structure within a node object. See the Property Nesting section of JSON-LD 1.1 for a normative description.

node object

A node object represents zero or more properties of a node in the graph serialized by the JSON-LD document. A map 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 map in the JSON-LD document consisting of no other entries than `@graph` and `@context`.

The entries of a node object whose keys are not keywords are also called properties of the node object. See the Node Objects section of JSON-LD 1.1 for a normative description.

node reference

A node object used to reference a node having only the `@id` key.

prefix

A prefix is the first component of a compact IRI which comes from a term that maps to a string that, when prepended to the suffix of the compact IRI, results in an IRI.

processing mode

The processing mode defines how a JSON-LD document is processed. By default, all documents are assumed to be conformant with this specification. By defining a different version using the `@version` entry in a

context, publishers can ensure that processors conformant with *JSON-LD 1.0* [JSON-LD10] will not accidentally process JSON-LD 1.1 documents, possibly creating a different output. The API provides an option for setting the processing mode to `json-ld-1.0`, which will prevent JSON-LD 1.1 features from being activated, or error if `@version` entry in a context is explicitly set to `1.1`. This specification extends *JSON-LD 1.0* via the `json-ld-1.1` processing mode.

scoped context

A scoped context is part of an expanded term definition using the `@context` entry. It has the same form as an embedded context. When the term is used as a type, it defines a ***type-scoped context***, when used as a property it defines a ***property-scoped context***.

set object

A set object is a map that has an `@set` entry. It may also have an `@index` key, but no other entries. See the Lists and Sets section of JSON-LD 1.1 for a normative description.

term

A term is a short word defined in a context that may be expanded to an IRI. See the Terms section of JSON-LD 1.1 for a normative description.

term definition

A term definition is an entry in a context, where the key defines a term which may be used within a map as a key, type, or elsewhere that a string is interpreted as a vocabulary item. Its value is either a string (***simple term definition***), expanding to an IRI, or a map (expanded term definition).

type map

A type map is a map value of a term defined with `@container` set to `@type`, whose keys are interpreted as IRIs representing the `@type` of the associated node object; the value must be a node object, or array of node objects. If the value contains a term expanding to `@type`, its values are merged with the map value when expanding. See the Type Maps section of JSON-LD 1.1 for a normative description.

typed value

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

value object

A value object is a map that has an `@value` entry.

vocabulary mapping

The vocabulary mapping is set in the context using the `@vocab` key whose value must be an IRI, a compact IRI, a term, or `null`. See the Value

Objects section of JSON-LD 1.1 for a normative description.

§ 1.5 Design Goals and Rationale

This section is non-normative.

JSON-LD satisfies the following design goals:

Simplicity
No extra processors or software libraries are necessary to use JSON-LD in its most basic form. The language provides developers with a very easy learning curve. Developers not concerned with Linked Data only need to understand JSON, and know to include but ignore the `@context` property, to use the basic functionality in JSON-LD.

Compatibility
A JSON-LD document is always a valid JSON document. This ensures that all of the standard JSON libraries work seamlessly with JSON-LD documents.

Expressiveness
The syntax serializes labeled directed graphs. This ensures that almost every real world data model can be expressed.

Terseness
The JSON-LD syntax is very terse and human readable, requiring as little effort as possible from the developer.

Zero Edits, most of the time
JSON-LD ensures a smooth and simple transition from existing JSON-based systems. In many cases, zero edits to the JSON document and the addition of one line to the HTTP response should suffice (see § 6.1 Interpreting JSON as JSON-LD). This allows organizations that have already deployed large JSON-based infrastructure to use JSON-LD's features in a way that is not disruptive to their day-to-day operations and is transparent to their current customers. However, there are times where mapping JSON to a graph representation is a complex undertaking. In these instances, rather than extending JSON-LD to support esoteric use cases, we chose not to support the use case. While Zero Edits is a design goal, it is not always possible without adding great complexity to the language. JSON-LD focuses on simplicity when possible.

Usable as RDF
JSON-LD is usable by developers as idiomatic JSON, with no need to understand RDF [RDF11-CONCEPTS]. JSON-LD is also usable as RDF, so people intending to use JSON-LD with RDF tools will find it can be used

like any other RDF syntax. Complete details of how JSON-LD relates to RDF are in section § 10. Relationship to RDF.

§ 1.6 Data Model Overview

This section is non-normative.

Generally speaking, the data model described by a JSON-LD document is a labeled, directed graph. The graph contains nodes, which are connected by directed-arcs. A node is either a resource with properties, or the data values of those properties including strings, numbers, typed values (like dates and times) and IRIs.

Within a directed graph, nodes are resources, and may be *unnamed*, i.e., not identified by an IRI; which are called blank nodes, and may be identified using a blank node identifier. These identifiers may be required to represent a fully connected graph using a tree structure, such as JSON, but otherwise have no intrinsic meaning. Literal values, such as strings and numbers, are also considered resources, and JSON-LD distinguishes between node objects and value objects to distinguish between the different kinds of resource.

This simple data model is incredibly flexible and powerful, capable of modeling almost any kind of data. For a deeper explanation of the data model, see section § 8. Data Model.

Developers who are familiar with Linked Data technologies will recognize the data model as the RDF Data Model. To dive deeper into how JSON-LD and RDF are related, see section § 10. Relationship to RDF.

At the surface level, a JSON-LD document is simply *JSON*, detailed in [RFC8259]. For the purpose of describing the core data structures, this is limited to arrays, maps (the parsed version of a JSON Object), strings, numbers, booleans, and null, called the JSON-LD internal representation. This allows surface syntaxes other than JSON to be manipulated using the same algorithms, when the syntax maps to equivalent core data structures.

NOTE

Although not discussed in this specification, parallel work using *YAML Ain't Markup Language (YAML™) Version 1.2* [YAML] and binary representations such as *Concise Binary Object Representation (CBOR)* [RFC7049] could be used to map into the internal representation, allowing the JSON-LD 1.1 API [JSON-LD11-API] to operate as if the source was a JSON document.

§ 1.7 Syntax Tokens and Keywords

This section is non-normative.

JSON-LD specifies a number of syntax tokens and keywords that are a core part of the language. A normative description of the keywords is given in § 9.16 Keywords.

:
: The separator for JSON keys and values that use compact IRIs.

@base
: Used to set the base IRI against which to resolve those relative IRI references which are otherwise interpreted relative to the document. This keyword is described in § 4.1.3 Base IRI.

@container
: Used to set the default container type for a term. This keyword is described in the following sections:

- § 4.3 Value Ordering,
- § 4.6.1 Data Indexing,
- § 4.6.2 Language Indexing,
- § 4.6.3 Node Identifier Indexing,
- § 4.6.4 Node Type Indexing
- § 4.9 Named Graphs,
- § 4.9.3 Named Graph Indexing, and
- § 4.9.2 Named Graph Data Indexing

@context
: Used to define the short-hand names that are used throughout a JSON-LD document. These short-hand names are called terms and help developers to express specific identifiers in a compact manner. The `@context` keyword

is described in detail in § 3.1 The Context.

@direction
Used to set the base direction of a JSON-LD value, which are not typed values (e.g. strings, or language-tagged strings). This keyword is described in § 4.2.4 String Internationalization.

@graph
Used to express a graph. This keyword is described in § 4.9 Named Graphs.

@id
Used to uniquely identify node objects that are being described in the document with IRIs or blank node identifiers. This keyword is described in § 3.3 Node Identifiers. A node reference is a node object containing only the `@id` property, which may represent a reference to a node object found elsewhere in the document.

@import
Used in a context definition to load an external context within which the containing context definition is merged. This can be useful to add JSON-LD 1.1 features to JSON-LD 1.0 contexts.

@included
Used in a top-level node object to define an included block, for including secondary node objects within another node object.

@index
Used to specify that a container is used to index information and that processing should continue deeper into a JSON data structure. This keyword is described in § 4.6.1 Data Indexing.

@json
Used as the `@type` value of a JSON literal. This keyword is described in § 4.2.2 JSON Literals.

@language
Used to specify the language for a particular string value or the default language of a JSON-LD document. This keyword is described in § 4.2.4 String Internationalization.

@list
Used to express an ordered set of data. This keyword is described in § 4.3.1 Lists.

@nest
Used to define a property of a node object that groups together properties of that node, but is not an edge in the graph.

@none
Used as an index value in an index map, id map, language map, type map,

or elsewhere where a map is used to index into other values, when the indexed node does not have the feature being indexed.

@prefix
With the value `true`, allows this term to be used to construct a compact IRI when compacting.

@propagate
Used in a context definition to change the scope of that context. By default, it is `true`, meaning that contexts propagate across node objects (other than for type-scoped contexts, which default to `false`). Setting this to `false` causes term definitions created within that context to be removed when entering a new node object.

@protected
Used to prevent term definitions of a context to be overridden by other contexts. This keyword is described in § 4.1.11 Protected Term Definitions.

@reverse
Used to express reverse properties. This keyword is described in § 4.8 Reverse Properties.

@set
Used to express an unordered set of data and to ensure that values are always represented as arrays. This keyword is described in § 4.3.2 Sets.

@type
Used to set the type of a node or the datatype of a typed value. This keyword is described further in § 3.5 Specifying the Type and § 4.2.1 Typed Values.

> NOTE
> The use of `@type` to define a type for both node objects and value objects addresses the basic need to type data, be it a literal value or a more complicated resource. Experts may find the overloaded use of the `@type` keyword for both purposes concerning, but should note that Web developer usage of this feature over multiple years has not resulted in its misuse due to the far less frequent use of `@type` to express typed literal values.

@value
Used to specify the data that is associated with a particular property in the graph. This keyword is described in § 4.2.4 String Internationalization and § 4.2.1 Typed Values.

@version
Used in a context definition to set the processing mode. New features

since *JSON-LD 1.0* [JSON-LD10] described in this specification are not available when processing mode has been explicitly set to `json-ld-1.0`.

> NOTE
> Within a context definition `@version` takes the specific value `1.1`, not `"json-ld-1.1"`, as a JSON-LD 1.0 processor may accept a string value for `@version`, but will reject a numeric value.

> NOTE
> The use of `1.1` for the value of `@version` is intended to cause a JSON-LD 1.0 processor to stop processing. Although it is clearly meant to be related to JSON-LD 1.1, it does not otherwise adhere to the requirements for Semantic Versioning.

`@vocab`
: Used to expand properties and values in `@type` with a common prefix IRI. This keyword is described in § 4.1.2 Default Vocabulary.

All keys, keywords, and values in JSON-LD are case-sensitive.

§ 2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *RECOMMENDED*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

A JSON-LD document complies with this specification if it follows the normative statements in appendix § 9. JSON-LD Grammar. JSON documents can be interpreted as JSON-LD by following the normative statements in § 6.1 Interpreting JSON as JSON-LD. For convenience, normative statements for documents are often phrased as statements on the properties of the document.

This specification makes use of the following namespace prefixes:

Prefix	IRI

Prefix	IRI
dc11	http://purl.org/dc/elements/1.1/
dcterms	http://purl.org/dc/terms/
cred	https://w3id.org/credentials#
foaf	http://xmlns.com/foaf/0.1/
geojson	https://purl.org/geojson/vocab#
prov	http://www.w3.org/ns/prov#
i18n	https://www.w3.org/ns/i18n#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
schema	http://schema.org/
skos	http://www.w3.org/2004/02/skos/core#
xsd	http://www.w3.org/2001/XMLSchema#

These are used within this document as part of a compact IRI as a shorthand for the resulting IRI, such as `dcterms:title` used to represent `http://purl.org/dc/terms/title`.

§ 3. Basic Concepts

This section is non-normative.

JSON [RFC8259] is a lightweight, language-independent data interchange format. It is easy to parse and easy to generate. However, it is difficult to integrate JSON from different sources as the data may contain keys that conflict with other data sources. Furthermore, JSON has no built-in support for hyperlinks, which are a fundamental building block on the Web. Let's start by looking at an example that we will be using for the rest of this section:

EXAMPLE 2: Sample JSON document

```
{
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

It's obvious to humans that the data is about a person whose `name` is "Manu

Sporny" and that the `homepage` property contains the URL of that person's homepage. A machine doesn't have such an intuitive understanding and sometimes, even for humans, it is difficult to resolve ambiguities in such representations. This problem can be solved by using unambiguous identifiers to denote the different concepts instead of tokens such as "name", "homepage", etc.

Linked Data, and the Web in general, uses IRIs (Internationalized Resource Identifiers as described in [RFC3987]) for unambiguous identification. The idea is to use IRIs to assign unambiguous identifiers to data that may be of use to other developers. It is useful for terms, like `name` and `homepage`, to expand to IRIs so that developers don't accidentally step on each other's terms. Furthermore, developers and machines are able to use this IRI (by using a web browser, for instance) to go to the term and get a definition of what the term means. This process is known as IRI dereferencing.

Leveraging the popular schema.org vocabulary, the example above could be unambiguously expressed as follows:

EXAMPLE 3: Sample JSON-LD document using full IRIs instead of terms

Expanded (Input) | Statements | Turtle (Result) | Open in playground

Input

```
{
  "http://schema.org/name": "Manu Sporny",
  "http://schema.org/url": {
    "@id": "http://manu.sporny.org/"
    ↑ The '@id' keyword means 'This value is an identifier that is an I
  },
  "http://schema.org/image": {
    "@id": "http://manu.sporny.org/images/manu.png"
  }
}
```

In the example above, every property is unambiguously identified by an IRI and all values representing IRIs are explicitly marked as such by the `@id` keyword. While this is a valid JSON-LD document that is very specific about its data, the document is also overly verbose and difficult to work with for human developers. To address this issue, JSON-LD introduces the notion of a context as described in the next section.

This section only covers the most basic features of JSON-LD. More advanced features, including typed values, indexed values, and named graphs, can be found in § 4. Advanced Concepts.

§ 3.1 The Context

This section is non-normative.

When two people communicate with one another, the conversation takes place in a shared environment, typically called "the context of the conversation". This shared context allows the individuals to use shortcut terms, like the first name of a mutual friend, to communicate more quickly but without losing accuracy. A context in JSON-LD works in the same way. It allows two applications to use shortcut terms to communicate with one another more efficiently, but without losing accuracy.

Simply speaking, a context is used to map terms to IRIs. Terms are case sensitive and most valid strings that are not reserved JSON-LD keywords can be used as a term. Exceptions are the empty string `""` and strings that have the form of a keyword (i.e., starting with `"@"` followed exclusively by one or more *ALPHA* characters (see [RFC5234])), which must not be used as terms. Strings that have the form of an IRI (e.g., containing a `":"`) should not be used as terms.

For the sample document in the previous section, a context would look something like this:

EXAMPLE 4: Context for the sample document in the previous section

Context

```
{
  "@context": {
    "name": "http://schema.org/name",
    ↑ This means that 'name' is shorthand for 'http://schema.org/name'
    "image": {
      "@id": "http://schema.org/image",
      ↑ This means that 'image' is shorthand for 'http://schema.org/ima
      "@type": "@id"
      ↑ This means that a string value associated with 'image'
        should be interpreted as an identifier that is an IRI
    },
    "homepage": {
      "@id": "http://schema.org/url",
      ↑ This means that 'homepage' is shorthand for 'http://schema.org/
      "@type": "@id"
      ↑ This means that a string value associated with 'homepage'
        should be interpreted as an identifier that is an IRI
    }
  }
}
```

As the context above shows, the value of a term definition can either be a simple string, mapping the term to an IRI, or a map.

A context is introduced using an entry with the key `@context` and may appear within a node object or a value object.

When an entry with a term key has a map value, the map is called an expanded term definition. The example above specifies that the values of `image` and `homepage`, if they are strings, are to be interpreted as IRIs. Expanded term definitions also allow terms to be used for index maps and to specify whether array values are to be interpreted as sets or lists. Expanded term definitions may be defined using IRIs or compact IRIs as keys, which is mainly used to associate type or language information with an IRIs or compact IRI.

Contexts can either be directly embedded into the document (an embedded context) or be referenced using a URL. Assuming the context document in the previous example can be retrieved at `https://json-ld.org/contexts/person.jsonld`, it can be referenced by adding a single line and allows a JSON-LD document to be expressed much more concisely as shown in the example below:

EXAMPLE 5: Referencing a JSON-LD context

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": "https://json-ld.org/contexts/person.jsonld",
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

The referenced context not only specifies how the terms map to IRIs in the Schema.org vocabulary but also specifies that string values associated with the `homepage` and `image` property can be interpreted as an IRI (`"@type": "@id"`, see § 3.2 IRIs for more details). This information allows developers to re-use each other's data without having to agree to how their data will interoperate on a site-by-site basis. External JSON-LD context documents may contain extra information located outside of the `@context` key, such as documentation about the terms declared in the document. Information contained outside of the `@context` value is ignored when the document is used as an external ***JSON-LD context document***.

A remote context may also be referenced using a relative URL, which is resolved relative to the location of the document containing the reference. For example, if a document were located at `http://example.org/document.jsonld` and contained a relative reference to `context.jsonld`, the referenced context document would be found relative at `http://example.org/context.jsonld`.

EXAMPLE 6: Loading a relative context

```
{
  "@context": "context.jsonld",
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

NOTE

Resolution of relative references to context URLs also applies to remote context documents, as they may themselves contain references to other contexts.

JSON documents can be interpreted as JSON-LD without having to be modified by referencing a context via an HTTP Link Header as described in § 6.1 Interpreting JSON as JSON-LD. It is also possible to apply a custom context using the JSON-LD 1.1 API [JSON-LD11-API].

In JSON-LD documents, contexts may also be specified inline. This has the advantage that documents can be processed even in the absence of a connection to the Web. Ultimately, this is a modeling decision and different use cases may require different handling. See Security Considerations in § C. IANA Considerations for a discussion on using remote contexts.

EXAMPLE 7: In-line context definition

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "name": "http://schema.org/name",
    "image": {
      "@id": "http://schema.org/image",
      "@type": "@id"
    },
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  },
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

This section only covers the most basic features of the JSON-LD Context. The Context can also be used to help interpret other more complex JSON data structures, such as indexed values, ordered values, and nested properties. More advanced features related to the JSON-LD Context are covered in § 4. Advanced Concepts.

§ 3.2 IRIs

This section is non-normative.

IRIs (Internationalized Resource Identifiers [RFC3987]) are fundamental to Linked Data as that is how most nodes and properties are identified. In JSON-LD, IRIs may be represented as an IRI reference. An IRI is defined in [RFC3987] as containing a *scheme* along with *path* and optional *query* and *fragment* segments. A relative IRI reference is an IRI that is relative to some other IRI. In JSON-LD, with exceptions that are as described below, all relative IRI references are resolved relative to the base IRI.

> NOTE
>
> As noted in § 1.1 How to Read this Document, IRIs can often be confused with URLs (Uniform Resource Locators), the primary distinction is that a URL *locates* a resource on the web, an IRI *identifies* a resource. While it is a good practice for resource identifiers to be dereferenceable, sometimes this is not practical. In particular, note the [URN] scheme for Uniform Resource Names, such as *UUID*. An example UUID is `urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6`.

> NOTE
>
> Properties, values of `@type`, and values of properties with a term definition that defines them as being relative to the vocabulary mapping, may have the form of a relative IRI reference, but are resolved using the vocabulary mapping, and not the base IRI.

A string is interpreted as an IRI when it is the value of a map entry with the key `@id`:

EXAMPLE 8: Values of @id are interpreted as IRI

```
{
  ...
  "homepage": { "@id": "http://example.com/" }
  ...
}
```

Values that are interpreted as IRIs, can also be expressed as relative IRI

references. For example, assuming that the following document is located at `http://example.com/about/`, the relative IRI reference `../` would expand to `http://example.com/` (for more information on where relative IRI references can be used, please refer to section § 9. JSON-LD Grammar).

EXAMPLE 9: IRIs can be relative

```
{
  ...
  "homepage": { "@id": "../" }
  ...
}
```

IRIs can be expressed directly in the key position like so:

EXAMPLE 10: IRI as a key

```
{
  ...
  "http://schema.org/name": "Manu Sporny",
  ...
}
```

In the example above, the key `http://schema.org/name` is interpreted as an IRI.

Term-to-IRI expansion occurs if the key matches a term defined within the active context:

EXAMPLE 11: Term expansion from context definition

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "name": "http://schema.org/name"
  },
  "name": "Manu Sporny",
  "status": "trollin'"
}
```

JSON keys that do not expand to an IRI, such as `status` in the example above, are not Linked Data and thus ignored when processed.

If type coercion rules are specified in the `@context` for a particular term or property IRI, an IRI is generated:

EXAMPLE 12: Type coercion

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    ...
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
    ...
  },
  ...
  "homepage": "http://manu.sporny.org/"
  ...
}
```

In the example above, since the value `http://manu.sporny.org/` is expressed as a JSON string, the type coercion rules will transform the value into an IRI when processing the data. See § 4.2.3 Type Coercion for more details about this feature.

In summary, IRIs can be expressed in a variety of different ways in JSON-LD:

1. Map entries that have a key mapping to a term in the active context expand to an IRI (only applies outside of the context definition).
2. An IRI is generated for the string value specified using `@id` or `@type`.
3. An IRI is generated for the string value of any key for which there are coercion rules that contain an `@type` key that is set to a value of `@id` or `@vocab`.

This section only covers the most basic features associated with IRIs in JSON-LD. More advanced features related to IRIs are covered in section § 4. Advanced Concepts.

§ 3.3 Node Identifiers

This section is non-normative.

To be able to externally reference nodes in an RDF graph, it is important that nodes have an identifier. IRIs are a fundamental concept of Linked Data, for nodes to be truly linked, dereferencing the identifier should result in a representation of that node. This may allow an application to retrieve further information about a node.

In JSON-LD, a node is identified using the `@id` keyword:

EXAMPLE 13: Identifying a node

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    ...
    "name": "http://schema.org/name"
  },
  "@id": "http://me.markus-lanthaler.com/",
  "name": "Markus Lanthaler",
  ...
}
```

The example above contains a node object identified by the IRI `http://me.markus-lanthaler.com/`.

This section only covers the most basic features associated with node identifiers in JSON-LD. More advanced features related to node identifiers are covered in section § 4. Advanced Concepts.

§ 3.4 Uses of JSON Objects

This section is non-normative.

As a syntax, JSON has only a limited number of syntactic elements:

- Numbers, which describe literal numeric values,
- Strings, which may describe literal string values, or be used as the keys in a JSON object.
- Boolean `true` and `false`, which describe literal boolean values,
- `null`, which describes the absence of a value,
- Arrays, which describe an ordered set of values of any type, and
- JSON objects, which provide a set of map entries, relating keys with

values.

The JSON-LD data model allows for a richer set of resources, based on the RDF data model. The data model is described more fully in § 8. Data Model. JSON-LD uses JSON objects to describe various resources, along with the relationships between these resources:

Node objects
: Node objects are used to define nodes in the linked data graph which may have both incoming and outgoing edges. Node objects are principle structure for defining resources having properties. See § 9.2 Node Objects for the normative definition.

Value objects
: Value objects are used for describing literal nodes in a linked data graph which may have only incoming edges. In JSON, some literal nodes may be described without the use of a JSON object (e.g., numbers, strings, and boolean values), but in the expanded form, all literal nodes are described using value objects. See § 4.2 Describing Values for more information, and § 9.5 Value Objects for the normative definition.

List Objects and Set objects
: List Objects are a special kind of JSON-LD maps, distinct from node objects and value objects, used to express ordered values by wrapping an array in a map under the key `@list`. Set Objects exist for uniformity, and are equivalent to the array value of the `@set` key. See § 4.3.1 Lists and § 4.3.2 Sets for more detail.

Map Objects
: JSON-LD uses various forms of maps as ways to more easily access values of a property.

  Language Maps
  : Allows multiple values differing in their associated language to be indexed by language tag. See § 4.6.2 Language Indexing for more information, and § 9.8 Language Maps for the normative definition.

  Index Maps
  : Allows multiple values (node objects or value objects) to be indexed by an associated `@index`. See § 4.6.1 Data Indexing for more information, and § 9.9 Index Maps for the normative definition.

  Id Maps
  : Allows multiple node objects to be indexed by an associated `@id`. See § 4.6.3 Node Identifier Indexing for more information, and § 9.11 Id Maps for the normative definition.

  Type Maps
  : Allows multiple node objects to be indexed by an associated `@type`.

See § 4.6.4 Node Type Indexing for more information, and § 9.12 Type Maps for the normative definition.

Named Graph Indexing
Allows multiple named graphs to be indexed by an associated graph name. See § 4.9.3 Named Graph Indexing for more information.

Graph objects
A graph object is much like a node object, except that it defines a named graph. See § 4.9 Named Graphs for more information, and § 9.4 Graph Objects for the normative definition. A node object may also describe a named graph, in addition to other properties defined on the node. The notable difference is that a graph object only describes a named graph.

Context Definitions
A Context Definition uses the JSON object form, but is not itself data in a linked data graph. A Context Definition also may contain expanded term definitions, which are also represented using JSON objects. See § 3.1 The Context, § 4.1 Advanced Context Usage for more information, and § 9.15 Context Definitions for the normative definition.

§ 3.5 Specifying the Type

This section is non-normative.

In Linked Data, it is common to specify the type of a graph node; in many cases, this can be inferred based on the properties used within a given node object, or the property for which a node is a value. For example, in the *schema.org* vocabulary, the *givenName* property is associated with a *Person*. Therefore, one may reason that if a node object contains the property *givenName*, that the type is a *Person*; making this explicit with `@type` helps to clarify the association.

The type of a particular node can be specified using the `@type` keyword. In Linked Data, types are uniquely identified with an IRI.

EXAMPLE 14: Specifying the type for a node

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    ...
    "givenName": "http://schema.org/givenName",
    "familyName": "http://schema.org/familyName"
  },
  "@id": "http://me.markus-lanthaler.com/",
  "@type": "http://schema.org/Person",
  "givenName": "Markus",
  "familyName": "Lanthaler",
  ...
}
```

A node can be assigned more than one type by using an array:

EXAMPLE 15: Specifying multiple types for a node

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  ...
  "@id": "http://me.markus-lanthaler.com/",
  "@type": [
     "http://schema.org/Person",
     "http://xmlns.com/foaf/0.1/Person"
    ],
  ...
}
```

The value of a `@type` key may also be a term defined in the active context:

EXAMPLE 16: Using a term to specify the type

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    ...
    "Person": "http://schema.org/Person"
  },
  "@id": "http://example.org/places#BrewEats",
  "@type": "Person",
  ...
}
```

In addition to setting the type of nodes, `@type` can also be used to set the type of a value to create a typed value. This use of `@type` is similar to that used to define the type of a node object, but value objects are restricted to having just a single type. The use of `@type` to create typed values is discussed more fully in § 4.2.1 Typed Values.

Typed values can also be defined implicitly, by specifying `@type` in an expanded term definition. This is covered more fully in § 4.2.3 Type Coercion.

§ 4. Advanced Concepts

This section is non-normative.

JSON-LD has a number of features that provide functionality above and beyond the core functionality described above. JSON can be used to express data using such structures, and the features described in this section can be used to interpret a variety of different JSON structures as Linked Data. A JSON-LD processor will make use of provided and embedded contexts to interpret property values in a number of different idiomatic ways.

Describing values

One pattern in JSON is for the value of a property to be a string. Often times, this string actually represents some other typed value, for example an IRI, a date, or a string in some specific language. See § 4.2 Describing Values for details on how to describe such value typing.

Value ordering

In JSON, a property with an array value implies an implicit order; arrays in JSON-LD do not convey any ordering of the contained elements by

default, unless defined using embedded structures or through a context definition. See § 4.3 Value Ordering for a further discussion.

Property nesting

Another JSON idiom often found in APIs is to use an intermediate object to group together related properties of an object; in JSON-LD these are referred to as nested properties and are described in § 4.4 Nested Properties.

Referencing objects

Linked Data is all about describing the relationships between different resources. Sometimes these relationships are between resources defined in different documents described on the web, sometimes the resources are described within the same document.

EXAMPLE 17: Referencing Objects on the Web

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/",
    "knows": {"@type": "@id"}
  },
  "@id": "http://manu.sporny.org/about#manu",
  "@type": "Person",
  "name": "Manu Sporny",
  "knows": "https://greggkellogg.net/foaf#me"
}
```

In this case, a document residing at `http://manu.sporny.org/about` may contain the example above, and reference another document at `https://greggkellogg.net/foaf` which could include a similar representation.

A common idiom found in JSON usage is objects being specified as the value of other objects, called object embedding in JSON-LD; for example, a friend specified as an object value of a *Person*:

EXAMPLE 18: Embedding Objects

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/"
  },
  "@id": "http://manu.sporny.org/about#manu",
  "@type": "Person",
  "name": "Manu Sporny",
  "knows": {
    "@id": "https://greggkellogg.net/foaf#me",
    "@type": "Person",
    "name": "Gregg Kellogg"
  }
}
```

See § 4.5 Embedding details these relationships.

Indexed values

Another common idiom in JSON is to use an intermediate object to represent property values via indexing. JSON-LD allows data to be indexed in a number of different ways, as detailed in § 4.6 Indexed Values.

Reverse Properties

JSON-LD serializes directed graphs. That means that every property points from a node to another node or value. However, in some cases, it is desirable to serialize in the reverse direction, as detailed in § 4.8 Reverse Properties.

The following sections describe such advanced functionality in more detail.

§ 4.1 Advanced Context Usage

This section is non-normative.

Section § 3.1 The Context introduced the basics of what makes JSON-LD work. This section expands on the basic principles of the context and demonstrates how more advanced use cases can be achieved using JSON-LD.

In general, contexts may be used any time a map is defined. The only time that one cannot express a context is as a direct child of another context

definition (other than as part of an expanded term definition). For example, a JSON-LD document may have the form of an array composed of one or more node objects, which use a context definition in each top-level node object:

EXAMPLE 19: Using multiple contexts

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
[
  {
    "@context": "https://json-ld.org/contexts/person.jsonld",
    "name": "Manu Sporny",
    "homepage": "http://manu.sporny.org/",
    "depiction": "http://twitter.com/account/profile_image/manusporny"
  }, {
    "@context": "https://json-ld.org/contexts/place.jsonld",
    "name": "The Empire State Building",
    "description": "The Empire State Building is a 102-story landmark i
    "geo": {
      "latitude": "40.75",
      "longitude": "73.98"
    }
  }
]
```

The outer array is standard for a document in expanded document form and flattened document form, and may be necessary when describing a disconnected graph, where nodes may not reference each other. In such cases, using a top-level map with a `@graph` property can be useful for saving the repetition of `@context`. See § 4.5 Embedding for more.

EXAMPLE 20: Describing disconnected nodes with @graph

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": [
    "https://json-ld.org/contexts/person.jsonld",
    "https://json-ld.org/contexts/place.jsonld",
    {"title": "http://purl.org/dc/terms/title"}
  ],
  "@graph": [{
    "http://xmlns.com/foaf/0.1/name": "Manu Sporny",
    "homepage": "http://manu.sporny.org/",
    "depiction": "http://twitter.com/account/profile_image/manusporny"
  }, {
    "title": "The Empire State Building",
    "description": "The Empire State Building is a 102-story landmark i
    "geo": {
      "latitude": "40.75",
      "longitude": "73.98"
    }
  }]
}
```

Duplicate context terms are overridden using a most-recently-defined-wins mechanism.

EXAMPLE 21: Embedded contexts within node objects

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "name": "http://example.com/person#name",
    "details": "http://example.com/person#details"
  },
  "name": "Markus Lanthaler",
  ...
  "details": {
    "@context": {
      "name": "http://example.com/organization#name"
    },
    "name": "Graz University of Technology"
  }
}
```

In the example above, the `name` term is overridden in the more deeply nested `details` structure, which uses its own embedded context. Note that this is rarely a good authoring practice and is typically used when working with legacy applications that depend on a specific structure of the map. If a term is redefined within a context, all previous rules associated with the previous definition are removed. If a term is redefined to `null`, the term is effectively removed from the list of terms defined in the active context.

Multiple contexts may be combined using an array, which is processed in order. The set of contexts defined within a specific map are referred to as local contexts. The active context refers to the accumulation of local contexts that are in scope at a specific point within the document. Setting a local context to `null` effectively resets the active context to an empty context, without term definitions, default language, or other things defined within previous contexts. The following example specifies an external context and then layers an embedded context on top of the external context:

In JSON-LD 1.1, there are other mechanisms for introducing contexts, including scoped contexts and imported contexts, and there are new ways of protecting term definitions, so there are cases where the last defined inline context is not necessarily one which defines the scope of terms. See § 4.1.8 Scoped Contexts, § 4.1.9 Context Propagation, § 4.1.10 Imported Contexts, and § 4.1.11 Protected Term Definitions for further information.

EXAMPLE 22: Combining external and local contexts

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": [
    "https://json-ld.org/contexts/person.jsonld",
    {
      "pic": {
        "@id": "http://xmlns.com/foaf/0.1/depiction",
        "@type": "@id"
      }
    }
  ],
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "pic": "http://twitter.com/account/profile_image/manusporny"
}
```

NOTE

When possible, the context definition should be put at the top of a JSON-LD document. This makes the document easier to read and might make streaming parsers more efficient. Documents that do not have the context at the top are still conformant JSON-LD.

NOTE

To avoid forward-compatibility issues, terms starting with an `@` character followed exclusively by one or more *ALPHA* characters (see [RFC5234]) are to be avoided as they might be used as keyword in future versions of JSON-LD. Terms starting with an `@` character that are not JSON-LD 1.1 keywords are treated as any other term, i.e., they are ignored unless mapped to an IRI. Furthermore, the use of empty terms (`""`) is not allowed as not all programming languages are able to handle empty JSON keys.

§ 4.1.1 JSON-LD 1.1 Processing Mode

This section is non-normative.

New features defined in JSON-LD 1.1 are available unless the processing mode is set to `json-ld-1.0`. This may be set through an API option. The processing mode may be explicitly set to `json-ld-1.1` using the `@version` entry in a context set to the value `1.1` as a number, or through an API option. Explicitly setting the processing mode to `json-ld-1.1` will prohibit JSON-LD 1.0 processors from incorrectly processing a JSON-LD 1.1 document.

EXAMPLE 23: Setting @version in context

```
{
  "@context": {
    "@version": 1.1,
    ...
  },
  ...
}
```

The first `context` encountered when processing a document which contains `@version` determines the `processing mode`, unless it is defined explicitly

through an API option. This means that if `"@version": 1.1` is encountered after processing a context without `@version`, the former will be interpreted as having had `"@version": 1.1` defined within it.

> NOTE
>
> Setting the processing mode explicitly to `json-ld-1.1` is *RECOMMENDED* to prevent a *JSON-LD 1.0* processor from incorrectly processing a JSON-LD 1.1 document and producing different results.

§ 4.1.2 Default Vocabulary

This section is non-normative.

At times, all properties and types may come from the same vocabulary. JSON-LD's `@vocab` keyword allows an author to set a common prefix which is used as the vocabulary mapping and is used for all properties and types that do not match a term and are neither an IRI nor a compact IRI (i.e., they do not contain a colon).

EXAMPLE 24: Using a default vocabulary

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://example.com/vocab/"
  },
  "@id": "http://example.org/places#BrewEats",
  "@type": "Restaurant",
  "name": "Brew Eats"
  ...
}
```

If `@vocab` is used but certain keys in an map should not be expanded using the vocabulary IRI, a term can be explicitly set to null in the context. For instance, in the example below the `databaseId` entry would not expand to an IRI causing the property to be dropped when expanding.

EXAMPLE 25: Using the null keyword to ignore data

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
     "@vocab": "http://example.com/vocab/",
     "databaseId": null
  },
  "@id": "http://example.org/places#BrewEats",
  "@type": "Restaurant",
  "name": "Brew Eats",
  "databaseId": "23987520"
}
```

Since JSON-LD 1.1, the vocabulary mapping in a local context can be set to the a relative IRI reference, which is concatenated to any vocabulary mapping in the active context (see § 4.1.4 Using the Document Base for the Default Vocabulary for how this applies if there is no vocabulary mapping in the active context).

The following example illustrates the affect of expanding a property using a relative IRI reference, which is shown in the Expanded (Result) tab below.

EXAMPLE 26: Using a default vocabulary relative to a previous default vocabulary

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": [{
    "@vocab": "http://example.com/"
  }, {
    "@version": 1.1,
    "@vocab": "vocab/"
  }],
  "@id": "http://example.org/places#BrewEats",
  "@type": "Restaurant",
  "name": "Brew Eats"
  ...
}
```

NOTE

The grammar for `@vocab`, as defined in § 9.15 Context Definitions allows the value to be a term or compact IRI. Note that terms used in the value of `@vocab` must be in scope at the time the context is introduced, otherwise there would be a circular dependency between `@vocab` and other terms defined in the same context.

§ 4.1.3 Base IRI

This section is non-normative.

JSON-LD allows IRIs to be specified in a relative form which is resolved against the document base according section 5.1 Establishing a Base URI of [RFC3986]. The base IRI may be explicitly set with a context using the `@base` keyword.

For example, if a JSON-LD document was retrieved from `http://example.com/document.jsonld`, relative IRI references would resolve against that IRI:

EXAMPLE 27: Use a relative IRI reference as node identifier

```
{
  "@context": {
    "label": "http://www.w3.org/2000/01/rdf-schema#label"
  },
  "@id": "",
  "label": "Just a simple document"
}
```

This document uses an empty `@id`, which resolves to the document base. However, if the document is moved to a different location, the IRI would change. To prevent this without having to use an IRI, a context may define an `@base` mapping, to overwrite the base IRI for the document.

EXAMPLE 28: Setting the document base in a document

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@base": "http://example.com/document.jsonld",
    "label": "http://www.w3.org/2000/01/rdf-schema#label"
  },
  "@id": "",
  "label": "Just a simple document"
}
```

Setting `@base` to null will prevent relative IRI references from being expanded to IRIs.

Please note that the `@base` will be ignored if used in external contexts.

§ 4.1.4 Using the Document Base for the Default Vocabulary

This section is non-normative.

In some cases, vocabulary terms are defined directly within the document itself, rather than in an external vocabulary. Since JSON-LD 1.1, the vocabulary mapping in a local context can be set to a relative IRI reference, which is, if there is no vocabulary mapping in scope, resolved against the base IRI. This causes terms which are expanded relative to the vocabulary, such as the keys of node objects, to be based on the base IRI to create IRIs.

EXAMPLE 29: Using "#" as the vocabulary mapping

Input

```
{
  "@context": {
    "@version": 1.1,
    "@base": "http://example/document",
    "@vocab": "#"
  },
  "@id": "http://example.org/places#BrewEats",
  "@type": "Restaurant",
  "name": "Brew Eats"
  ...
}
```

If this document were located at `http://example/document`, it would expand as follows:

EXAMPLE 30: Using "#" as the vocabulary mapping (expanded)

Expanded (Result) | Statements | Turtle | Open in playground

Result

```
[{
  "@id": "http://example.org/places#BrewEats",
  "@type": ["http://example/document#Restaurant"],
  "http://example/document#name": [{"@value": "Brew Eats"}]
}]
```

§ 4.1.5 Compact IRIs

This section is non-normative.

A compact IRI is a way of expressing an IRI using a *prefix* and *suffix* separated by a colon (`:`). The prefix is a term taken from the active context and is a short string identifying a particular IRI in a JSON-LD document. For example, the prefix `foaf` may be used as a shorthand for the Friend-of-a-Friend vocabulary, which is identified using the IRI `http://xmlns.com/foaf/0.1/`. A developer may append any of the FOAF vocabulary terms to the end of the prefix to specify a short-hand version of the IRI for the vocabulary term. For example, `foaf:name` would be expanded to the IRI `http://xmlns.com/foaf/0.1/name`.

EXAMPLE 31: Prefix expansion

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/"
    ...
  },
  "@type": "foaf:Person",
  "foaf:name": "Dave Longley",
  ...
}
```

In the example above, `foaf:name` expands to the IRI `http://xmlns.com/foaf/0.1/name` and `foaf:Person` expands to `http://xmlns.com/foaf/0.1`

/Person.

Prefixes are expanded when the form of the value is a compact IRI represented as a prefix:suffix combination, the *prefix* matches a term defined within the active context, and the *suffix* does not begin with two slashes (//). The compact IRI is expanded by concatenating the IRI mapped to the *prefix* to the (possibly empty) *suffix*. If the *prefix* is not defined in the active context, or the suffix begins with two slashes (such as in http://example.com), the value is interpreted as IRI instead. If the prefix is an underscore (_), the value is interpreted as blank node identifier instead.

It's also possible to use compact IRIs within the context as shown in the following example:

EXAMPLE 32: Using vocabularies

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "foaf": "http://xmlns.com/foaf/0.1/",
    "foaf:homepage": { "@type": "@id" },
    "picture": { "@id": "foaf:depiction", "@type": "@id" }
  },
  "@id": "http://me.markus-lanthaler.com/",
  "@type": "foaf:Person",
  "foaf:name": "Markus Lanthaler",
  "foaf:homepage": "http://www.markus-lanthaler.com/",
  "picture": "http://twitter.com/account/profile_image/markuslanthaler"
}
```

When operating explicitly with the processing mode for *JSON-LD 1.0* compatibility, terms may be chosen as compact IRI prefixes when compacting only if a simple term definition is used where the value ends with a URI gen-delim character (e.g, /, # and others, see [RFC3986]).

In JSON-LD 1.1, terms may be chosen as compact IRI prefixes when compacting only if a simple term definition is used where the value ends with a URI gen-delim character, or if their expanded term definition contains a @prefix entry with the value true.

NOTE

The term selection behavior for 1.0 processors was changed as a result of an errata against *JSON-LD 1.0* reported here. This does not affect the behavior of processing existing JSON-LD documents, but creates a slight change when compacting documents using Compact IRIs.

The behavior when compacting can be illustrated by considering the following input document in expanded form:

EXAMPLE 33: Expanded document used to illustrate compact IRI creation

Input

```
[{
  "http://example.com/vocab/property": [{"@value": "property"}
  "http://example.com/vocab/propertyOne": [{"@value": "propert
}]
```

Using the following context in the 1.0 processing mode will now select the term *vocab* rather than *property*, even though the IRI associated with *property* captures more of the original IRI.

EXAMPLE 34: Compact IRI generation context (1.0)

Context

```
{
  "@context": {
    "vocab": "http://example.com/vocab/",
    "property": "http://example.com/vocab/property"
  }
}
```

Compacting using the previous context with the above expanded input document results in the following compacted result:

EXAMPLE 35: Compact IRI generation term selection (1.0)

Compacted (Result) | Statements | Turtle | Open in playground

Result

```
{
  "@context": {
    "vocab": "http://example.com/vocab/",
    "property": "http://example.com/vocab/property"
  },
  "property": "property",
  "vocab:propertyOne": "propertyOne"
}
```

In the original [JSON-LD10], the term selection algorithm would have selected *property*, creating the Compact IRI *property:One*. The original behavior can be made explicit using `@prefix`:

EXAMPLE 36: Compact IRI generation context (1.1)

Context

```
{
  "@context": {
    "@version": 1.1,
    "vocab": "http://example.com/vocab/",
    "property": {
      "@id": "http://example.com/vocab/property",
      "@prefix": true
    }
  }
}
```

EXAMPLE 37: Compact IRI generation term selection (1.1)

Compacted (Input) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "vocab": "http://example.com/vocab/",
    "property": {
      "@id": "http://example.com/vocab/property",
      "@prefix": true
    }
  },
  "property": "property",
  "property:One": "propertyOne"
}
```

In this case, the *property* term would not normally be usable as a prefix, both because it is defined with an expanded term definition, and because its `@id` does not end in a gen-delim character. Adding `"@prefix": true` allows it to be used as the prefix portion of the compact IRI *property:One*.

§ 4.1.6 Aliasing Keywords

This section is non-normative.

Each of the JSON-LD keywords, except for `@context`, may be aliased to application-specific keywords. This feature allows legacy JSON content to be utilized by JSON-LD by re-using JSON keys that already exist in legacy documents. This feature also allows developers to design domain-specific implementations using only the JSON-LD context.

EXAMPLE 38: Aliasing keywords

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "url": "@id",
    "a": "@type",
    "name": "http://xmlns.com/foaf/0.1/name"
  },
  "url": "http://example.com/about#gregg",
  "a": "http://xmlns.com/foaf/0.1/Person",
  "name": "Gregg Kellogg"
}
```

In the example above, the `@id` and `@type` keywords have been given the aliases **url** and **a**, respectively.

Other than for `@type`, properties of expanded term definitions where the term is a keyword result in an error. Unless the processing mode is set to `json-ld-1.0`, there is also an exception for `@type`; see § 4.3.3 Using `@set` with `@type` for further details and usage examples.

Unless the processing mode is set to `json-ld-1.0`, aliases of keywords are either simple term definitions, where the value is a keyword, or a expanded term definitions with an `@id` entry and optionally an `@protected` entry; no other entries are allowed. There is also an exception for aliases of `@type`, as indicated above. See § 4.1.11 Protected Term Definitions for further details of using `@protected`.

Since keywords cannot be redefined, they can also not be aliased to other keywords.

NOTE

Aliased keywords may not be used within a context, itself.

See § 9.16 Keywords for a normative definition of all keywords.

§ 4.1.7 IRI Expansion within a Context

This section is non-normative.

In general, normal IRI expansion rules apply anywhere an IRI is expected (see § 3.2 IRIs). Within a context definition, this can mean that terms defined within the context may also be used within that context as long as there are no circular dependencies. For example, it is common to use the `xsd` namespace when defining typed values:

EXAMPLE 39: IRI expansion within a context

```
{
  "@context": {
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "name": "http://xmlns.com/foaf/0.1/name",
    "age": {
      "@id": "http://xmlns.com/foaf/0.1/age",
      "@type": "xsd:integer"
    },
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  },
  ...
}
```

In this example, the `xsd` term is defined and used as a prefix for the `@type` coercion of the `age` property.

Terms may also be used when defining the IRI of another term:

EXAMPLE 40: Using a term to define the IRI of another term within a context

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "name": "foaf:name",
    "age": {
      "@id": "foaf:age",
      "@type": "xsd:integer"
    },
    "homepage": {
      "@id": "foaf:homepage",
      "@type": "@id"
    }
  },
  ...
}
```

Compact IRIs and IRIs may be used on the left-hand side of a term definition.

EXAMPLE 41: Using a compact IRI as a term

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "name": "foaf:name",
    "foaf:age": {
      "@id": "http://xmlns.com/foaf/0.1/age",
      "@type": "xsd:integer"
    },
    "foaf:homepage": {
      "@type": "@id"
    }
  },
  ...
}
```

In this example, the compact IRI form is used in two different ways. In the first approach, `foaf:age` declares both the IRI for the term (using short-form) as well as the `@type` associated with the term. In the second approach, only the `@type` associated with the term is specified. The full IRI for `foaf:homepage`

is determined by looking up the `foaf` prefix in the context.

⚠ Warning

If a compact IRI is used as a term, it must expand to the value that compact IRI would have on its own when expanded. This represents a change to the original 1.0 algorithm to prevent terms from expanding to a different IRI, which could lead to undesired results.

EXAMPLE 42: Illegal Aliasing of a compact IRI to a different IRI

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "name": "foaf:name",
    "foaf:age": {
      "@id": "http://xmlns.com/foaf/0.1/age",
      "@type": "xsd:integer"
    },
    "foaf:homepage": {
     "@id": "http://schema.org/url",
     "@type": "@id"
    }
  },
  ...
}
```

IRIs may also be used in the key position in a context:

EXAMPLE 43: Associating context definitions with IRIs

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "name": "foaf:name",
    "foaf:age": {
      "@id": "http://xmlns.com/foaf/0.1/age",
      "@type": "xsd:integer"
    },
    "http://xmlns.com/foaf/0.1/homepage": {
      "@type": "@id"
    }
  },
  ...
}
```

In order for the IRI to match above, the IRI needs to be used in the JSON-LD document. Also note that `foaf:homepage` will not use the `{ "@type": "@id" }` declaration because `foaf:homepage` is not the same as `http://xmlns.com/foaf/0.1/homepage`. That is, terms are looked up in a context using direct string comparison before the prefix lookup mechanism is applied.

⚠ Warning

Neither an IRI reference nor a compact IRI may expand to some other unrelated IRI. This represents a change to the original 1.0 algorithm which allowed this behavior but discouraged it.

The only other exception for using terms in the context is that circular definitions are not allowed. That is, a definition of *term1* cannot depend on the definition of *term2* if *term2* also depends on *term1*. For example, the following context definition is illegal:

EXAMPLE 44: Illegal circular definition of terms within a context

```
{
  "@context": {
    "term1": "term2:foo",
    "term2": "term1:bar"
  },
  ...
}
```

§ 4.1.8 Scoped Contexts

This section is non-normative.

An expanded term definition can include a `@context` property, which defines a context (a scoped context) for values of properties defined using that term. When used for a property, this is called a property-scoped context. This allows values to use term definitions, the base IRI, vocabulary mappings or the default language which are different from the node object they are contained in, as if the context was specified within the value itself.

EXAMPLE 45: Defining an @context within a term definition

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "name": "http://schema.org/name",
    "interest": {
      "@id": "http://xmlns.com/foaf/0.1/interest",
      "@context": {"@vocab": "http://xmlns.com/foaf/0.1/"}
    }
  },
  "name": "Manu Sporny",
  "interest": {
    "@id": "https://www.w3.org/TR/json-ld11/",
    "name": "JSON-LD",
    "topic": "Linking Data"
  }
}
```

In this case, the social profile is defined using the schema.org vocabulary, but

interest is imported from FOAF, and is used to define a node describing one of Manu's interests where those properties now come from the FOAF vocabulary.

Expanding this document, uses a combination of terms defined in the outer context, and those defined specifically for that term in a property-scoped context.

Scoping can also be performed using a term used as a value of `@type`:

EXAMPLE 46: Defining an @context within a term definition used on @type

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "name": "http://schema.org/name",
    "interest": "http://xmlns.com/foaf/0.1/interest",
    "Person": "http://schema.org/Person",
    "Document": {
      "@id": "http://xmlns.com/foaf/0.1/Document",
      "@context": {"@vocab": "http://xmlns.com/foaf/0.1/"}
    }
  },
  "@type": "Person",
  "name": "Manu Sporny",
  "interest": {
    "@id": "https://www.w3.org/TR/json-ld11/",
    "@type": "Document",
    "name": "JSON-LD",
    "topic": "Linking Data"
  }
}
```

Scoping on `@type` is useful when common properties are used to relate things of different types, where the vocabularies in use within different entities calls for different context scoping. For example, `hasPart`/`partOf` may be common terms used in a document, but mean different things depending on the context. A type-scoped context is only in effect for the node object on which the type is used; the previous in-scope contexts are placed back into effect when traversing into another node object. As described further in § 4.1.9 Context Propagation, this may be controlled using the `@propagate` keyword.

> NOTE
>
> Any property-scoped or local contexts that were introduced in the node object would still be in effect when traversing into another node object.

When expanding, each value of `@type` is considered (ordering them lexicographically) where that value is also a term in the active context having its own type-scoped context. If so, that the scoped context is applied to the active context.

> NOTE
>
> The values of `@type` are unordered, so if multiple types are listed, the order that type-scoped contexts are applied is based on lexicographical ordering.

For example, consider the following semantically equivalent examples. The first example, shows how properties and types can define their own scoped contexts, which are included when expanding.

EXAMPLE 47: Expansion using embedded and scoped contexts

```
{
  "@context": {
    "@version": 1.1,
    "@vocab": "http://example.com/vocab/",
    "property": {
      "@id": "http://example.com/vocab/property",
      "@context": {
        "term1": "http://example.com/vocab/term1"
          ↑ Scoped context for "property" defines term1
      }
    },
    "Type1": {
      "@id": "http://example.com/vocab/Type1",
      "@context": {
        "term3": "http://example.com/vocab/term3"
          ↑ Scoped context for "Type1" defines term3
      }
    },
    "Type2": {
      "@id": "http://example.com/vocab/Type2",
      "@context": {
        "term4": "http://example.com/vocab/term4"
          ↑ Scoped context for "Type2" defines term4
      }
    }
  },
  "property": {
    "@context": {
      "term2": "http://example.com/vocab/term2"
          ↑ Embedded context defines term2
    },
    "@type": ["Type2", "Type1"],
    "term1": "a",
    "term2": "b",
    "term3": "c",
    "term4": "d"
  }
}
```

Contexts are processed depending on how they are defined. A property-scoped context is processed first, followed by any embedded context, followed lastly by the type-scoped contexts, in the appropriate order. The

previous example is logically equivalent to the following:

EXAMPLE 48: Expansion using embedded and scoped contexts (embedding equivalent)

```
{
  "@context": {
    "@vocab": "http://example.com/vocab/",
    "property": "http://example.com/vocab/property",
    "Type1": "http://example.com/vocab/Type1",
    "Type2": "http://example.com/vocab/Type2"
  },
  "property": {
    "@context": [{
        "term1": "http://example.com/vocab/term1"
         ↑ Previously scoped context for "property" defines term1
      }, {
        "term2": "http://example.com/vocab/term2"
         ↑ Embedded context defines term2
      }, {
        "term3": "http://example.com/vocab/term3"
         ↑ Previously scoped context for "Type1" defines term3
      }, {
      "term4": "http://example.com/vocab/term4"
         ↑ Previously scoped context for "Type2" defines term4
    }],
    "@type": ["Type2", "Type1"],
    "term1": "a",
    "term2": "b",
    "term3": "c",
    "term4": "d"
  }
}
```

NOTE

If a term defines a scoped context, and then that term is later redefined, the association of the context defined in the earlier expanded term definition is lost within the scope of that redefinition. This is consistent with term definitions of a term overriding previous term definitions from earlier less deeply nested definitions, as discussed in § 4.1 Advanced Context Usage.

NOTE

Scoped Contexts are a new feature in JSON-LD 1.1.

§ 4.1.9 Context Propagation

This section is non-normative.

Once introduced, contexts remain in effect until a subsequent context removes it by setting `@context` to `null`, or by redefining terms, with the exception of type-scoped contexts, which limit the effect of that context until the next node object is entered. This behavior can be changed using the `@propagate` keyword.

The following example illustrates how terms defined in a context with `@propagate` set to `false` are effectively removed when descending into new node object.

EXAMPLE 49: Marking a context to not propagate

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "term": {
      "@id": "http://example.org/original",
      "@context": {
        "@propagate": false,
         ↑ Scoped context only lasts in one node-object
        "term": "http://example.org/non-propagated-term"
      }
    }
  },
  "term": {
   ↑ This term is the original
    "term": {
     ↑ This term is from the scoped context
      "term": "This term is from the first context"
       ↑ This term is the original again
    }
  }
}
```

> NOTE
>
> Contexts included within an array must all have the same value for `@propagate` due to the way that rollback is defined in *JSON-LD 1.1 Processing Algorithms and API*.

§ 4.1.10 Imported Contexts

This section is non-normative.

JSON-LD 1.0 included mechanisms for modifying the context that is in effect. This included the capability to load and process a remote context and then apply further changes to it via new contexts.

However, with the introduction of JSON-LD 1.1, it is also desirable to be able to load a remote context, in particular an existing JSON-LD 1.0 context, and apply JSON-LD 1.1 features to it prior to processing.

By using the `@import` keyword in a context, another remote context, referred to as an imported context, can be loaded and modified prior to processing. The modifications are expressed in the context that includes the `@import` keyword, referred to as the wrapping context. Once an imported context is loaded, the contents of the wrapping context are merged into it prior to processing. The merge operation will cause each key-value pair in the wrapping context to be added to the loaded imported context, with the wrapping context key-value pairs taking precedence.

By enabling existing contexts to be reused and edited inline prior to processing, context-wide keywords can be applied to adjust all term definitions in the imported context. Similarly, term definitions can be replaced prior to processing, enabling adjustments that, for instance, ensure term definitions match previously protected terms or that they include additional type coercion information.

The following examples illustrate how `@import` can be used to express a type-scoped context that loads an imported context and sets `@propagate` to `true`, as a technique for making other similar modifications.

Suppose there was a context that could be referenced remotely via the URL `https://json-ld.org/contexts/remote-context.jsonld`:

EXAMPLE 50: A remote context to be imported in a type-scoped context

```
{
  "@context": {
    "Type1": "http://example.com/vocab/Type1",
    "Type2": "http://example.com/vocab/Type2",
    "term1": "http://example.com/vocab#term1",
    "term2": "http://example.com/vocab#term2",
    ...
  }
}
```

A wrapping context could be used to source it and modify it:

EXAMPLE 51: Sourcing a context in a type-scoped context and setting it to propagate

```
{
  "@context": {
    "@version": 1.1,
    "MyType": {
      "@id": "http://example.com/vocab#MyType",
      "@context": {
        "@version": 1.1,
        "@import": "https://json-ld.org/contexts/remote-context.jsonld"
        "@propagate": true
      }
    }
  }
}
```

The effect would be the same as if the entire imported context had been copied into the type-scoped context:

EXAMPLE 52: Result of sourcing a context in a type-scoped context and setting it to propagate

```
{
  "@context": {
    "@version": 1.1,
    "MyType": {
      "@id": "http://example.com/vocab#MyType",
      "@context": {
        "@version": 1.1,
        "Type1": "http://example.com/vocab/Type1",
        "Type2": "http://example.com/vocab/Type2",
        "term1": "http://example.com/vocab#term1",
        "term2": "http://example.com/vocab#term2",
        ...
        "@propagate": true
      }
    }
  }
}
```

Similarly, the wrapping context may replace term definitions or set other context-wide keywords that may affect how the imported context term definitions will be processed:

EXAMPLE 53: Sourcing a context to modify @vocab and a term definition

```
{
  "@context": {
    "@version": 1.1,
    "@import": "https://json-ld.org/contexts/remote-context.jsonld",
    "@vocab": "http://example.org/vocab#",
     ↑ This will replace any previous @vocab definition prior to proces
    "term1": {
      "@id": "http://example.org/vocab#term1",
      "@type": "http://www.w3.org/2001/XMLSchema#integer"
    }
     ↑ This will replace the old term1 definition prior to processing i
  }
}
```

Again, the effect would be the same as if the entire imported context had been copied into the context:

EXAMPLE 54: Result of sourcing a context to modify @vocab and a term definition

```
{
  "@context": {
    "@version": 1.1,
    "Type1": "http://example.com/vocab/Type1",
    "Type2": "http://example.com/vocab/Type2",
    "term1": {
      "@id": "http://example.org/vocab#term1",
      "@type": "http://www.w3.org/2001/XMLSchema#integer"
    },
     ↑ Note term1 has been replaced prior to processing
    "term2": "http://example.com/vocab#term2",
    ...,
    "@vocab": "http://example.org/vocab#"
  }
}
```

The result of loading imported contexts must be context definition, not an IRI or an array. Additionally, the imported context cannot include an `@import` entry.

§ 4.1.11 Protected Term Definitions

This section is non-normative.

JSON-LD is used in many specifications as the specified data format. However, there is also a desire to allow some JSON-LD contents to be processed as plain JSON, without using any of the JSON-LD algorithms. Because JSON-LD is very flexible, some terms from the original format may be locally overridden through the use of embedded contexts, and take a different meaning for JSON-LD based implementations. On the other hand, "plain JSON" implementations may not be able to interpret these embedded contexts, and hence will still interpret those terms with their original meaning. To prevent this divergence of interpretation, JSON-LD 1.1 allows term definitions to be *protected*.

A ***protected term definition*** is a term definition with an entry `@protected` set to `true`. It generally prevents further contexts from overriding this term definition, either through a new definition of the same term, or through clearing the context with `"@context": null`. Such attempts will raise an error

and abort the processing (except in some specific situations described below).

EXAMPLE 55: A protected term definition can generally not be overridden

```
{
  "@context": [
    {
      "@version": 1.1,
      "Person": "http://xmlns.com/foaf/0.1/Person",
      "knows": "http://xmlns.com/foaf/0.1/knows",
      "name": {
        "@id": "http://xmlns.com/foaf/0.1/name",
        "@protected": true
      }
    },
    {
      – this attempt will fail with an error
      "name": "http://schema.org/name"
    }
  ],
  "@type": "Person",
  "name": "Manu Sporny",
  "knows": {
    "@context": [
      – this attempt would also fail with an error
      null,
      "http://schema.org/"
    ],
    "name": "Gregg Kellogg"
  }
}
```

When all or most term definitions of a context need to be protected, it is possible to add an entry `@protected` set to `true` to the context itself. It has the same effect as protecting each of its term definitions individually. Exceptions can be made by adding an entry `@protected` set to `false` in some term definitions.

EXAMPLE 56: A protected @context with an exception

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": [
    {
      "@version": 1.1,
      "@protected": true,
      "name": "http://schema.org/name",
      "member": "http://schema.org/member",
      "Person": {
        "@id": "http://schema.org/Person",
        "@protected": false
      }
    }
  ],
  "name": "Digital Bazaar",
  "member": {
    "@context": {
      – name *is* protected, so the following would fail with an error
      –   "name": "http://xmlns.com/foaf/0.1/Person",
      – Person is *not* protected, and can be overridden
      "Person": "http://xmlns.com/foaf/0.1/Person"
    },
    "@type": "Person",
    "name": "Manu Sporny"
  }
}
```

While protected terms can in general not be overridden, there are two exceptions to this rule. The first exception is that a context is allowed to redefine a protected term if the new definition is identical to the protected term definition (modulo the `@protected` flag). The rationale is that the new definition does not violate the protection, as it does not change the semantics of the protected term. This is useful for widespread term definitions, such as aliasing `@type` to `type`, which may occur (including in a protected form) in several contexts.

EXAMPLE 57: Overriding permitted if both definitions are identical

Original | Expanded | Statements | Turtle | Open in playground

```
{
  "@context": [
    {
      "@version": 1.1,
      "@protected": true,
      "id": "@id",
      "type": "@type",
      "Organization": "http://example.org/orga/Organization",
      "member": {
        "@id": "http://example.org/orga/member",
        "@type": "@id"
      }
    },
    {
      "id": "@id",
      "type": "@type",
      – Those "redefinitions" do not raise an error.
      – Note however that the terms are still protected
      "Person": "http://schema.org/Person",
      "name": "http://schema.org/name"
    }
  ],
  "id": "https://digitalbazaar.com/",
  "type": "Organization",
  "member" : {
    "id": "http://manu.sporny.org/about#manu",
    "type": "Person",
    "name": "Manu Sporny"
  }
}
```

The second exception is that a property-scoped context is not affected by protection, and can therefore override protected terms, either with a new term definition, or by clearing the context with `"@context": null`.

The rationale is that "plain JSON" implementations, relying on a given specification, will only traverse properties defined by that specification. Scoped contexts belonging to the specified properties are part of the specification, so the "plain JSON" implementations are expected to be aware of the change of semantics they induce. Scoped contexts belonging to other properties apply to parts of the document that "plain JSON" implementations

will ignore. In both cases, there is therefore no risk of diverging interpretations between JSON-LD-aware implementations and "plain JSON" implementations, so overriding is permitted.

EXAMPLE 58: overriding permitted in property scoped context

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": [
    {
      – This context reflects the specification used by "plain JSON" im
      "@version": 1.1,
      "@protected": true,
      "Organization": "http://schema.org/Organization",
      "name": "http://schema.org/name",
      "employee": {
        "@id": "http://schema.org/employee",
        "@context": {
          "@protected": true,
          "name": "http://schema.org/familyName"
        }
        ↑ overrides the definition of "name"
      }
    },
    {
      – This context extends the previous one,
      – only JSON-LD-aware implementations are expected to use it
      "location": {
        "@id": "http://xmlns.com/foaf/0.1/based_near",
        "@context": [
          null,
          ↑ clears the context entirely, including all protected terms
          { "@vocab": "http://xmlns.com/foaf/0.1/" }
        ]
      }
    }
  ],
  "@type": "Organization",
  "name": "Digital Bazaar",
  "employee" : {
    "name": "Sporny",
    "location": {"name": "Blacksburg, Virginia"}
  }
}
```

NOTE

By preventing terms from being overridden, protection also prevents any adaptation of a term (e.g., defining a more precise datatype, restricting the term's use to lists, etc.). This kind of adaptation is frequent with some general purpose contexts, for which protection would therefore hinder their usability. As a consequence, context publishers should use this feature with care.

NOTE

Protected term definitions are a new feature in JSON-LD 1.1.

§ 4.2 Describing Values

This section is non-normative.

Values are leaf nodes in a graph associated with scalar values such as strings, dates, times, and other such atomic values.

§ 4.2.1 Typed Values

This section is non-normative.

A value with an associated type, also known as a typed value, is indicated by associating a value with an IRI which indicates the value's type. Typed values may be expressed in JSON-LD in three ways:

1. By utilizing the `@type` keyword when defining a term within an `@context` section.
2. By utilizing a value object.
3. By using a native JSON type such as number, `true`, or `false`.

The first example uses the `@type` keyword to associate a type with a particular term in the `@context`:

EXAMPLE 59: Expanded term definition with type coercion

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "modified": {
      "@id": "http://purl.org/dc/terms/modified",
      "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
    }
  },
  ...
  "@id": "http://example.com/docs/1",
  "modified": "2010-05-29T14:17:39+02:00",
  ...
}
```

The *modified* key's value above is automatically interpreted as a *dateTime* value because of the information specified in the `@context`. The example tabs show how a JSON-LD processor will interpret the data.

The second example uses the expanded form of setting the type information in the body of a JSON-LD document:

EXAMPLE 60: Expanded value with type

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "modified": {
      "@id": "http://purl.org/dc/terms/modified"
    }
  },
  ...
  "modified": {
    "@value": "2010-05-29T14:17:39+02:00",
    "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
  }
  ...
}
```

Both examples above would generate the value `2010-05-29T14:17:39+02:00` with the type `http://www.w3.org/2001/XMLSchema#dateTime`. Note that it is also possible to use a term or a compact IRI to express the value of a type.

NOTE

The `@type` keyword is also used to associate a type with a node. The concept of a node type and a value type are distinct. For more on adding types to nodes, see § 3.5 Specifying the Type.

NOTE

When expanding, an `@type` defined within a term definition can be associated with a string value to create an expanded value object, which is described in § 4.2.3 Type Coercion. Type coercion only takes place on string values, not for values which are maps, such as node objects and value objects in their expanded form.

A ***node type*** specifies the type of thing that is being described, like a person, place, event, or web page. A ***value type*** specifies the data type of a particular value, such as an integer, a floating point number, or a date.

EXAMPLE 61: Example demonstrating the context-sensitivity for @type

```
{
  ...
  "@id": "http://example.org/posts#TripToWestVirginia",
  "@type": "http://schema.org/BlogPosting",  ← This is a node type
  "http://purl.org/dc/terms/modified": {
    "@value": "2010-05-29T14:17:39+02:00",
    "@type": "http://www.w3.org/2001/XMLSchema#dateTime"  ← This is a v
  }
  ...
}
```

The first use of `@type` associates a node type (`http://schema.org/BlogPosting`) with the node, which is expressed using the `@id` keyword. The second use of `@type` associates a value type (`http://www.w3.org/2001/XMLSchema#dateTime`) with the value expressed using the `@value` keyword. As a general rule, when `@value` and `@type` are used in the same map, the `@type` keyword is expressing a value type. Otherwise, the `@type` keyword is expressing a node type. The example above expresses the following data:

EXAMPLE 62: Example demonstrating the context-sensitivity for @type (statements)

Compacted (Input) Turtle Open in playground

Subject	Property	Value
http://example.org/posts#TripToWestVirginia	rdf:type	schema:BlogPosting
http://example.org/posts#TripToWestVirginia	dcterms:modified	2010-05-29T14:17:39+02:0

§ 4.2.2 JSON Literals

This section is non-normative.

At times, it is useful to include JSON within JSON-LD that is not interpreted as JSON-LD. Generally, a JSON-LD processor will ignore properties which don't map to IRIs, but this causes them to be excluded when performing various algorithmic transformations. But, when the data that is being described is, itself, JSON, it's important that it survive algorithmic transformations.

⚠ Warning

JSON-LD is intended to allow native JSON to be interpreted through the use of a context. The use of JSON literals creates blobs of data which are not available for interpretation. It is for use only in the rare cases that JSON cannot be represented as JSON-LD.

When a term is defined with `@type` set to `@json`, a JSON-LD processor will treat the value as a JSON literal, rather than interpreting it further as JSON-LD. In the expanded document form, such JSON will become the value of `@value` within a value object having `"@type": "@json"`.

When transformed into RDF, the JSON literal will have a lexical form based on a specific serialization of the JSON, as described in Compaction algorithm of [JSON-LD11-API] and the JSON datatype.

The following example shows an example of a JSON Literal contained as the value of a property. Note that the RDF results use a canonicalized form of the JSON to ensure interoperability between different processors. JSON

canonicalization is described in Data Round Tripping in [JSON-LD11-API].

EXAMPLE 63: JSON Literal

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "e": {"@id": "http://example.com/vocab/json", "@type": "@json"}
  },
  "e": [
    56.0,
    {
      "d": true,
      "10": null,
      "1": [ ]
    }
  ]
}
```

NOTE

Generally, when a JSON-LD processor encounters `null`, the associated entry or value is removed. However, `null` is a valid JSON token; when used as the value of a JSON literal, a `null` value will be preserved.

§ 4.2.3 Type Coercion

This section is non-normative.

JSON-LD supports the coercion of string values to particular data types. Type ***coercion*** allows someone deploying JSON-LD to use string property values and have those values be interpreted as typed values by associating an IRI with the value in the expanded value object representation. Using type coercion, string value representation can be used without requiring the data type to be specified explicitly with each piece of data.

Type coercion is specified within an expanded term definition using the `@type` key. The value of this key expands to an IRI. Alternatively, the keyword `@id` or `@vocab` may be used as value to indicate that within the body of a JSON-LD document, a string value of a term coerced to `@id` or `@vocab` is to be

interpreted as an IRI. The difference between `@id` and `@vocab` is how values are expanded to IRIs. `@vocab` first tries to expand the value by interpreting it as term. If no matching term is found in the active context, it tries to expand it as an IRI or a compact IRI if there's a colon in the value; otherwise, it will expand the value using the active context's vocabulary mapping, if present. Values coerced to `@id` in contrast are expanded as an IRI or a compact IRI if a colon is present; otherwise, they are interpreted as relative IRI references.

> NOTE
>
> The ability to coerce a value using a term definition is distinct from setting one or more types on a node object, as the former does not result in new data being added to the graph, while the later manages node types through adding additional relationships to the graph.

Terms or compact IRIs used as the value of a `@type` key may be defined within the same context. This means that one may specify a term like `xsd` and then use `xsd:integer` within the same context definition.

The example below demonstrates how a JSON-LD author can coerce values to typed values and IRIs.

EXAMPLE 64: Expanded term definition with types

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "name": "http://xmlns.com/foaf/0.1/name",
    "age": {
      "@id": "http://xmlns.com/foaf/0.1/age",
      "@type": "xsd:integer"
    },
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  },
  "@id": "http://example.com/people#john",
  "name": "John Smith",
  "age": "41",
  "homepage": [
    "http://personal.example.org/",
    "http://work.example.com/jsmith/"
  ]
}
```

It is important to note that terms are only used in expansion for vocabulary-relative positions, such as for keys and values of map entries. Values of `@id` are considered to be document-relative, and do not use term definitions for expansion. For example, consider the following:

EXAMPLE 65: Term expansion for values, not identifiers

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@base": "http://example1.com/",
    "@vocab": "http://example2.com/",
    "knows": {"@type": "@vocab"}
  },
  "@id": "fred",
  "knows": [
    {"@id": "barney", "mnemonic": "the sidekick"},
    "barney"
  ]
}
```

The unexpected result is that "barney" expands to both `http://example1.com/barney` and `http://example2.com/barney`, depending where it is encountered. String values interpreted as IRIs because of the associated term definitions are typically considered to be document-relative. In some cases, it makes sense to interpret these relative to the vocabulary, prescribed using `"@type": "@vocab"` in the term definition, though this can lead to unexpected consequences such as these.

In the previous example, "barney" appears twice, once as the value of `@id`, which is always interpreted as a document-relative IRI, and once as the value of "fred", which is defined to be vocabulary-relative, thus the different expanded values.

For more on this see § 4.1.2 Default Vocabulary.

A variation on the previous example using `"@type": "@id"` instead of `@vocab` illustrates the behavior of interpreting "barney" relative to the document:

EXAMPLE 66: Terms not expanded when document-relative

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@base": "http://example1.com/",
    "@vocab": "http://example2.com/",
    "knows": {"@type": "@id"}
  },
  "@id": "fred",
  "knows": [
    {"@id": "barney", "mnemonic": "the sidekick"},
    "barney"
  ]
}
```

NOTE

The triple `ex1:fred ex2:knows ex1:barney .` is emitted twice, but exists only once in an output dataset, as it is a duplicate triple.

Terms may also be defined using IRIs or compact IRIs. This allows coercion rules to be applied to keys which are not represented as a simple term. For example:

EXAMPLE 67: Term definitions using IRIs and compact IRIs

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "foaf": "http://xmlns.com/foaf/0.1/",
    "foaf:age": {
      "@id": "http://xmlns.com/foaf/0.1/age",
      "@type": "xsd:integer"
    },
    "http://xmlns.com/foaf/0.1/homepage": {
      "@type": "@id"
    }
  },
  "foaf:name": "John Smith",
  "foaf:age": "41",
  "http://xmlns.com/foaf/0.1/homepage": [
    "http://personal.example.org/",
    "http://work.example.com/jsmith/"
  ]
}
```

In this case the `@id` definition in the term definition is optional. If it does exist, the IRI or compact IRI representing the term will always be expanded to IRI defined by the `@id` key—regardless of whether a prefix is defined or not.

Type coercion is always performed using the unexpanded value of the key. In the example above, that means that type coercion is done looking for `foaf:age` in the active context and not for the corresponding, expanded IRI `http://xmlns.com/foaf/0.1/age`.

NOTE

Keys in the context are treated as terms for the purpose of expansion and value coercion. At times, this may result in multiple representations for the same expanded IRI. For example, one could specify that `dog` and `cat` both expanded to `http://example.com/vocab#animal`. Doing this could be useful for establishing different type coercion or language specification rules.

§ 4.2.4 String Internationalization

This section is non-normative.

At times, it is important to annotate a string with its language. In JSON-LD this is possible in a variety of ways. First, it is possible to define a default language for a JSON-LD document by setting the `@language` key in the context:

EXAMPLE 68: Setting the default language of a JSON-LD document

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "name": "http://example.org/name",
    "occupation": "http://example.org/occupation",
    ...
    "@language": "ja"
  },
  "name": "花澄",
  "occupation": "科学者"
}
```

The example above would associate the `ja` language tag with the two strings *花澄* and *科学者* Languages tags are defined in [BCP47]. The default language applies to all string values that are not type coerced.

To clear the default language for a subtree, `@language` can be set to `null` in an intervening context, such as a scoped context as follows:

EXAMPLE 69: Clearing default language

```
{
  "@context": {
    ...
    "@version": 1.1,
    "@vocab": "http://example.com/",
    "@language": "ja",
    "details": {
      "@context": {
        "@language": null
      }
    }
  },
  "name": "花澄",
  "details": {"occupation": "Ninja"}
}
```

Second, it is possible to associate a language with a specific term using an expanded term definition:

EXAMPLE 70: Expanded term definition with language

```
{
  "@context": {
    ...
    "ex": "http://example.com/vocab/",
    "@language": "ja",
    "name": { "@id": "ex:name", "@language": null },
    "occupation": { "@id": "ex:occupation" },
    "occupation_en": { "@id": "ex:occupation", "@language": "en" },
    "occupation_cs": { "@id": "ex:occupation", "@language": "cs" }
  },
  "name": "Yagyū Muneyoshi",
  "occupation": "忍者",
  "occupation_en": "Ninja",
  "occupation_cs": "Nindža",
  ...
}
```

The example above would associate *忍者* with the specified default language tag `ja`, *Ninja* with the language tag `en`, and *Nindža* with the language tag `cs`. The value of `name`, *Yagyū Muneyoshi* wouldn't be associated with any language tag since `@language` was reset to null in the expanded term

definition.

> NOTE
>
> Language associations are only applied to plain strings. Typed values or values that are subject to type coercion are not language tagged.

Just as in the example above, systems often need to express the value of a property in multiple languages. Typically, such systems also try to ensure that developers have a programmatically easy way to navigate the data structures for the language-specific data. In this case, language maps may be utilized.

EXAMPLE 71: Language map expressing a property in three languages

```
{
  "@context": {
    ...
    "occupation": { "@id": "ex:occupation", "@container": "@language" }
  },
  "name": "Yagyū Muneyoshi",
  "occupation": {
    "ja": "忍者",
    "en": "Ninja",
    "cs": "Nindža"
  }
  ...
}
```

The example above expresses exactly the same information as the previous example but consolidates all values in a single property. To access the value in a specific language in a programming language supporting dot-notation accessors for object properties, a developer may use the `property.language` pattern (when languages are limited to the primary language sub-tag, and do not depend on other sub-tags, such as `"en-us"`). For example, to access the occupation in English, a developer would use the following code snippet: `obj.occupation.en`.

Third, it is possible to override the default language by using a value object:

EXAMPLE 72: Overriding default language using an expanded value

```
{
  "@context": {
    ...
    "@language": "ja"
  },
  "name": "花澄",
  "occupation": {
    "@value": "Scientist",
    "@language": "en"
  }
}
```

This makes it possible to specify a plain string by omitting the `@language` tag or setting it to `null` when expressing it using a value object:

EXAMPLE 73: Removing language information using an expanded value

```
{
  "@context": {
    ...
    "@language": "ja"
  },
  "name": {
    "@value": "Frank"
  },
  "occupation": {
    "@value": "Ninja",
    "@language": "en"
  },
  "speciality": "手裏剣"
}
```

See § 9.8 Language Maps for a description of using language maps to set the language of mapped values.

§ 4.2.4.1 Base Direction

This section is non-normative.

It is also possible to annotate a string, or language-tagged string, with its base direction. As with language, it is possible to define a default base

direction for a JSON-LD document by setting the `@direction` key in the context:

EXAMPLE 74: Setting the default base direction of a JSON-LD document

Compacted (Input) | Expanded (Result) | Statements | Turtle (drops direction) | Turtle (with datatype) | Turtle (with bnode structure) | Open in playground

Input

```
{
  "@context": {
    "title": "http://example.org/title",
    "publisher": "http://example.org/publisher",
    ...
    "@language": "ar-EG",
    "@direction": "rtl"
  },
  "title": "HTML و CSS: تصميم و إنشاء مواقع الويب",
  "publisher": "مكتبة"
}
```

The example above would associate the `ar-EG` language tag and "rtl" base direction with the two strings *HTML و CSS: تصميم و إنشاء مواقع الويب* and *مكتبة*. The default base direction applies to all string values that are not type coerced.

To clear the default base direction for a subtree, `@direction` can be set to `null` in an intervening context, such as a scoped context as follows:

EXAMPLE 75: Clearing default base direction

```
{
  "@context": {
    ...
    "@version": 1.1,
    "@vocab": "http://example.com/",
    "@language": "ar-EG",
    "@direction": "rtl",
    "details": {
      "@context": {
        "@direction": null
      }
    }
  },
  "title": "HTML و CSS: تصميم و إنشاء مواقع الويب",
  "details": {"genre": "Technical Publication"}
}
```

Second, it is possible to associate a base direction with a specific term using an expanded term definition:

EXAMPLE 76: Expanded term definition with language and direction

```
{
  "@context": {
    ...
    "@version": 1.1,
    "@language": "ar-EG",
    "@direction": "rtl",
    "ex": "http://example.com/vocab/",
    "publisher": { "@id": "ex:publisher", "@direction": null },
    "title": { "@id": "ex:title" },
    "title_en": { "@id": "ex:title", "@language": "en", "@direction": "
  },
  "publisher": "مكتبة",
  "title": "HTML و CSS: تصميم و إنشاء مواقع الويب",
  "title_en": "HTML and CSS: Design and Build Websites",
  ...
}
```

The example above would create three properties:

Subject	Property	Value	Language	Direction
_:b0	http://example.com/vocab/publisher	مكتبة	ar-EG	
_:b0	http://example.com/vocab/title	HTML و CSS: تصميم و إنشاء مواقع الويب	ar-EG	rtl
_:b0	http://example.com/vocab/title	HTML and CSS: Design and Build Websites	en	ltr

NOTE

Base direction associations are only applied to plain strings and language-tagged strings. Typed values or values that are subject to type coercion are not given a base direction.

Third, it is possible to override the default base direction by using a value object:

EXAMPLE 77: Overriding default language and default base direction using an expanded value

```
{
  "@context": {
    ...
    "@language": "ar-EG",
    "@direction": "rtl"
  },
  "title": "تصميم و إنشاء مواقع الويب: CSS و HTML",
  "author": {
    "@value": "Jon Duckett",
    "@language": "en",
    "@direction": null
  }
}
```

See *Strings on the Web: Language and Direction Metadata* [string-meta] for a deeper discussion of base direction.

§ 4.3 Value Ordering

This section is non-normative.

A JSON-LD author can express multiple values in a compact way by using arrays. Since graphs do not describe ordering for links between nodes, arrays in JSON-LD do not convey any ordering of the contained elements by default. This is exactly the opposite from regular JSON arrays, which are ordered by default. For example, consider the following simple document:

EXAMPLE 78: Multiple values with no inherent order

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {"foaf": "http://xmlns.com/foaf/0.1/"},
  ...
  "@id": "http://example.org/people#joebob",
  "foaf:nick": [ "joe", "bob", "JB" ],
  ...
}
```

Multiple values may also be expressed using the expanded form:

EXAMPLE 79: Using an expanded form to set multiple values

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {"dcterms": "http://purl.org/dc/terms/"},
  "@id": "http://example.org/articles/8",
  "dcterms:title": [
    {
      "@value": "Das Kapital",
      "@language": "de"
    },
    {
      "@value": "Capital",
      "@language": "en"
    }
  ]
}
```

NOTE

The example shown above would generates statement, again with no inherent order.

Although multiple values of a property are typically of the same type, JSON-LD places no restriction on this, and a property may have values of different types:

EXAMPLE 80: Multiple array values of different types

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {"ex": "http://example.org/"},
  "@id": "http://example.org/people#michael",
  "ex:name": [
    "Michael",
    {"@value": "Mike"},
    {"@value": "Miguel", "@language": "es"},
    { "@id": "https://www.wikidata.org/wiki/Q4927524" },
    42
  ]
}
```

NOTE

When viewed as statements, the values have no inherent order.

§ 4.3.1 Lists

This section is non-normative.

As the notion of ordered collections is rather important in data modeling, it is useful to have specific language support. In JSON-LD, a list may be represented using the `@list` keyword as follows:

EXAMPLE 81: An ordered collection of values in JSON-LD

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {"foaf": "http://xmlns.com/foaf/0.1/"},
  ...
  "@id": "http://example.org/people#joebob",
  "foaf:nick": {
    "@list": [ "joe", "bob", "jaybee" ]
  },
  ...
}
```

This describes the use of this array as being ordered, and order is maintained when processing a document. If every use of a given multi-valued property is a list, this may be abbreviated by setting `@container` to `@list` in the context:

EXAMPLE 82: Specifying that a collection is ordered in the context

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    ...
    "nick": {
      "@id": "http://xmlns.com/foaf/0.1/nick",
      "@container": "@list"
    }
  },
  ...
  "@id": "http://example.org/people#joebob",
  "nick": [ "joe", "bob", "jaybee" ],
  ...
}
```

The implementation of lists in RDF depends on linking anonymous nodes together using the properties `rdf:first` and `rdf:rest`, with the end of the list defined as the resource `rdf:nil`, as the "statements" tab illustrates. This allows order to be represented within an unordered set of statements.

Both JSON-LD and Turtle provide shortcuts for representing ordered lists.

In JSON-LD 1.1, lists of lists, where the value of a list object, may itself be a list object, are fully supported.

Note that the `"@container": "@list"` definition recursively describes array values of lists as being, themselves, lists. For example, in *The GeoJSON Format* (see [RFC7946]), *coordinates* are an ordered list of *positions*, which are represented as an array of two or more numbers:

EXAMPLE 83: Coordinates expressed in GeoJSON

```
{
  "type": "Feature",
  "bbox": [-10.0, -10.0, 10.0, 10.0],
  "geometry": {
    "type": "Polygon",
    "coordinates": [
        [
            [-10.0, -10.0],
            [10.0, -10.0],
            [10.0, 10.0],
            [-10.0, -10.0]
        ]
    ]
  }
  //...
}
```

For these examples, it's important that values expressed within *bbox* and *coordinates* maintain their order, which requires the use of embedded list structures. In JSON-LD 1.1, we can express this using recursive lists, by simply adding the appropriate context definition:

EXAMPLE 84: Coordinates expressed in JSON-LD

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "https://purl.org/geojson/vocab#",
    "type": "@type",
    "bbox": {"@container": "@list"},
    "coordinates": {"@container": "@list"}
  },
  "type": "Feature",
  "bbox": [-10.0, -10.0, 10.0, 10.0],
  "geometry": {
    "type": "Polygon",
    "coordinates": [
        [
            [-10.0, -10.0],
            [10.0, -10.0],
            [10.0, 10.0],
            [-10.0, -10.0]
        ]
    ]
  }
  //...
}
```

Note that *coordinates* includes three levels of lists.

Values of terms associated with an `@list` container are always represented in the form of an array, even if there is just a single value or no value at all.

§ 4.3.2 Sets

This section is non-normative.

While `@list` is used to describe *ordered lists*, the `@set` keyword is used to describe *unordered sets*. The use of `@set` in the body of a JSON-LD document is optimized away when processing the document, as it is just syntactic sugar. However, `@set` is helpful when used within the context of a document. Values of terms associated with an `@set` container are always represented in the form of an array, even if there is just a single value that would otherwise be optimized to a non-array form in compact form (see § 5.2 Compacted

Document Form). This makes post-processing of JSON-LD documents easier as the data is always in array form, even if the array only contains a single value.

EXAMPLE 85: An unordered collection of values in JSON-LD

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {"foaf": "http://xmlns.com/foaf/0.1/"},
  ...
  "@id": "http://example.org/people#joebob",
  "foaf:nick": {
    "@set": [ "joe", "bob", "jaybee" ]
  },
  ...
}
```

This describes the use of this array as being unordered, and order may change when processing a document. By default, arrays of values are unordered, but this may be made explicit by setting `@container` to `@set` in the context:

EXAMPLE 86: Specifying that a collection is unordered in the context

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    ...
    "nick": {
      "@id": "http://xmlns.com/foaf/0.1/nick",
      "@container": "@set"
    }
  },
  ...
  "@id": "http://example.org/people#joebob",
  "nick": [ "joe", "bob", "jaybee" ],
  ...
}
```

Since JSON-LD 1.1, the `@set` keyword may be combined with other container specifications within an expanded term definition to similarly cause compacted values of indexes to be consistently represented using arrays. See § 4.6 Indexed Values for a further discussion.

§ 4.3.3 Using `@set` with `@type`

This section is non-normative.

Unless the processing mode is set to `json-ld-1.0`, `@type` may be used with an expanded term definition with `@container` set to `@set`; no other entries may be set within such an expanded term definition. This is used by the Compaction algorithm to ensure that the values of `@type` (or an alias) are always represented in an array.

EXAMPLE 87: Setting @container: @set on @type

```
{
  "@context": {
    "@version": 1.1,
    "@type": {"@container": "@set"}
  },
  "@type": ["http:/example.org/type"]
}
```

§ 4.4 Nested Properties

This section is non-normative.

Many JSON APIs separate properties from their entities using an intermediate object; in JSON-LD these are called nested properties. For example, a set of possible labels may be grouped under a common property:

EXAMPLE 88: Nested properties

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "skos": "http://www.w3.org/2004/02/skos/core#",
    "labels": "@nest",
    "main_label": {"@id": "skos:prefLabel"},
    "other_label": {"@id": "skos:altLabel"},
    "homepage": {"@id": "http://xmlns.com/foaf/0.1/homepage", "@type":
  },
  "@id": "http://example.org/myresource",
  "homepage": "http://example.org",
  "labels": {
     "main_label": "This is the main label for my resource",
     "other_label": "This is the other label"
  }
}
```

By defining *labels* using the keyword `@nest`, a JSON-LD processor will ignore the nesting created by using the *labels* property and process the contents as if it were declared directly within containing object. In this case, the *labels* property is semantically meaningless. Defining it as equivalent to `@nest` causes it to be ignored when expanding, making it equivalent to the following:

EXAMPLE 89: Nested properties folded into containing object

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "skos": "http://www.w3.org/2004/02/skos/core#",
    "main_label": {"@id": "skos:prefLabel"},
    "other_label": {"@id": "skos:altLabel"},
    "homepage": {"@id": "http://xmlns.com/foaf/0.1/homepage", "@type":
  },
  "@id": "http://example.org/myresource",
  "homepage": "http://example.org",
  "main_label": "This is the main label for my resource",
  "other_label": "This is the other label"
}
```

Similarly, term definitions may contain a `@nest` property referencing a term aliased to `@nest` which will cause such properties to be nested under that aliased term when compacting. In the example below, both `main_label` and `other_label` are defined with `"@nest": "labels"`, which will cause them to be serialized under `labels` when compacting.

EXAMPLE 90: Defining property nesting - Expanded Input

Input

```
[{
  "@id": "http://example.org/myresource",
  "http://xmlns.com/foaf/0.1/homepage": [
    {"@id": "http://example.org"}
  ],
  "http://www.w3.org/2004/02/skos/core#prefLabel": [
    {"@value": "This is the main label for my resource"}
  ],
  "http://www.w3.org/2004/02/skos/core#altLabel": [
    {"@value": "This is the other label"}
  ]
}]
```

EXAMPLE 91: Defining property nesting - Context

Context

```
{
  "@context": {
    "@version": 1.1,
    "skos": "http://www.w3.org/2004/02/skos/core#",
    "labels": "@nest",
    "main_label": {"@id": "skos:prefLabel", "@nest": "labels"},
    "other_label": {"@id": "skos:altLabel", "@nest": "labels"},
    "homepage": {"@id": "http://xmlns.com/foaf/0.1/homepage", "@type":
  }
}
```

EXAMPLE 92: Defining property nesting

Compacted (Result) | Statements | Turtle | Open in playground

Result

```
{
  "@context": {
    "@version": 1.1,
    "skos": "http://www.w3.org/2004/02/skos/core#",
    "labels": "@nest",
    "main_label": {"@id": "skos:prefLabel", "@nest": "labels"},
    "other_label": {"@id": "skos:altLabel", "@nest": "labels"},
    "homepage": {"@id": "http://xmlns.com/foaf/0.1/homepage", "@type":
  },
  "@id": "http://example.org/myresource",
  "homepage": "http://example.org",
  "labels": {
     "main_label": "This is the main label for my resource",
     "other_label": "This is the other label"
  }
}
```

NOTE

Nested properties are a new feature in JSON-LD 1.1.

§ 4.5 Embedding

This section is non-normative.

Embedding is a JSON-LD feature that allows an author to use node objects as property values. This is a commonly used mechanism for creating a parent-child relationship between two nodes.

Without embedding, node objects can be linked by referencing the identifier of another node object. For example:

EXAMPLE 93: Referencing node objects

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/",
    "knows": {"@type": "@id"}
  },
  "@graph": [{
    "name": "Manu Sporny",
    "@type": "Person",
    "knows": "https://greggkellogg.net/foaf#me"
  }, {
    "@id": "https://greggkellogg.net/foaf#me",
    "@type": "Person",
    "name": "Gregg Kellogg"
  }]
}
```

The previous example describes two node objects, for Manu and Gregg, with the `knows` property defined to treat string values as identifiers. Embedding allows the node object for Gregg to be *embedded* as a value of the `knows` property:

EXAMPLE 94: Embedding a node object as property value of another node object

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/"
  },
  "@type": "Person",
  "name": "Manu Sporny",
  "knows": {
    "@id": "https://greggkellogg.net/foaf#me",
    "@type": "Person",
    "name": "Gregg Kellogg"
  }
}
```

A node object, like the one used above, may be used in any value position in the body of a JSON-LD document.

While it is considered a best practice to identify nodes in a graph, at times this is impractical. In the data model, nodes without an explicit identifier are called blank nodes, which can be represented in a serialization such as JSON-LD using a blank node identifier. In the previous example, the top-level node for Manu does not have an identifier, and does not need one to describe it within the data model. However, if we were to want to describe a *knows* relationship from Gregg to Manu, we would need to introduce a blank node identifier (here `_:b0`).

EXAMPLE 95: Referencing an unidentified node

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/"
  },
  "@id": "_:b0",
  "@type": "Person",
  "name": "Manu Sporny",
  "knows": {
    "@id": "https://greggkellogg.net/foaf#me",
    "@type": "Person",
    "name": "Gregg Kellogg",
    "knows": {"@id": "_:b0"}
  }
}
```

Blank node identifiers may be automatically introduced by algorithms such as flattening, but they are also useful for authors to describe such relationships directly.

§ 4.5.1 Identifying Blank Nodes

This section is non-normative.

At times, it becomes necessary to be able to express information without being able to uniquely identify the node with an IRI. This type of node is called a blank node. JSON-LD does not require all nodes to be identified using `@id`. However, some graph topologies may require identifiers to be serializable. Graphs containing loops, e.g., cannot be serialized using embedding alone, `@id` must be used to connect the nodes. In these situations, one can use blank node identifiers, which look like IRIs using an underscore

(_) as scheme. This allows one to reference the node locally within the document, but makes it impossible to reference the node from an external document. The blank node identifier is scoped to the document in which it is used.

EXAMPLE 96: Specifying a local blank node identifier

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": "http://schema.org/",
   ...
   "@id": "_:n1",
   "name": "Secret Agent 1",
   "knows": {
     "name": "Secret Agent 2",
     "knows": { "@id": "_:n1" }
   }
}
```

The example above contains information about two secret agents that cannot be identified with an IRI. While expressing that *agent 1* knows *agent 2* is possible without using blank node identifiers, it is necessary to assign *agent 1* an identifier so that it can be referenced from *agent 2*.

It is worth noting that blank node identifiers may be relabeled during processing. If a developer finds that they refer to the blank node more than once, they should consider naming the node using a dereferenceable IRI so that it can also be referenced from other documents.

§ 4.6 Indexed Values

This section is non-normative.

Sometimes multiple property values need to be accessed in a more direct fashion than iterating though multiple array values. JSON-LD provides an indexing mechanism to allow the use of an intermediate map to associate specific indexes with associated values.

Data Indexing

As described in § 4.6.1 Data Indexing, data indexing allows an arbitrary key to reference a node or value.

Language Indexing

As described in § 4.6.2 Language Indexing, language indexing allows a language to reference a string and be interpreted as the language associated with that string.

Node Identifier Indexing
As described in § 4.6.3 Node Identifier Indexing, node identifier indexing allows an IRI to reference a node and be interpreted as the identifier of that node.

Node Type Indexing
As described in § 4.6.4 Node Type Indexing, node type indexing allows an IRI to reference a node and be interpreted as a type of that node.

See § 4.9 Named Graphs for other uses of indexing in JSON-LD.

§ 4.6.1 Data Indexing

This section is non-normative.

Databases are typically used to make access to data more efficient. Developers often extend this sort of functionality into their application data to deliver similar performance gains. This data may have no meaning from a Linked Data standpoint, but is still useful for an application.

JSON-LD introduces the notion of index maps that can be used to structure data into a form that is more efficient to access. The data indexing feature allows an author to structure data using a simple key-value map where the keys do not map to IRIs. This enables direct access to data instead of having to scan an array in search of a specific item. In JSON-LD such data can be specified by associating the `@index` keyword with a `@container` declaration in the context:

EXAMPLE 97: Indexing data in JSON-LD

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "schema": "http://schema.org/",
    "name": "schema:name",
    "body": "schema:articleBody",
    "athletes": {
      "@id": "schema:athlete",
      "@container": "@index"
    },
    "position": "schema:jobTitle"
  },
  "@id": "http://example.com/",
  "@type": "schema:SportsTeam",
  "name": "San Franciso Giants",
  "athletes": {
    "catcher": {
      "@type": "schema:Person",
      "name": "Buster Posey",
      "position": "Catcher"
    },
    "pitcher": {
      "@type": "schema:Person",
      "name": "Madison Bumgarner",
      "position": "Starting Pitcher"
    },
    ....
  }
}
```

In the example above, the **athletes** term has been marked as an index map. The **catcher** and **pitcher** keys will be ignored semantically, but preserved syntactically, by the JSON-LD Processor. If used in JavaScript, this can allow a developer to access a particular athlete using the following code snippet: `obj.athletes.pitcher`.

The interpretation of the data is expressed in the statements table. **Note how the index keys do not appear in the statements**, but would continue to exist if the document were compacted or expanded (see § 5.2 Compacted Document Form and § 5.1 Expanded Document Form) using a JSON-LD processor.

⚠ Warning

As data indexes are not preserved when round-tripping to RDF; this feature should be used judiciously. Often, other indexing mechanisms, which are preserved, are more appropriate.

The value of `@container` can also be an array containing both `@index` and `@set`. When *compacting*, this ensures that a JSON-LD Processor will use the array form for all values of indexes.

Unless the processing mode is set to `json-ld-1.0`, the special index `@none` is used for indexing data which does not have an associated index, which is useful to maintain a normalized representation.

EXAMPLE 98: Indexing data using @none

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
     "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "body": "schema:articleBody",
    "athletes": {
      "@id": "schema:athlete",
      "@container": "@index"
    },
    "position": "schema:jobTitle"
  },
  "@id": "http://example.com/",
  "@type": "schema:SportsTeam",
  "name": "San Franciso Giants",
  "athletes": {
    "catcher": {
      "@type": "schema:Person",
      "name": "Buster Posey",
      "position": "Catcher"
    },
    "pitcher": {
      "@type": "schema:Person",
      "name": "Madison Bumgarner",
      "position": "Starting Pitcher"
    },
    "@none": {
      "name": "Lou Seal",
      "position": "Mascot"
    },
    ....
  }
}
```

§ 4.6.1.1 Property-based data indexing

This section is non-normative.

In its simplest form (as in the examples above), data indexing assigns no semantics to the keys of an index map. However, in some situations, the keys

used to index objects are semantically linked to these objects, and should be preserved not only syntactically, but also semantically.

Unless the processing mode is set to `json-ld-1.0`, `"@container": "@index"` in a term description can be accompanied with an `"@index"` key. The value of that key must map to an IRI, which identifies the semantic property linking each object to its key.

EXAMPLE 99: Property-based data indexing

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "body": "schema:articleBody",
    "athletes": {
      "@id": "schema:athlete",
      "@container": "@index",
      "@index": "schema:jobTitle"
    }
  },
  "@id": "http://example.com/",
  "@type": "schema:SportsTeam",
  "name": "San Franciso Giants",
  "athletes": {
    "Catcher": {
      ↑ "Catcher" will add `"schema:jobTitle": "Catcher"` when expanded
      "@type": "schema:Person",
      "name": "Buster Posey"
    },
    "Starting Pitcher": {
      "@type": "schema:Person",
      "name": "Madison Bumgarner"
    },
    ....
  }
}
```

NOTE

When using property-based data indexing, index maps can only be used on node objects, not value objects or graph objects. Value objects are restricted to have only certain keys and do not support arbitrary properties.

§ 4.6.2 Language Indexing

This section is non-normative.

JSON which includes string values in multiple languages may be represented using a language map to allow for easily indexing property values by language tag. This enables direct access to language values instead of having to scan an array in search of a specific item. In JSON-LD such data can be specified by associating the `@language` keyword with a `@container` declaration in the context:

EXAMPLE 100: Indexing languaged-tagged strings in JSON-LD

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "vocab": "http://example.com/vocab/",
    "label": {
      "@id": "vocab:label",
      "@container": "@language"
    }
  },
  "@id": "http://example.com/queen",
  "label": {
    "en": "The Queen",
    "de": [ "Die Königin", "Ihre Majestät" ]
  }
}
```

In the example above, the **label** term has been marked as a language map. The **en** and **de** keys are implicitly associated with their respective values by the JSON-LD Processor. This allows a developer to access the German version of the **label** using the following code snippet: `obj.label.de`, which, again, is only appropriate when languages are limited to the primary language sub-tag

and do not depend on other sub-tags, such as `"de-at"`.

The value of `@container` can also be an array containing both `@language` and `@set`. When *compacting*, this ensures that a JSON-LD Processor will use the array form for all values of language tags.

EXAMPLE 101: Indexing languaged-tagged strings in JSON-LD with @set representation

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "vocab": "http://example.com/vocab/",
    "label": {
      "@id": "vocab:label",
      "@container": ["@language", "@set"]
    }
  },
  "@id": "http://example.com/queen",
  "label": {
    "en": ["The Queen"],
    "de": [ "Die Königin", "Ihre Majestät" ]
  }
}
```

Unless the processing mode is set to `json-ld-1.0`, the special index `@none` is used for indexing strings which do not have a language; this is useful to maintain a normalized representation for string values not having a datatype.

EXAMPLE 102: Indexing languaged-tagged strings using @none for no language

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "vocab": "http://example.com/vocab/",
    "label": {
      "@id": "vocab:label",
      "@container": "@language"
    }
  },
  "@id": "http://example.com/queen",
  "label": {
    "en": "The Queen",
    "de": [ "Die Königin", "Ihre Majestät" ],
    "@none": "The Queen"
  }
}
```

§ 4.6.3 Node Identifier Indexing

This section is non-normative.

In addition to index maps, JSON-LD introduces the notion of id maps for structuring data. The id indexing feature allows an author to structure data using a simple key-value map where the keys map to IRIs. This enables direct access to associated node objects instead of having to scan an array in search of a specific item. In JSON-LD such data can be specified by associating the `@id` keyword with a `@container` declaration in the context:

EXAMPLE 103: Indexing data in JSON-LD by node identifiers

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "body": "schema:articleBody",
    "words": "schema:wordCount",
    "post": {
      "@id": "schema:blogPost",
      "@container": "@id",
      "@context": {
        "@base": "http://example.com/posts/"
      }
    }
  },
  "@id": "http://example.com/",
  "@type": "schema:Blog",
  "name": "World Financial News",
  "post": {
    "1/en": {
      "body": "World commodities were up today with heavy trading of cr
      "words": 1539
    },
    "1/de": {
      "body": "Die Werte an Warenbörsen stiegen im Sog eines starken Ha
      "words": 1204
    }
  }
}
```

In the example above, the `post` term has been marked as an id map. The `http://example.com/posts/1/en` and `http://example.com/posts/1/de` keys will be interpreted as the `@id` property of the node object value.

The interpretation of the data above is exactly the same as that in § 4.6.1 Data Indexing using a JSON-LD processor.

The value of `@container` can also be an array containing both `@id` and `@set`. When *compacting*, this ensures that a JSON-LD processor will use the array form for all values of node identifiers.

EXAMPLE 104: Indexing data in JSON-LD by node identifiers with @set representation

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "body": "schema:articleBody",
    "words": "schema:wordCount",
    "post": {
      "@id": "schema:blogPost",
      "@container": ["@id", "@set"]
    }
  },
  "@id": "http://example.com/",
  "@type": "schema:Blog",
  "name": "World Financial News",
  "post": {
    "http://example.com/posts/1/en": [{
      "body": "World commodities were up today with heavy trading of cr
      "words": 1539
    }],
    "http://example.com/posts/1/de": [{
      "body": "Die Werte an Warenbörsen stiegen im Sog eines starken Ha
      "words": 1204
    }]
  }
}
```

The special index `@none` is used for indexing node objects which do not have an `@id`, which is useful to maintain a normalized representation. The `@none` index may also be a term which expands to `@none`, such as the term *none* used in the example below.

EXAMPLE 105: Indexing data in JSON-LD by node identifiers using @none

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "body": "schema:articleBody",
    "words": "schema:wordCount",
    "post": {
      "@id": "schema:blogPost",
      "@container": "@id"
    },
    "none": "@none"
  },
  "@id": "http://example.com/",
  "@type": "schema:Blog",
  "name": "World Financial News",
  "post": {
    "http://example.com/posts/1/en": {
      "body": "World commodities were up today with heavy trading of cr
      "words": 1539
    },
    "http://example.com/posts/1/de": {
      "body": "Die Werte an Warenbörsen stiegen im Sog eines starken Ha
      "words": 1204
    },
    "none": {
      "body": "Description for object without an @id",
      "words": 20
    }
  }
}
```

NOTE

Id maps are a new feature in JSON-LD 1.1.

§ 4.6.4 Node Type Indexing

This section is non-normative.

In addition to id and index maps, JSON-LD introduces the notion of type maps for structuring data. The type indexing feature allows an author to structure data using a simple key-value map where the keys map to IRIs. This enables data to be structured based on the `@type` of specific node objects. In JSON-LD such data can be specified by associating the `@type` keyword with a `@container` declaration in the context:

EXAMPLE 106: Indexing data in JSON-LD by type

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "affiliation": {
      "@id": "schema:affiliation",
      "@container": "@type"
    }
  },
  "name": "Manu Sporny",
  "affiliation": {
    "schema:Corporation": {
      "@id": "https://digitalbazaar.com/",
      "name": "Digital Bazaar"
    },
    "schema:ProfessionalService": {
      "@id": "https://spec-ops.io",
      "name": "Spec-Ops"
    }
  }
}
```

In the example above, the `affiliation` term has been marked as a type map. The `schema:Corporation` and `schema:ProfessionalService` keys will be interpreted as the `@type` property of the node object value.

The value of `@container` can also be an array containing both `@type` and `@set`. When *compacting*, this ensures that a JSON-LD processor will use the array form for all values of types.

EXAMPLE 107: Indexing data in JSON-LD by type with @set representation

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "affiliation": {
      "@id": "schema:affiliation",
      "@container": ["@type", "@set"]
    }
  },
  "name": "Manu Sporny",
  "affiliation": {
    "schema:Corporation": [{
      "@id": "https://digitalbazaar.com/",
      "name": "Digital Bazaar"
    }],
    "schema:ProfessionalService": [{
      "@id": "https://spec-ops.io",
      "name": "Spec-Ops"
    }]
  }
}
```

The special index `@none` is used for indexing node objects which do not have an `@type`, which is useful to maintain a normalized representation. The `@none` index may also be a term which expands to `@none`, such as the term *none* used in the example below.

EXAMPLE 108: Indexing data in JSON-LD by type using @none

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "schema": "http://schema.org/",
    "name": "schema:name",
    "affiliation": {
      "@id": "schema:affiliation",
      "@container": "@type"
    },
    "none": "@none"
  },
  "name": "Manu Sporny",
  "affiliation": {
    "schema:Corporation": {
      "@id": "https://digitalbazaar.com/",
      "name": "Digital Bazaar"
    },
    "schema:ProfessionalService": {
      "@id": "https://spec-ops.io",
      "name": "Spec-Ops"
    },
    "none": {
      "@id": "https://greggkellogg.net/",
      "name": "Gregg Kellogg"
    }
  }
}
```

As with id maps, when used with `@type`, a container may also include `@set` to ensure that key values are always contained in an array.

NOTE

Type maps are a new feature in JSON-LD 1.1.

§ 4.7 Included Nodes

This section is non-normative.

Sometimes it is also useful to list node objects as part of another node object. For instance, to represent a set of resources which are used by some other resource. Included blocks may be also be used to collect such secondary node objects which can be referenced from a primary node object. For an example, consider a node object containing a list of different items, some of which share some common elements:

EXAMPLE 109: Included Blocks

Input

```
{
  "@context": {
    "@version": 1.1,
    "@vocab": "http://example.org/",
    "classification": {"@type": "@vocab"},
    "service": {"@type": "@vocab"}
  },
  "@id": "http://example.org/base/1",
  "@type": "Thing-with-Items",
  "items": [{
    "@id":"http://example.org/base/2",
    "classification": "enum#c6",
    "service": "enum#s2"
  }, {
    "@id": "http://example.org/base/3",
    "classification": "enum#c6"
  }],
  "@included": [{
    "@id": "http://example.org/enum#c6",
    "@type": "Type",
    "label": "Classification 6"
  }, {
    "@id": "http://example.org/enum#s2",
    "@type": "Service",
    "label": "Login Service"
  }]
}
```

When flattened, this will move the `enum:c6` and `enum:s2` elements from the included block into the outer array.

EXAMPLE 110: Flattened form for included blocks

Flattened (Result) | Statements | Turtle | Open in playground

Result

```
[{
  "@id": "http://example.org/base/1",
  "@type": ["http://example.org/Thing-with-Items"],
  "http://example.org/items": [
    {"@id": "http://example.org/base/2"},
    {"@id": "http://example.org/base/3"}
  ]
}, {
  "@id": "http://example.org/enum#c6",
  "@type": ["http://example.org/Type"],
  "http://example.org/label": [{"@value": "Classification 6
}, {
  "@id": "http://example.org/enum#s2",
  "@type": ["http://example.org/Service"],
  "http://example.org/label": [{"@value": "Login Service"}]
}, {
  "@id": "http://example.org/base/2",
  "http://example.org/classification": [
    {"@id": "http://example.org/enum#c6"}
  ],
  "http://example.org/service": [
    {"@id": "http://example.org/enum#s2"}
  ]
}, {
  "@id": "http://example.org/base/3",
  "http://example.org/classification": [
    {"@id": "http://example.org/enum#c6"}
  ]
}
]
```

Included resources are described in Inclusion of Related Resources of *JSON API* [JSON.API] as a way to include related resources associated with some primary resource; `@included` provides an analogous possibility in JSON-LD.

As a by product of the use of `@included` within node objects, a map may contain only `@included`, to provide a feature similar to that described in § 4.1 Advanced Context Usage, where `@graph` is used to described disconnected nodes.

EXAMPLE 111: Describing disconnected nodes with @included

Compacted (Input) | Expanded (Result) | Flattened | Statements | Turtle

Open in playground

Input

```
{
  "@context": {
    "Person": "http://xmlns.com/foaf/0.1/Person",
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": {"@id": "http://xmlns.com/foaf/0.1/knows", "@type": "@id"}
  },
  "@included": [{
    "@id": "http://manu.sporny.org/about#manu",
    "@type": "Person",
    "name": "Manu Sporny",
    "knows": "https://greggkellogg.net/foaf#me"
  }, {
    "@id": "https://greggkellogg.net/foaf#me",
    "@type": "Person",
    "name": "Gregg Kellogg",
    "knows": "http://manu.sporny.org/about#manu"
  }]
}
```

However, in contrast to `@graph`, `@included` does not interact with other properties contained within the same map, a feature discussed further in § 4.9 Named Graphs.

§ 4.8 Reverse Properties

This section is non-normative.

JSON-LD serializes directed graphs. That means that every property points from a node to another node or value. However, in some cases, it is desirable to serialize in the reverse direction. Consider for example the case where a person and its children should be described in a document. If the used vocabulary does not provide a *children* property but just a *parent* property, every node representing a child would have to be expressed with a property pointing to the parent as in the following example.

EXAMPLE 112: A document with children linking to their parent

Compacted (Input) | Expanded (Result) | Statements | Turtle | Open in playground

Input

```
[
  {
    "@id": "#homer",
    "http://example.com/vocab#name": "Homer"
  }, {
    "@id": "#bart",
    "http://example.com/vocab#name": "Bart",
    "http://example.com/vocab#parent": { "@id": "#homer" }
  }, {
    "@id": "#lisa",
    "http://example.com/vocab#name": "Lisa",
    "http://example.com/vocab#parent": { "@id": "#homer" }
  }
]
```

Expressing such data is much simpler by using JSON-LD's `@reverse` keyword:

EXAMPLE 113: A person and its children using a reverse property

Compacted (Input) | Expanded (Result) | Flattened | Statements | Turtle

Open in playground

Input

```
{
  "@id": "#homer",
  "http://example.com/vocab#name": "Homer",
  "@reverse": {
    "http://example.com/vocab#parent": [
      {
        "@id": "#bart",
        "http://example.com/vocab#name": "Bart"
      }, {
        "@id": "#lisa",
        "http://example.com/vocab#name": "Lisa"
      }
    ]
  }
}
```

The `@reverse` keyword can also be used in expanded term definitions to create reverse properties as shown in the following example:

EXAMPLE 114: Using @reverse to define reverse properties

Compacted (Input) | Expanded (Result) | Flattened | Statements | Turtle

Open in playground

Input

```
{
  "@context": { "name": "http://example.com/vocab#name",
    "children": { "@reverse": "http://example.com/vocab#parent" }
  },
  "@id": "#homer",
  "name": "Homer",
  "children": [
    {
      "@id": "#bart",
      "name": "Bart"
    }, {
      "@id": "#lisa",
      "name": "Lisa"
    }
  ]
}
```

§ 4.9 Named Graphs

This section is non-normative.

At times, it is necessary to make statements about a graph itself, rather than just a single node. This can be done by grouping a set of nodes using the `@graph` keyword. A developer may also name data expressed using the `@graph` keyword by pairing it with an `@id` keyword as shown in the following example:

EXAMPLE 115: Identifying and making statements about a graph

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
    "generatedAt": {
      "@id": "http://www.w3.org/ns/prov#generatedAtTime",
      "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
    },
    "Person": "http://xmlns.com/foaf/0.1/Person",
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": {"@id": "http://xmlns.com/foaf/0.1/knows", "@type": "@id"}
  },
  "@id": "http://example.org/foaf-graph",
  "generatedAt": "2012-04-09T00:00:00",
  "@graph": [
    {
      "@id": "http://manu.sporny.org/about#manu",
      "@type": "Person",
      "name": "Manu Sporny",
      "knows": "https://greggkellogg.net/foaf#me"
    }, {
      "@id": "https://greggkellogg.net/foaf#me",
      "@type": "Person",
      "name": "Gregg Kellogg",
      "knows": "http://manu.sporny.org/about#manu"
    }
  ]
}
```

The example above expresses a named graph that is identified by the IRI `http://example.org/foaf-graph`. That graph is composed of the statements about Manu and Gregg. Metadata about the graph itself is expressed via the `generatedAt` property, which specifies when the graph was generated.

When a JSON-LD document's top-level structure is a map that contains no other keys than `@graph` and optionally `@context` (properties that are not mapped to an IRI or a keyword are ignored), `@graph` is considered to express the otherwise implicit default graph. This mechanism can be useful when a number of nodes exist at the document's top level that share the same context, which is, e.g., the case when a document is flattened. The `@graph` keyword collects such nodes in an array and allows the use of a shared context.

EXAMPLE 116: Using @graph to explicitly express the default graph

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/"
  },
  "@graph": [
    {
      "@id": "http://manu.sporny.org/about#manu",
      "@type": "Person",
      "name": "Manu Sporny"
    }, {
      "@id": "https://greggkellogg.net/foaf#me",
      "@type": "Person",
      "name": "Gregg Kellogg"
    }
  ]
}
```

In this case, embedding can not be used as the graph contains unrelated nodes. This is equivalent to using multiple node objects in array and defining the `@context` within each node object:

EXAMPLE 117: Context needs to be duplicated if @graph is not used

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
[
  {
    "@context": {
      "@vocab": "http://xmlns.com/foaf/0.1/",
      "knows": {"@type": "@id"}
    },
    "@id": "http://manu.sporny.org/about#manu",
    "@type": "Person",
    "name": "Manu Sporny",
    "knows": "https://greggkellogg.net/foaf#me"
  },
  {
    "@context": {
      "@vocab": "http://xmlns.com/foaf/0.1/",
      "knows": {"@type": "@id"}
    },
    "@id": "https://greggkellogg.net/foaf#me",
    "@type": "Person",
    "name": "Gregg Kellogg",
    "knows": "http://manu.sporny.org/about#manu"
  }
]
```

§ 4.9.1 Graph Containers

This section is non-normative.

In some cases, it is useful to logically partition data into separate graphs, without making this explicit within the JSON expression. For example, a JSON document may contain data against which other metadata is asserted and it is useful to separate this data in the data model using the notion of named graphs, without the syntactic overhead associated with the `@graph` keyword.

An expanded term definition can use `@graph` as the value of `@container`. This indicates that values of this term should be considered to be named graphs, where the graph name is an automatically assigned blank node identifier creating an implicitly named graph. When expanded, these become simple graph objects.

A different example uses an anonymously named graph as follows:

EXAMPLE 118: Implicitly named graph

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "@base": "http://dbpedia.org/resource/",
    "said": "http://example.com/said",
    "wrote": {"@id": "http://example.com/wrote", "@container": "@graph"
  },
  "@id": "William_Shakespeare",
  "wrote": {
    "@id": "Richard_III_of_England",
    "said": "My kingdom for a horse"
  }
}
```

The example above expresses an anonymously named graph making a statement. The default graph includes a statement saying that the subject wrote that statement. This is an example of separating statements into a named graph, and then making assertions about the statements contained within that named graph.

NOTE

Strictly speaking, the value of such a term is not a named graph, rather it is the graph name associated with the named graph, which exists separately within the dataset.

NOTE

Graph Containers are a new feature in JSON-LD 1.1.

§ 4.9.2 Named Graph Data Indexing

This section is non-normative.

In addition to indexing node objects by index, graph objects may also be indexed by an index. By using the `@graph` container type, introduced in § 4.9.1 Graph Containers in addition to `@index`, an object value of such a property is

treated as a key-value map where the keys do not map to IRIs, but are taken from an `@index` property associated with named graphs which are their values. When expanded, these must be simple graph objects

The following example describes a default graph referencing multiple named graphs using an index map.

EXAMPLE 119: Indexing graph data in JSON-LD

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
     "@version": 1.1,
     "schema": "http://schema.org/",
     "name": "schema:name",
     "body": "schema:articleBody",
     "words": "schema:wordCount",
     "post": {
       "@id": "schema:blogPost",
       "@container": ["@graph", "@index"]
     }
  },
  "@id": "http://example.com/",
  "@type": "schema:Blog",
  "name": "World Financial News",
  "post": {
     "en": {
       "@id": "http://example.com/posts/1/en",
       "body": "World commodities were up today with heavy trading of c
       "words": 1539
     },
     "de": {
       "@id": "http://example.com/posts/1/de",
       "body": "Die Werte an Warenbörsen stiegen im Sog eines starken H
       "words": 1204
     }
  }
}
```

As with index maps, when used with `@graph`, a container may also include `@set` to ensure that key values are always contained in an array.

The special index `@none` is used for indexing graphs which do not have an `@index` key, which is useful to maintain a normalized representation. Note, however, that compacting a document where multiple unidentified named

graphs are compacted using the `@none` index will result in the content of those graphs being merged. To prevent this, give each graph a distinct `@index` key.

EXAMPLE 120: Indexing graphs using @none for no index

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
     "@version": 1.1,
     "schema": "http://schema.org/",
     "name": "schema:name",
     "body": "schema:articleBody",
     "words": "schema:wordCount",
     "post": {
       "@id": "schema:blogPost",
       "@container": ["@graph", "@index"]
     }
  },
  "@id": "http://example.com/",
  "@type": "schema:Blog",
  "name": "World Financial News",
  "post": {
     "en": {
       "@id": "http://example.com/posts/1/en",
       "body": "World commodities were up today with heavy trading of c
       "words": 1539
     },
     "@none": {
       "@id": "http://example.com/posts/1/no-language",
       "body": "Die Werte an Warenbörsen stiegen im Sog eines starken H
       "words": 1204
     }
  }
}
```

NOTE

Named Graph Data Indexing is a new feature in JSON-LD 1.1.

§ 4.9.3 Named Graph Indexing

This section is non-normative.

In addition to indexing node objects by identifier, graph objects may also be indexed by their graph name. By using the `@graph` container type, introduced in § 4.9.1 Graph Containers in addition to `@id`, an object value of such a property is treated as a key-value map where the keys represent the identifiers of named graphs which are their values.

The following example describes a default graph referencing multiple named graphs using an id map.

EXAMPLE 121: Referencing named graphs using an id map

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "generatedAt": {
      "@id": "http://www.w3.org/ns/prov#generatedAtTime",
      "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
    },
    "Person": "http://xmlns.com/foaf/0.1/Person",
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": {
      "@id": "http://xmlns.com/foaf/0.1/knows",
      "@type": "@id"
    },
    "graphMap": {
      "@id": "http://example.org/graphMap",
      "@container": ["@graph", "@id"]
    }
  },
  "@id": "http://example.org/foaf-graph",
  "generatedAt": "2012-04-09T00:00:00",
  "graphMap": {
    "http://manu.sporny.org/about": {
      "@id": "http://manu.sporny.org/about#manu",
      "@type": "Person",
      "name": "Manu Sporny",
      "knows": "https://greggkellogg.net/foaf#me"
    },
    "https://greggkellogg.net/foaf": {
      "@id": "https://greggkellogg.net/foaf#me",
      "@type": "Person",
      "name": "Gregg Kellogg",
      "knows": "http://manu.sporny.org/about#manu"
    }
  }
}
```

As with id maps, when used with `@graph`, a container may also include `@set` to ensure that key values are always contained in an array.

As with id maps, the special index `@none` is used for indexing named graphs which do not have an `@id`, which is useful to maintain a normalized representation. The `@none` index may also be a term which expands to `@none`.

Note, however, that if multiple graphs are represented without an `@id`, they will be merged on expansion. To prevent this, use `@none` judiciously, and consider giving graphs their own distinct identifier.

EXAMPLE 122: Referencing named graphs using an id map with @none

Compacted (Input) | Expanded (Result) | Statements | TriG | Open in playground

Input

```
{
  "@context": {
    "@version": 1.1,
    "generatedAt": {
      "@id": "http://www.w3.org/ns/prov#generatedAtTime",
      "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
    },
    "Person": "http://xmlns.com/foaf/0.1/Person",
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": {"@id": "http://xmlns.com/foaf/0.1/knows", "@type": "@id"}
    "graphMap": {
      "@id": "http://example.org/graphMap",
      "@container": ["@graph", "@id"]
    }
  },
  "@id": "http://example.org/foaf-graph",
  "generatedAt": "2012-04-09T00:00:00",
  "graphMap": {
    "@none": [{
      "@id": "http://manu.sporny.org/about#manu",
      "@type": "Person",
      "name": "Manu Sporny",
      "knows": "https://greggkellogg.net/foaf#me"
    }, {
      "@id": "https://greggkellogg.net/foaf#me",
      "@type": "Person",
      "name": "Gregg Kellogg",
      "knows": "http://manu.sporny.org/about#manu"
    }]
  }
}
```

NOTE

Graph Containers are a new feature in JSON-LD 1.1.

§ 4.10 Loading Documents

This section is non-normative.

The JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] defines the interface to a JSON-LD Processor and includes a number of methods used for manipulating different forms of JSON-LD (see § 5. Forms of JSON-LD). This includes a general mechanism for loading remote documents, including referenced JSON-LD documents and remote contexts, and potentially extracting embedded JSON-LD from other formats such as [HTML]. This is more fully described in Remote Document and Context Retrieval in [JSON-LD11-API].

A ***documentLoader*** can be useful in a number of contexts where loading remote documents can be problematic:

- Remote context documents should be cached to prevent overloading the location of the remote context for each request. Normally, an HTTP caching infrastructure might be expected to handle this, but in some contexts this might not be feasible. A documentLoader implementation might provide separate logic for performing such caching.
- Non-standard URL schemes may not be widely implemented, or may have behavior specific to a given application domain. A documentLoader can be defined to implement document retrieval semantics.
- Certain well-known contexts may be statically cached within a documentLoader implementation. This might be particularly useful in embedded applications, where it is not feasible, or even possible, to access remote documents.
- For security purposes, the act of remotely retrieving a document may provide a signal of application behavior. The judicious use of a documentLoader can isolate the application and reduce its online fingerprint.

§ 5. Forms of JSON-LD

This section is non-normative.

As with many data formats, there is no single correct way to describe data in JSON-LD. However, as JSON-LD is used for describing graphs, certain transformations can be used to change the shape of the data, without changing its meaning as Linked Data.

Expanded Document Form
: Expansion is the process of taking a JSON-LD document and applying a context so that the `@context` is no longer necessary. This process is described further in § 5.1 Expanded Document Form.

Compacted Document Form
: Compaction is the process of applying a provided context to an existing JSON-LD document. This process is described further in § 5.2 Compacted Document Form.

Flattened Document Form
: Flattening is the process of extracting embedded nodes to the top level of the JSON tree, and replacing the embedded node with a reference, creating blank node identifiers as necessary. This process is described further in § 5.3 Flattened Document Form.

Framed Document Form
: Framing is used to shape the data in a JSON-LD document, using an example frame document which is used to both match the flattened data and show an example of how the resulting data should be shaped. This process is described further in § 5.4 Framed Document Form.

§ 5.1 Expanded Document Form

This section is non-normative.

The JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] defines a method for *expanding* a JSON-LD document. ***Expansion*** is the process of taking a JSON-LD document and applying a context such that all IRIs, types, and values are expanded so that the `@context` is no longer necessary.

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

EXAMPLE 123: Sample JSON-LD document to be expanded

Input

```
{
   "@context": {
      "name": "http://xmlns.com/foaf/0.1/name",
      "homepage": {
        "@id": "http://xmlns.com/foaf/0.1/homepage",
        "@type": "@id"
      }
   },
   "name": "Manu Sporny",
   "homepage": "http://manu.sporny.org/"
}
```

Running the JSON-LD Expansion algorithm against the JSON-LD input document provided above would result in the following output:

EXAMPLE 124: Expanded form for the previous example

Expanded (Result) | Statements | Turtle | Open in playground

Result

```
[
  {
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Manu Sporny" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://manu.sporny.org/" }
    ]
  }
]
```

JSON-LD's media type defines a `profile` parameter which can be used to signal or request ***expanded document form***. The profile URI identifying expanded document form is `http://www.w3.org/ns/json-ld#expanded`.

§ 5.2 Compacted Document Form

This section is non-normative.

The JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] defines a method for *compacting* a JSON-LD document. ***Compaction*** is

the process of applying 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. Often this makes it simpler to work with document as the data is expressed in application-specific terms. Compacted documents are also typically easier to read for humans.

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

EXAMPLE 125: Sample expanded JSON-LD document

Input

```
[
  {
    "http://xmlns.com/foaf/0.1/name": [ "Manu Sporny" ],
    "http://xmlns.com/foaf/0.1/homepage": [
      {
       "@id": "http://manu.sporny.org/"
      }
    ]
  }
]
```

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

EXAMPLE 126: Sample context

Context

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

Running the JSON-LD Compaction algorithm given the context supplied above against the JSON-LD input document provided above would result in the following output:

EXAMPLE 127: Compact form of the sample document once sample context has been applied

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  },
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/"
}
```

JSON-LD's media type defines a `profile` parameter which can be used to signal or request ***compacted document form***. The profile URI identifying compacted document form is `http://www.w3.org/ns/json-ld#compacted`.

The details of Compaction are described in the Compaction algorithm in [JSON-LD11-API]. This section provides a short description of how the algorithm operates as a guide to authors creating contexts to be used for *compacting* JSON-LD documents.

The purpose of compaction is to apply the term definitions, vocabulary mapping, default language, and base IRI to an existing JSON-LD document to cause it to be represented in a form that is tailored to the use of the JSON-LD document directly as JSON. This includes representing values as strings, rather than value objects, where possible, shortening the use of list objects into simple arrays, reversing the relationship between nodes, and using data maps to index into multiple values instead of representing them as an array of values.

§ 5.2.1 Shortening IRIs

This section is non-normative.

In an expanded JSON-LD document, IRIs are always represented as absolute IRIs. In many cases, it is preferable to use a shorter version, either a relative IRI reference, compact IRI, or term. Compaction uses a combination of elements in a context to create a shorter form of these IRIs. See § 4.1.2

Default Vocabulary, § 4.1.3 Base IRI, and § 4.1.5 Compact IRIs for more details.

The vocabulary mapping can be used to shorten IRIs that may be *vocabulary relative* by removing the IRI prefix that matches the vocabulary mapping. This is done whenever an IRI is determined to be vocabulary relative, i.e., used as a property, or a value of `@type`, or as the value of a term described as `"@type": "@vocab"`.

EXAMPLE 128: Compacting using a default vocabulary

Given the following expanded document:

Input

```
[{
  "@id": "http://example.org/places#BrewEats",
  "@type": ["http://example.org/Restaurant"],
  "http://example.org/name": [{"@value": "Brew Eats"}]
}]
```

And the following context:

Context

```
{
  "@context": {
    "@vocab": "http://example.org/"
  }
}
```

The compaction algorithm will shorten all vocabulary-relative IRIs that begin with `http://xmlns.com/foaf/0.1/`:

Result

```
{
  "@context": {
    "@vocab": "http://example.org/"
  },
  "@id": "http://example.org/places#BrewEats",
  "@type": "Restaurant",
  "name": "Brew Eats"
}
```

Note that two IRIs were shortened, unnecessary arrays are removed, and simple string values are replaced with the string.

See Security Considerations in § C. IANA Considerations for a discussion on how string vocabulary-relative IRI resolution via concatenation.

EXAMPLE 129: Compacting using a base IRI

Given the following expanded document:

Input

```
[{
  "@id": "http://example.com/document.jsonld",
  "http://www.w3.org/2000/01/rdf-schema#label": [{"@value": "Ju
}]
```

And the following context:

Context

```
{
  "@context": {
    "@base": "http://example.com/",
    "label": "http://www.w3.org/2000/01/rdf-schema#label"
  }
}
```

The compaction algorithm will shorten all document-relative IRIs that begin with `http://example.com/`:

Result

```
{
  "@context": {
    "@base": "http://example.com/",
    "label": "http://www.w3.org/2000/01/rdf-schema#label"
  },
  "@id": "document.jsonld",
  "label": "Just a simple document"
}
```

§ 5.2.2 Representing Values as Strings

This section is non-normative.

To be unambiguous, the expanded document form always represents nodes and values using node objects and value objects. Moreover, property values are always contained within an array, even when there is only one value. Sometimes this is useful to maintain a uniformity of access, but most JSON data use the simplest possible representation, meaning that properties have single values, which are represented as strings or as structured values such as node objects. By default, compaction will represent values which are simple strings as strings, but sometimes a value is an IRI, a date, or some other typed value for which a simple string representation would loose

information. By specifying this within a term definition, the semantics of a string value can be inferred from the definition of the term used as a property. See § 4.2 Describing Values for more details.

EXAMPLE 130: Coercing Values to Strings

Given the following expanded document:

Input

```
[{
  "http://example.com/plain": [
    {"@value": "string"},
    {"@value": true},
    {"@value": 1}
  ],
  "http://example.com/date": [
    {
      "@value": "2018-02-16",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    }
  ],
  "http://example.com/en": [
    {"@value": "English", "@language": "en"}
  ],
  "http://example.com/iri": [
    {"@id": "http://example.com/some-location"}
  ]
}]
```

And the following context:

Context

```
{
  "@context": {
    "@vocab": "http://example.com/",
    "date": {"@type": "http://www.w3.org/2001/XMLSchema#dat
    "en":   {"@language": "en"},
    "iri": {"@type": "@id"}
  }
}
```

The compacted version will use string values for the defined terms when the values match the term definition. Note that there is no term defined for "plain", that is created automatically using the vocabulary mapping. Also, the other native values, `1` and `true`, can be represented without defining a specific type mapping.

```
{
  "@context": {
    "@vocab": "http://example.com/",
    "date": {"@type": "http://www.w3.org/2001/XMLSchema#date"},
    "en":   {"@language": "en"},
    "iri": {"@type": "@id"}
  },
  "plain": ["string", true, 1],
  "date": "2018-02-16",
  "en": "English",
  "iri": "http://example.com/some-location"
}
```

§ 5.2.3 Representing Lists as Arrays

This section is non-normative.

As described in § 4.3.1 Lists, JSON-LD has an expanded syntax for representing ordered values, using the `@list` keyword. To simplify the representation in JSON-LD, a term can be defined with `"@container": "@list"` which causes all values of a property using such a term to be considered ordered.

EXAMPLE 131: Using Arrays for Lists

Given the following expanded document:

Input

```
[{
  "http://xmlns.com/foaf/0.1/nick": [{
    "@list": [
      {"@value": "joe"},
      {"@value": "bob"},
      {"@value": "jaybee"}
    ]
  }]
}]
```

And the following context:

Context

```
{
  "@context": {
    "nick": {
      "@id": "http://xmlns.com/foaf/0.1/nick",
      "@container": "@list"
    }
  }
}
```

The compacted version eliminates the explicit list object.

```
{
  "@context": {
    "nick": {
      "@id": "http://xmlns.com/foaf/0.1/nick",
      "@container": "@list"
    }
  },
  "nick": [ "joe", "bob", "jaybee" ]
}
```

§ 5.2.4 Reversing Node Relationships

This section is non-normative.

In some cases, the property used to relate two nodes may be better expressed if the nodes have a reverse direction, for example, when describing a

relationship between two people and a common parent. See § 4.8 Reverse Properties for more details.

EXAMPLE 132: Reversing Node Relationships

Given the following expanded document:

Input

```
[{
  "@id": "http://example.org/#homer",
  "http://example.com/vocab#name": [{"@value": "Homer"}],
  "@reverse": {
    "http://example.com/vocab#parent": [{
      "@id": "http://example.org/#bart",
      "http://example.com/vocab#name": [{"@value": "Bart"}]
    }, {
      "@id": "http://example.org/#lisa",
      "http://example.com/vocab#name": [{"@value": "Lisa"}]
    }]
  }
}]
```

And the following context:

Context

```
{
  "@context": {
    "name": "http://example.com/vocab#name",
    "children": { "@reverse": "http://example.com/vocab#par
  }
}
```

The compacted version eliminates the `@reverse` property by describing "children" as the reverse of "parent".

```
{
  "@context": {
    "name": "http://example.com/vocab#name",
    "children": { "@reverse": "http://example.com/vocab#parent" }
  },
  "@id": "#homer",
  "name": "Homer",
  "children": [
    { "@id": "#bart", "name": "Bart"},
    { "@id": "#lisa", "name": "Lisa"}
  ]
}
```

Reverse properties can be even more useful when combined with framing,

which can actually make node objects defined at the top-level of a document to become embedded nodes. JSON-LD provides a means to index such values, by defining an appropriate @container definition within a term definition.

§ 5.2.5 Indexing Values

This section is non-normative.

Properties with multiple values are typically represented using an unordered array. This means that an application working on an internalized representation of that JSON would need to iterate through the values of the array to find a value matching a particular pattern, such as a language-tagged string using the language en.

EXAMPLE 133: Indexing language-tagged strings

Given the following expanded document:

Input

```
[{
  "@id": "http://example.com/queen",
  "http://example.com/vocab/label": [
    {"@value": "The Queen", "@language": "en"},
    {"@value": "Die Königin", "@language": "de"},
    {"@value": "Ihre Majestät", "@language": "de"}
  ]
}]
```

And the following context:

Context

```
{
  "@context": {
    "vocab": "http://example.com/vocab/",
    "label": {
      "@id": "vocab:label",
      "@container": "@language"
    }
  }
}
```

The compacted version uses a map value for "label", with the keys representing the language tag and the values are the strings associated with the relevant language tag.

```
{
  "@context": {
    "vocab": "http://example.com/vocab/",
    "label": {
      "@id": "vocab:label",
      "@container": "@language"
    }
  },
  "@id": "http://example.com/queen",
  "label": {
    "en": "The Queen",
    "de": [ "Die Königin", "Ihre Majestät" ]
  }
}
```

Data can be indexed on a number of different keys, including @id, @type, @language, @index and more. See § 4.6 Indexed Values and § 4.9 Named Graphs for more details.

§ 5.2.6 Normalizing Values as Objects

This section is non-normative.

Sometimes it's useful to compact a document, but keep the node object and value object representations. For this, a term definition can set `"@type": "@none"`. This causes the Value Compaction algorithm to always use the object form of values, although components of that value may be compacted.

EXAMPLE 134: Forcing Object Values

Given the following expanded document:

Input

```
[{
  "http://example.com/notype": [
    {"@value": "string"},
    {"@value": true},
    {"@value": false},
    {"@value": 1},
    {"@value": 10.0},
    {"@value": "plain"},
    {"@value": "false", "@type": "http://www.w3.org/2001/XMLSc
    {"@value": "english", "@language": "en"},
    {"@value": "2018-02-17", "@type": "http://www.w3.org/2001/
    {"@id": "http://example.com/iri"}
  ]
}]
```

And the following context:

Context

```
{
  "@context": {
    "@version": 1.1,
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "notype": {"@id": "http://example.com/notype", "@type":
  }
}
```

The compacted version will use string values for the defined terms when the values match the term definition. Also, the other native values, `1` and `true`, can be represented without defining a specific type mapping.

```
{
  "@context": {
    "@version": 1.1,
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "notype": {"@id": "http://example.com/notype", "@type": "@none"}
  },
  "notype": [
    {"@value": "string"},
    {"@value": true},
    {"@value": false},
    {"@value": 1},
    {"@value": 10.0},
    {"@value": "plain"},
    {"@value": "false", "@type": "xsd:boolean"},
    {"@value": "english", "@language": "en"},
    {"@value": "2018-02-17", "@type": "xsd:date"},
    {"@id": "http://example.com/iri"}
  ]
}
```

§ 5.2.7 Representing Singular Values as Arrays

This section is non-normative.

Generally, when compacting, properties having only one value are represented as strings or maps, while properties having multiple values are represented as an array of strings or maps. This means that applications accessing such properties need to be prepared to accept either representation. To force all values to be represented using an array, a term definition can set `"@container": "@set"`. Moreover, `@set` can be used in combination with other container settings, for example looking at our language-map example from § 5.2.5 Indexing Values:

EXAMPLE 135: Indexing language-tagged strings and @set

Given the following expanded document:

Input

```
[{
  "@id": "http://example.com/queen",
  "http://example.com/vocab/label": [
    {"@value": "The Queen", "@language": "en"},
    {"@value": "Die Königin", "@language": "de"},
    {"@value": "Ihre Majestät", "@language": "de"}
  ]
}]
```

And the following context:

Context

```
{
  "@context": {
    "@version": 1.1,
    "@vocab": "http://example.com/vocab/",
    "label": {
      "@container": ["@language", "@set"]
    }
  }
}
```

The compacted version uses a map value for "label" as before. and the values are the relevant strings but always represented using an array.

Result

```
{
  "@context": {
    "@version": 1.1,
    "@vocab": "http://example.com/vocab/",
    "label": {
      "@container": ["@language", "@set"]
    }
  },
  "@id": "http://example.com/queen",
  "label": {
    "en": ["The Queen"],
    "de": [ "Die Königin", "Ihre Majestät" ]
  }
}
```

§ 5.2.8 Term Selection

This section is non-normative.

When compacting, the Compaction algorithm will compact using a term for a property only when the values of that property match the `@container`, `@type`, and `@language` specifications for that term definition. This can actually split values between different properties, all of which have the same IRI. In case there is no matching term definition, the compaction algorithm will compact using the absolute IRI of the property.

EXAMPLE 136: Term Selection

Given the following expanded document:

Input

```
[{
  "http://example.com/vocab/property": [
    {"@value": "string"},
    {"@value": true},
    {"@value": 1},
    {"@value": "false", "@type": "http://www.w3.org/2001/XMLSc
    {"@value": "10", "@type": "http://www.w3.org/2001/XMLSchem
    {"@value": "english", "@language": "en"},
    {"@value": "2018-02-17", "@type": "http://www.w3.org/2001/
    {"@id": "http://example.com/some-location"}
  ]
}]
```

And the following context:

Context

```
{
  "@context": {
    "vocab":   "http://example.com/vocab/",
    "xsd":     "http://www.w3.org/2001/XMLSchema#",
    "integer": {"@id": "vocab:property", "@type": "xsd:inte
    "date":    {"@id": "vocab:property", "@type": "xsd:date
    "english": {"@id": "vocab:property", "@language": "en"}
    "list":    {"@id": "vocab:property", "@container": "@li
    "iri":     {"@id": "vocab:property", "@type": "@id"}
  }
}
```

Note that the values that match the "integer", "english", "date", and "iri" terms are properly matched, and that everything that does not explicitly match is added to a property created using a compact IRI.

Result

```
{
  "@context": {
    "vocab":   "http://example.com/vocab/",
    "xsd":     "http://www.w3.org/2001/XMLSchema#",
    "integer": {"@id": "vocab:property", "@type": "xsd:intege
    "date":    {"@id": "vocab:property", "@type": "xsd:date"}
    "english": {"@id": "vocab:property", "@language": "en"},
    "list":    {"@id": "vocab:property", "@container": "@list
    "iri":     {"@id": "vocab:property", "@type": "@id"}
  },
  "vocab:property": [
    "string", true, 1,
    {"@value": "false", "@type": "xsd:boolean"}
  ],
  "integer": "10",
  "english": "english",
  "date": "2018-02-17",
  "iri": "http://example.com/some-location"
}
```

§ 5.3 Flattened Document Form

This section is non-normative.

The JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] defines a method for *flattening* a JSON-LD document. ***Flattening*** collects all properties of a node in a single map and labels all blank nodes with blank node identifiers. This ensures a shape of the data and consequently may drastically simplify the code required to process JSON-LD in certain applications.

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

EXAMPLE 137: Sample JSON-LD document to be flattened

Input

```
{
  "@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": [
    {
      "@id": "http://manu.sporny.org/about#manu",
      "name": "Manu Sporny"
    }, {
      "name": "Dave Longley"
    }
  ]
}
```

Running the JSON-LD Flattening algorithm against the JSON-LD input document in the example above and using the same context would result in the following output:

EXAMPLE 138: Flattened and compacted form for the previous example

Open in playground

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@graph": [{
    "@id": "http://me.markus-lanthaler.com/",
    "name": "Markus Lanthaler",
    "knows": [
      { "@id": "http://manu.sporny.org/about#manu" },
      { "@id": "_:b0" }
    ]
  }, {
    "@id": "http://manu.sporny.org/about#manu",
    "name": "Manu Sporny"
  }, {
    "@id": "_:b0",
    "name": "Dave Longley"
  }]
}
```

JSON-LD's media type defines a `profile` parameter which can be used to signal or request ***flattened document form***. The profile URI identifying flattened document form is `http://www.w3.org/ns/json-ld#flattened`. It can be combined with the profile URI identifying expanded document form or compacted document form.

§ 5.4 Framed Document Form

This section is non-normative.

The JSON-LD 1.1 Framing specification [JSON-LD11-FRAMING] defines a method for *framing* a JSON-LD document. ***Framing*** is used to shape the data in a JSON-LD document, using an example frame document which is used to both match the flattened data and show an example of how the resulting data should be shaped.

For example, assume the following JSON-LD frame:

EXAMPLE 139: Sample library frame

Frame

```
{
  "@context": {
    "@version": 1.1,
    "@vocab": "http://example.org/"
  },
  "@type": "Library",
  "contains": {
    "@type": "Book",
    "contains": {
      "@type": "Chapter"
    }
  }
}
```

This frame document describes an embedding structure that would place objects with type *Library* at the top, with objects of type *Book* that were linked to the library object using the *contains* property embedded as property values. It also places objects of type *Chapter* within the referencing *Book* object as embedded values of the *Book* object.

When using a flattened set of objects that match the frame components:

EXAMPLE 140: Flattened library objects

Input

```
{
  "@context": {
    "@vocab": "http://example.org/",
    "contains": {"@type": "@id"}
  },
  "@graph": [{
    "@id": "http://example.org/library",
    "@type": "Library",
    "contains": "http://example.org/library/the-republic"
  }, {
    "@id": "http://example.org/library/the-republic",
    "@type": "Book",
    "creator": "Plato",
    "title": "The Republic",
    "contains": "http://example.org/library/the-republic#intro
  }, {
    "@id": "http://example.org/library/the-republic#introducti
    "@type": "Chapter",
    "description": "An introductory chapter on The Republic.",
    "title": "The Introduction"
  }]
}
```

The Frame Algorithm can create a new document which follows the structure of the frame:

EXAMPLE 141: Framed library objects

Open in playground

```
{
  "@context": {
    "@version": 1.1,
    "@vocab": "http://example.org/"
  },
  "@id": "http://example.org/library",
  "@type": "Library",
  "contains": {
    "@id": "http://example.org/library/the-republic",
    "@type": "Book",
    "contains": {
      "@id": "http://example.org/library/the-republic#introduction",
      "@type": "Chapter",
      "description": "An introductory chapter on The Republic.",
      "title": "The Introduction"
    },
    "creator": "Plato",
    "title": "The Republic"
  }
}
```

JSON-LD's media type defines a `profile` parameter which can be used to signal or request ***framed document form***. The profile URI identifying framed document form is `http://www.w3.org/ns/json-ld#framed`.

JSON-LD's media type also defines a `profile` parameter which can be used to identify a script element in an HTML document containing a frame. The first script element of type `application/ld+json;profile=http://www.w3.org/ns/json-ld#frame` will be used to find a frame..

§ 6. Modifying Behavior with Link Relationships

Certain aspects of JSON-LD processing can be modified using ***HTTP Link Headers*** [RFC8288]. These can be used when retrieving resources that are not, themselves, JSON-LD, but can be interpreted as JSON-LD by using information in a Link Relation.

When processing normal JSON documents, a link relation can be specified using the HTTP Link Header returned when fetching a remote document, as described in § 6.1 Interpreting JSON as JSON-LD.

In other cases, a resource may be returned using a representation that cannot easily be interpreted as JSON-LD. Normally, HTTP content negotiation would be used to allow a client to specify a preference for JSON-LD over another representation, but in certain situations, it is not possible or practical for a server to respond appropriately to such requests. For this, an HTTP Link Header can be used to provide an alternate location for a document to be used in place of the originally requested resource, as described in § 6.2 Alternate Document Location.

§ 6.1 Interpreting JSON as JSON-LD

Ordinary JSON documents can be interpreted as JSON-LD by providing an explicit JSON-LD context document. One way to provide this is by using referencing a JSON-LD context document in an HTTP Link Header. Doing so allows JSON to be unambiguously machine-readable without requiring developers to drastically change their documents and provides an upgrade path for existing infrastructure without breaking existing clients that rely on the `application/json` media type or a media type with a `+json` suffix as defined in [RFC6839].

In order to use an external context with an ordinary JSON document, when retrieving an ordinary JSON document via HTTP, processors *MUST* attempt to retrieve any JSON-LD document referenced by a Link Header with:

- `rel="http://www.w3.org/ns/json-ld#context"`, and
- `type="application/ld+json"`.

The referenced document *MUST* have a top-level JSON object. The `@context` entry within that object is added to the top-level JSON object of the referencing document. If an array is at the top-level of the referencing document and its items are JSON objects, the `@context` subtree is added to all array items. All extra information located outside of the `@context` subtree in the referenced document *MUST* be discarded. Effectively this means that the active context is initialized with the referenced external context. A response *MUST NOT* contain more than one HTTP Link Header using the `http://www.w3.org/ns/json-ld#context` link relation.

Other mechanisms for providing a JSON-LD Context *MAY* be described for other URI schemes.

The JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] provides for an expandContext option for specifying a context to use

when expanding JSON documents programmatically.

The following example demonstrates the use of an external context with an ordinary JSON document over HTTP:

EXAMPLE 142: Referencing a JSON-LD context from a JSON document via an HTTP Link Header

```
GET /ordinary-json-document.json HTTP/1.1
Host: example.com
Accept: application/ld+json,application/json,*/*;q=0.1

====================================

HTTP/1.1 200 OK
...
Content-Type: application/json
Link: <https://json-ld.org/contexts/person.jsonld>; rel="http://www.w3.

{
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/",
  "image": "http://twitter.com/account/profile_image/markuslanthaler"
}
```

Please note that JSON-LD documents served with the `application/ld+json` media type *MUST* have all context information, including references to external contexts, within the body of the document. Contexts linked via a `http://www.w3.org/ns/json-ld#context` HTTP Link Header *MUST* be ignored for such documents.

§ 6.2 Alternate Document Location

Documents which can't be directly interpreted as JSON-LD can provide an alternate location containing JSON-LD. One way to provide this is by referencing a JSON-LD document in an HTTP Link Header. This might be useful, for example, when the URL associated with a namespace naturally contains an HTML document, but the JSON-LD context associated with that URL is located elsewhere.

To specify an alternate location, a non-JSON resource (i.e., one using a media type other than `application/json` or a derivative) can return the alternate location using a Link Header with:

- `rel="alternate"`, and
- `type="application/ld+json"`.

A response *MUST NOT* contain more than one HTTP Link Header using the `alternate` link relation with `type="application/ld+json"` .

Other mechanisms for providing an alternate location *MAY* be described for other URI schemes.

The following example demonstrates the use of an alternate location with an ordinary HTTP document over HTTP:

EXAMPLE 143: Specifying an alternate location via an HTTP Link Header

```
GET /index.html HTTP/1.1
Host: example.com
Accept: application/ld+json,application/json,*/*;q=0.1

====================================

HTTP/1.1 200 OK
...
Content-Type: text/html
Link: <alternate.jsonld>; rel="alternate"; type="application/ld+json"

<html>
  <head>
    <title>Primary Entrypoint</title>
  </head>
  <body>
    <p>This is the primary entrypoint for a vocabulary</p>
  </body>
</html>
```

A processor seeing a non-JSON result will note the presence of the link header and load that document instead.

§ 7. Embedding JSON-LD in HTML Documents

NOTE

This section describes features available with a documentLoader supporting HTML script extraction. See Remote Document and Context Retrieval for more information.

JSON-LD content can be easily embedded in HTML [HTML] by placing it in a script element with the `type` attribute set to `application/ld+json`. Doing so creates a data block.

EXAMPLE 144: Embedding JSON-LD in HTML

Compacted (Input) | Expanded (Result) | Statements | Turtle

Input

```
<script type="application/ld+json">
{
  "@context": "https://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
</script>
```

Defining how such data may be used is beyond the scope of this specification. The embedded JSON-LD document might be extracted as is or, e.g., be interpreted as RDF.

If JSON-LD content is extracted as RDF [RDF11-CONCEPTS], it *MUST* be expanded into an RDF Dataset using the Deserialize JSON-LD to RDF Algorithm [JSON-LD11-API]. Unless a specific script is targeted (see § 7.3 Locating a Specific JSON-LD Script Element), all script elements with `type` `application/ld+json` *MUST* be processed and merged into a single dataset with equivalent blank node identifiers contained in separate script elements treated as if they were in a single document (i.e., blank nodes are shared between different JSON-LD script elements).

EXAMPLE 145: Combining multiple JSON-LD script elements into a single dataset

HTML Embedded (Input) | Statements | Turtle (Result)

Input

```
<p>Data describing Dave</p>
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@id": "https://digitalbazaar.com/author/dlongley/",
  "@type": "Person",
  "name": "Dave Longley"
}
</script>

<p>Data describing Gregg</p>
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@id": "https://greggkellogg.net/foaf#me",
  "@type": "Person",
  "name": "Gregg Kellogg"
}
</script>
```

§ 7.1 Inheriting base IRI from HTML's `base` element

When processing a JSON-LD script element, the Document Base URL of the containing HTML document, as defined in [HTML], is used to establish the default base IRI of the enclosed JSON-LD content.

EXAMPLE 146: Using the document base URL to establish the default base IRI

Compacted (Input) | Expanded (Result) | Statements | Turtle

Input

```
<html>
  <head>
    <base href="http://dbpedia.org/resource/"/>
    <script type="application/ld+json">
    {
      "@context": "https://json-ld.org/contexts/person.jsonld"
      "@id": "John_Lennon",
      "name": "John Lennon",
      "born": "1940-10-09",
      "spouse": "Cynthia_Lennon"
    }
    </script>
  </head>
</html>
```

HTML allows for Dynamic changes to base URLs. This specification does not require any specific behavior, and to ensure that all systems process the base IRI equivalently, authors *SHOULD* either use IRIs, or explicitly as defined in § 4.1.3 Base IRI. Implementations (particularly those natively operating in the [DOM]) *MAY* take into consideration Dynamic changes to base URLs.

§ 7.2 Restrictions for contents of JSON-LD `script` elements

This section is non-normative.

Due to the HTML Restrictions for contents of `<script>` elements additional encoding restrictions are placed on JSON-LD data contained in script elements.

Authors should avoid using character sequences in scripts embedded in HTML which may be confused with a *comment-open*, *script-open*, *comment-close*, or *script-close*.

NOTE

Such content should be escaped as indicated below, however the content will remain escaped after processing through the JSON-LD API [JSON-LD11-API].

- `&` → & (ampersand, U+0026)
- `<` → < (less-than sign, U+003C)
- `>` → > (greater-than sign, U+003E)
- `"` → " (quotation mark, U+0022)
- `'` → ' (apostrophe, U+0027)

EXAMPLE 147: Embedding JSON-LD containing HTML in HTML

Compacted (Input) | Expanded (Result) | Turtle

Input

```
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@type": "WebPageElement",
  "name": "Encoding Issues",
  "description": "Issues list such as unescaped &lt;/script&gt
}
</script>
```

§ 7.3 Locating a Specific JSON-LD Script Element

A specific script element within an HTML document may be located using a fragment identifier matching the unique identifier of the script element within the HTML document located by a URL (see [DOM]). A JSON-LD processor *MUST* extract only the specified data block's contents parsing it as a standalone JSON-LD document and *MUST NOT* merge the result with any other markup from the same HTML document.

For example, given an HTML document located at `http://example.com/document`, a script element identified by "dave" can be targeted using the URL `http://example.com/document#dave`.

EXAMPLE 148: Targeting a specific script element by id

Compacted (Input) | Statements | Turtle

Targeting a script element with id "gregg"

Input

```
<p>Data describing Dave</p>
<script id="dave" type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@id": "https://digitalbazaar.com/author/dlongley/",
  "@type": "Person",
  "name": "Dave Longley"
}
</script>

<p>Data describing Gregg</p>
<script id="gregg" type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@id": "https://greggkellogg.net/foaf#me",
  "@type": "Person",
  "name": "Gregg Kellogg"
}
</script>
```

§ 8. Data Model

JSON-LD is a serialization format for Linked Data based on JSON. It is therefore important to distinguish between the syntax, which is defined by JSON in [RFC8259], and the ***data model*** which is an extension of the RDF data model [RDF11-CONCEPTS]. The precise details of how JSON-LD relates to the RDF data model are given in § 10. Relationship to RDF.

To ease understanding for developers unfamiliar with the RDF model, the following summary is provided:

- A JSON-LD document serializes a RDF Dataset [RDF11-CONCEPTS], which is a collection of graphs that comprises exactly one default graph and zero or more named graphs.
- The default graph does not have a name and *MAY* be empty.
- Each named graph is a pair consisting of an IRI or blank node identifier (the graph name) and a graph. Whenever practical, the graph name

SHOULD be an IRI.

- A graph is a labeled directed graph, i.e., a set of nodes connected by directed-arcs.
- Every directed-arc is labeled with an IRI or a blank node identifier. Within the JSON-LD syntax these arc labels are called properties. Whenever practical, a directed-arc *SHOULD* be labeled with an IRI.

> NOTE
> The use of blank node identifiers to label properties is obsolete, and may be removed in a future version of JSON-LD. Consider using a document-relative IRI, instead, such as `#`.

- Every node is an IRI, a blank node, or a literal, although syntactically lists and native JSON values may be represented directly.
- A node having an outgoing edge *MUST* be an IRI or a blank node.
- A graph *MUST NOT* contain unconnected nodes, i.e., nodes which are not connected by an property to any other node.

> EXAMPLE 149: Illegal Unconnected Node
>
> ```
> {
>  "@id": "http://example.org/1"
> }
> ```

> NOTE
> This effectively just prohibits unnested, empty node objects and unnested node objects that contain only an `@id`. A document may have nodes which are unrelated, as long as one or more properties are defined, or the node is referenced from another node object.

- An IRI (Internationalized Resource Identifier) is a string that conforms to the syntax defined in [RFC3987]. IRIs used within a graph *SHOULD* return a Linked Data document describing the resource denoted by that IRI when being dereferenced.
- A blank node is a node which is neither an IRI, nor a JSON-LD value, nor a list. A blank node is identified using a 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 `_:`.

- A JSON-LD value is a typed value, a string (which is interpreted as a typed value with type `xsd:string`), a number (numbers with a non-zero fractional part, i.e., the result of a modulo-1 operation, or which are too large to represent as integers (see Data Round Tripping) in [JSON-LD11-API]), are interpreted as typed values with type `xsd:double`, all other numbers are interpreted as typed values with type `xsd:integer`), `true` or `false` (which are interpreted as typed values with type `xsd:boolean`), or a language-tagged string.
- A typed value consists of a value, which is a string, and a type, which is an IRI.
- 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].
  Processors *MAY* normalize language tags to lowercase.
- Either strings, or language-tagged strings may include a base direction, which represents an extension to the underlying RDF data model.
- A list is a sequence of zero or more IRIs, blank nodes, and JSON-LD values. Lists are interpreted as ***RDF list structures*** [RDF11-MT].

JSON-LD documents *MAY* contain data that cannot be represented by the data model defined above. Unless otherwise specified, such data is ignored when a JSON-LD document is being processed. One result of this rule is that properties which are not mapped to an IRI, a blank node, or keyword will be ignored.

Additionally, the JSON serialization format is internally represented using the JSON-LD internal representation, which uses the generic concepts of lists, maps, strings, numbers, booleans, and null to describe the data represented by a JSON document.

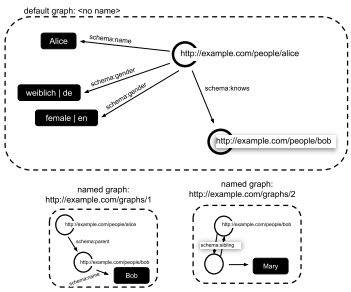

Figure 1 An illustration of a linked data dataset.
A description of the linked data dataset diagram is available in the Appendix. Image available in SVG and PNG formats.

The dataset described in this figure can be represented as follows:

EXAMPLE 150: Linked Data Dataset

Compacted (Input) | Expanded (Result) | Statements | TriG

Input

```
{
  "@context": [
    "http://schema.org/",
    {"@base": "http://example.com/"}
  ],
  "@graph": [{
    "@id": "people/alice",
    "gender": [
      {"@value": "weiblich", "@language": "de"},
      {"@value": "female",   "@language": "en"}
    ],
    "knows": {"@id": "people/bob"},
    "name": "Alice"
  }, {
    "@id": "graphs/1",
    "@graph": {
      "@id": "people/alice",
      "parent": {
        "@id": "people/bob",
        "name": "Bob"
      }
    }
  }, {
    "@id": "graphs/2",
    "@graph": {
      "@id": "people/bob",
      "sibling": {
        "name": "Mary",
        "sibling": {"@id": "people/bob"}
      }
    }
  }]
}
```

> NOTE
>
> Note the use of `@graph` at the outer-most level to describe three top-level resources (two of them named graphs). The named graphs use `@graph` in addition to `@id` to provide the name for each graph.

§ 9. JSON-LD Grammar

This section restates the syntactic conventions described in the previous sections more formally.

A JSON-LD document *MUST* be valid JSON text as described in [RFC8259], or some format that can be represented in the JSON-LD internal representation that is equivalent to valid JSON text.

A JSON-LD document *MUST* be a single node object, a map consisting of only the entries `@context` and/or `@graph`, or an array of zero or more node objects.

In contrast to JSON, in JSON-LD the keys in objects *MUST* be unique.

Whenever a keyword is discussed in this grammar, the statements also apply to an alias for that keyword.

> NOTE
>
> JSON-LD allows keywords to be aliased (see § 4.1.6 Aliasing Keywords for details). For example, if the active context defines the term `id` as an alias for `@id`, that alias may be legitimately used as a substitution for `@id`. Note that keyword aliases are not expanded during context processing.

§ 9.1 Terms

A term is a short-hand string that expands to an IRI, blank node identifier, or keyword.

A term *MUST NOT* equal any of the JSON-LD keywords, other than `@type`.

When used as the prefix in a Compact IRI, to avoid the potential ambiguity of a prefix being confused with an IRI scheme, terms *SHOULD NOT* come from the list of URI schemes as defined in [IANA-URI-SCHEMES]. Similarly, to

avoid confusion between a Compact IRI and a term, terms *SHOULD NOT* include a colon (`:`) and *SHOULD* be restricted to the form of `isegment-nz-nc` as defined in [RFC3987].

To avoid forward-compatibility issues, a term *SHOULD NOT* start with an `@` character followed exclusively by one or more *ALPHA* characters (see [RFC5234]) as future versions of JSON-LD may introduce additional keywords. Furthermore, the term *MUST NOT* be an empty string (`""`) as not all programming languages are able to handle empty JSON keys.

See § 3.1 The Context and § 3.2 IRIs for further discussion on mapping terms to IRIs.

§ 9.2 Node Objects

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

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

The properties of a node in a graph may be spread among different node objects within a document. When that happens, the keys of the different node objects need to be merged to create the properties of the resulting node.

A node object *MUST* be a map. All keys which are not IRIs, compact IRIs, terms valid in the active context, or one of the following keywords (or alias of such a keyword) *MUST* be ignored when processed:

- `@context`,
- `@id`,
- `@included`,
- `@graph`,
- `@nest`,
- `@type`,
- `@reverse`, or
- `@index`

If the node object contains the `@context` key, its value *MUST* be null, an IRI reference, a context definition, or an array composed of any of these.

If the node object contains the `@id` key, its value *MUST* be an IRI reference, or a compact IRI (including blank node identifiers). See § 3.3 Node Identifiers, § 4.1.5 Compact IRIs, and § 4.5.1 Identifying Blank Nodes for further discussion on `@id` values.

If the node object contains the `@graph` key, its value *MUST* be a node object or an array of zero or more node objects. If the node object also contains an `@id` keyword, its value is used as the graph name of a named graph. See § 4.9 Named Graphs for further discussion on `@graph` values. As a special case, if a map contains no keys other than `@graph` and `@context`, and the map is the root of the JSON-LD document, the map is not treated as a node object; this is used as a way of defining node objects that may not form a connected graph. This allows a context to be defined which is shared by all of the constituent node objects.

If the node object contains the `@type` key, its value *MUST* be either an IRI reference, a compact IRI (including blank node identifiers), a term defined in the active context expanding into an IRI, or an array of any of these. See § 3.5 Specifying the Type for further discussion on `@type` values.

If the node object contains the `@reverse` key, its value *MUST* be a map containing entries representing reverse properties. Each value of such a reverse property *MUST* be an IRI reference, a compact IRI, a blank node identifier, a node object or an array containing a combination of these.

If the node object contains the `@included` key, its value *MUST* be an included block. See § 9.13 Included Blocks for further discussion on included blocks.

If the node object contains the `@index` key, its value *MUST* be a string. See § 4.6.1 Data Indexing for further discussion on `@index` values.

If the node object contains the `@nest` key, its value *MUST* be a map or an array of map which *MUST NOT* include a value object. See § 9.14 Property Nesting for further discussion on `@nest` values.

Keys in a node object that are not keywords *MAY* expand to an IRI using the active context. The values associated with keys that expand to an IRI *MUST* be one of the following:

- string,
- number,

- `true`,
- `false`,
- null,
- node object,
- graph object,
- value object,
- list object,
- set object,
- an array of zero or more of any of the possibilities above,
- a language map,
- an index map,
- an included block
- an id map, or
- a type map

§ 9.3 Frame Objects

When framing, a frame object extends a node object to allow entries used specifically for framing.

- A frame object *MAY* include a default object as the value of any key which is not a keyword. Values of `@default` *MAY* include the value `@null`, or an array containing only `@null`, in addition to other values allowed in the grammar for values of entry keys expanding to IRIs.
- The values of `@id` and `@type` *MAY* additionally be an empty map (wildcard), an array containing only an empty map, an empty array (match none) an array of IRIs.
- A frame object *MAY* include an entry with the key `@embed` with any value from `@always`, `@list`, and `@never`.
- A frame object *MAY* include entries with the boolean valued keys `@explicit`, `@omitDefault`, or `@requireAll`
- In addition to other property values, a property of a frame object *MAY* include a value pattern (See § 9.6 Value Patterns).

See [JSON-LD11-FRAMING] for a description of how frame objects are used.

§ 9.4 Graph Objects

A graph object represents a named graph, which *MAY* include an explicit graph name. A map is a graph object if it exists outside of a JSON-LD context, it contains an `@graph` entry (or an alias of that keyword), it is not the top-most map in the JSON-LD document, and it consists of no entries other than `@graph`, `@index`, `@id` and `@context`, or an alias of one of these keywords.

If the graph object contains the `@context` key, its value *MUST* be null, an IRI reference, a context definition, or an array composed of any of these.

If the graph object contains the `@id` key, its value is used as the identifier (graph name) of a named graph, and *MUST* be an IRI reference, or a compact IRI (including blank node identifiers). See § 3.3 Node Identifiers, § 4.1.5 Compact IRIs, and § 4.5.1 Identifying Blank Nodes for further discussion on `@id` values.

A graph object without an `@id` entry is also a simple graph object and represents a named graph without an explicit identifier, although in the data model it still has a graph name, which is an implicitly allocated blank node identifier.

The value of the `@graph` key *MUST* be a node object or an array of zero or more node objects. See § 4.9 Named Graphs for further discussion on `@graph` values..

§ 9.5 Value Objects

A value object is used to explicitly associate a type or a language with a value to create a typed value or a language-tagged string and possibly associate a base direction.

A value object *MUST* be a map containing the `@value` key. It *MAY* also contain an `@type`, an `@language`, an `@direction`, an `@index`, or an `@context` key but *MUST NOT* contain both an `@type` and either `@language` or `@direction` keys at the same time. A value object *MUST NOT* contain any other keys that expand to an IRI or keyword.

The value associated with the `@value` key *MUST* be either a string, a number, `true`, `false` or null. If the value associated with the `@type` key is `@json`, the value *MAY* be either an array or an object.

The value associated with the `@type` key *MUST* be a term, an IRI, a compact

IRI, a string which can be turned into an IRI using the vocabulary mapping, `@json`, or null.

The value associated with the `@language` key *MUST* have the lexical form described in [BCP47], or be null.

The value associated with the `@direction` key *MUST* be one of `"ltr"` or `"rtl"`, or be null.

The value associated with the `@index` key *MUST* be a string.

See § 4.2.1 Typed Values and § 4.2.4 String Internationalization for more information on value objects.

§ 9.6 Value Patterns

When framing, a value pattern extends a value object to allow entries used specifically for framing.

- The values of `@value`, `@language`, `@direction` and `@type` *MAY* additionally be an empty map (wildcard), an array containing only an empty map, an empty array (match none), an array of strings.

§ 9.7 Lists and Sets

A list represents an *ordered* set of values. A set represents an *unordered* set of values. Unless otherwise specified, arrays are unordered in JSON-LD. As such, the `@set` keyword, when used in the body of a JSON-LD document, represents just syntactic sugar which is optimized away when processing the document. However, it is very helpful when used within the context of a document. Values of terms associated with an `@set` or `@list` container will always be represented in the form of an array when a document is processed—even if there is just a single value that would otherwise be optimized to a non-array form in compacted document form. This simplifies post-processing of the data as the data is always in a deterministic form.

A list object *MUST* be a map that contains no keys that expand to an IRI or keyword other than `@list` and `@index`.

A set object *MUST* be a map that contains no keys that expand to an IRI or keyword other than `@set` and `@index`. Please note that the `@index` key will be ignored when being processed.

In both cases, the value associated with the keys `@list` and `@set` *MUST* be one of the following types:

- string,
- number,
- `true`,
- `false`,
- null,
- node object,
- value object, or
- an array of zero or more of the above possibilities

See § 4.3 Value Ordering for further discussion on sets and lists.

§ 9.8 Language Maps

A language map is used to associate a language with a value in a way that allows easy programmatic access. A language map may be used as a term value within a node object if the term is defined with `@container` set to `@language`, or an array containing both `@language` and `@set` . The keys of a language map *MUST* be strings representing [BCP47] language tags, the keyword `@none`, or a term which expands to `@none`, and the values *MUST* be any of the following types:

- null,
- string, or
- an array of zero or more of the strings

See § 4.2.4 String Internationalization for further discussion on language maps.

§ 9.9 Index Maps

An index map allows keys that have no semantic meaning, but should be preserved regardless, to be used in JSON-LD documents. An index map may be used as a term value within a node object if the term is defined with `@container` set to `@index`, or an array containing both `@index` and `@set` . The values of the entries of an index map *MUST* be one of the following types:

- string,
- number,
- `true`,
- `false`,
- null,
- node object,
- value object,
- list object,
- set object,
- an array of zero or more of the above possibilities

See § 4.6.1 Data Indexing for further information on this topic.

Index Maps may also be used to map indexes to associated named graphs, if the term is defined with `@container` set to an array containing both `@graph` and `@index`, and optionally including `@set`. The value consists of the node objects contained within the named graph which is indexed using the referencing key, which can be represented as a simple graph object if the value does not include `@id`, or a named graph if it includes `@id`.

§ 9.10 Property-based Index Maps

A property-based index map is a variant of index map were indexes are semantically preserved in the graph as property values. A property-based index map may be used as a term value within a node object if the term is defined with `@container` set to `@index`, or an array containing both `@index` and `@set`, and with `@index` set to a string. The values of a property-based index map *MUST* be node objects or strings which expand to node objects.

When expanding, if the active context contains a term definition for the value of `@index`, this term definition will be used to expand the keys of the index map. Otherwise, the keys will be expanded as simple value objects. Each node object in the expanded values of the index map will be added an additional property value, where the property is the expanded value of `@index`, and the value is the expanded referencing key.

See § 4.6.1.1 Property-based data indexing for further information on this topic.

§ 9.11 Id Maps

An id map is used to associate an IRI with a value that allows easy programmatic access. An id map may be used as a term value within a node object if the term is defined with `@container` set to `@id`, or an array containing both `@id` and `@set`. The keys of an id map *MUST* be IRIs (IRI references or compact IRIs (including blank node identifiers)), the keyword `@none`, or a term which expands to `@none`, and the values *MUST* be node objects.

If the value contains a property expanding to `@id`, its value *MUST* be equivalent to the referencing key. Otherwise, the property from the value is used as the `@id` of the node object value when expanding.

Id Maps may also be used to map graph names to their named graphs, if the term is defined with `@container` set to an array containing both `@graph` and `@id`, and optionally including `@set`. The value consists of the node objects contained within the named graph which is named using the referencing key.

§ 9.12 Type Maps

A type map is used to associate an IRI with a value that allows easy programmatic access. A type map may be used as a term value within a node object if the term is defined with `@container` set to `@type`, or an array containing both `@type` and `@set`. The keys of a type map *MUST* be IRIs (IRI references or compact IRI (including blank node identifiers)), terms, or the keyword `@none`, and the values *MUST* be node objects or strings which expand to node objects.

If the value contains a property expanding to `@type`, and its value is contains the referencing key after suitable expansion of both the referencing key and the value, then the node object already contains the type. Otherwise, the property from the value is added as a `@type` of the node object value when expanding.

§ 9.13 Included Blocks

An included block is used to provide a set of node objects. An included block *MAY* appear as the value of a member of a node object with either the key of `@included` or an alias of `@included`. An included block is either a node object or an array of node objects.

When expanding, multiple included blocks will be coalesced into a single

included block.

§ 9.14 Property Nesting

A nested property is used to gather properties of a node object in a separate map, or array of maps which are not value objects. It is semantically transparent and is removed during the process of expansion. Property nesting is recursive, and collections of nested properties may contain further nesting.

Semantically, nesting is treated as if the properties and values were declared directly within the containing node object.

§ 9.15 Context Definitions

A ***context definition*** defines a local context in a node object.

A context definition *MUST* be a map whose keys *MUST* be either terms, compact IRIs, IRIs, or one of the keywords `@base`, `@import`, `@language`, `@propagate`, `@protected`, `@type`, `@version`, or `@vocab`.

If the context definition has an `@base` key, its value *MUST* be an IRI reference, or null.

If the context definition has an `@direction` key, its value *MUST* be one of `"ltr"` or `"rtl"`, or be null.

If the context definition contains the `@import` keyword, its value *MUST* be an IRI reference. When used as a reference from an `@import`, the referenced context definition *MUST NOT* include an `@import` key, itself.

If the context definition has an `@language` key, its value *MUST* have the lexical form described in [BCP47] or be null.

If the context definition has an `@propagate` key, its value *MUST* be `true` or `false`.

If the context definition has an `@protected` key, its value *MUST* be `true` or `false`.

If the context definition has an `@type` key, its value *MUST* be a map with only the entry `@container` set to `@set`, and optionally an entry `@protected`.

If the context definition has an `@version` key, its value *MUST* be a number with the value `1.1`.

If the context definition has an `@vocab` key, its value *MUST* be an IRI reference, a compact IRI, a blank node identifier, a term, or null.

The value of keys that are not keywords *MUST* be either an IRI, a compact IRI, a term, a blank node identifier, a keyword, null, or an expanded term definition.

§ 9.15.1 Expanded term definition

An expanded term definition is used to describe the mapping between a term and its expanded identifier, as well as other properties of the value associated with the term when it is used as key in a node object.

An expanded term definition *MUST* be a map composed of zero or more keys from `@id`, `@reverse`, `@type`, `@language`, `@container`, `@context`, `@prefix`, `@propagate`, or `@protected`. An expanded term definition *SHOULD NOT* contain any other keys.

When the associated term is `@type`, the expanded term definition *MUST NOT* contain keys other than `@container` and `@protected`. The value of `@container` is limited to the single value `@set`.

If the term being defined is not an IRI or a compact IRI and the active context does not have an `@vocab` mapping, the expanded term definition *MUST* include the `@id` key.

Term definitions with keys which are of the form of an IRI or a compact IRI *MUST NOT* expand to an IRI other than the expansion of the key itself.

If the expanded term definition contains the `@id` keyword, its value *MUST* be null, an IRI, a blank node identifier, a compact IRI, a term, or a keyword.

If an expanded term definition has an `@reverse` entry, it *MUST NOT* have `@id` or `@nest` entries at the same time, its value *MUST* be an IRI, a blank node identifier, a compact IRI, or a term. If an `@container` entry exists, its value *MUST* be null, `@set`, or `@index`.

If the expanded term definition contains the `@type` keyword, its value *MUST* be an IRI, a compact IRI, a term, null, or one of the keywords `@id`, `@json`, `@none`, or `@vocab`.

If the expanded term definition contains the `@language` keyword, its value *MUST* have the lexical form described in [BCP47] or be null.

If the expanded term definition contains the `@container` keyword, its value *MUST* be either `@list`, `@set`, `@language`, `@index`, `@id`, `@graph`, `@type`, or be null or an array containing exactly any one of those keywords, or a combination of `@set` and any of `@index`, `@id`, `@graph`, `@type`, `@language` in any order . `@container` may also be an array containing `@graph` along with either `@id` or `@index` and also optionally including `@set`. If the value is `@language`, when the term is used outside of the `@context`, the associated value *MUST* be a language map. If the value is `@index`, when the term is used outside of the `@context`, the associated value *MUST* be an index map.

If an expanded term definition has an `@context` entry, it *MUST* be a valid `context definition`.

If the expanded term definition contains the `@nest` keyword, its value *MUST* be either `@nest`, or a term which expands to `@nest`.

If the expanded term definition contains the `@prefix` keyword, its value *MUST* be `true` or `false`.

If the expanded term definition contains the `@propagate` keyword, its value *MUST* be `true` or `false`.

If the expanded term definition contains the `@protected` keyword, its value *MUST* be `true` or `false`.

Terms *MUST NOT* be used in a circular manner. That is, the definition of a term cannot depend on the definition of another term if that other term also depends on the first term.

See § 3.1 The Context for further discussion on contexts.

§ 9.16 Keywords

JSON-LD keywords are described in § 1.7 Syntax Tokens and Keywords, this section describes where each keyword may appear within different JSON-LD structures.

Within node objects, value objects, graph objects, list objects, set objects, and nested properties keyword aliases *MAY* be used instead of the corresponding keyword, except for `@context`. The `@context` keyword *MUST NOT* bee aliased. Within local contexts and expanded term definitions, keyword aliases *MAY*

NOT used.

@base

The unaliased @base keyword *MAY* be used as a key in a context definition. Its value *MUST* be an IRI reference, or null.

@container

The unaliased @container keyword *MAY* be used as a key in an expanded term definition. Its value *MUST* be either @list, @set, @language, @index, @id, @graph, @type, or be null, or an array containing exactly any one of those keywords, or a combination of @set and any of @index, @id, @graph, @type, @language in any order. The value may also be an array containing @graph along with either @id or @index and also optionally including @set.

@context

The @context keyword *MUST NOT* be aliased, and *MAY* be used as a key in the following objects:

- node objects (see § 9.2 Node Objects),
- value objects (see § 9.5 Value Objects),
- graph objects (see § 9.4 Graph Objects),
- list objects (see § 9.7 Lists and Sets),
- set objects (see § 9.7 Lists and Sets),
- nested properties (see § 9.14 Property Nesting), and
- expanded term definitions (see § 9.15 Context Definitions).

The value of @context *MUST* be null, an IRI reference, a context definition, or an array composed of any of these.

@direction

The @direction keyword *MAY* be aliased and *MAY* be used as a key in a value object. Its value *MUST* be one of "ltr" or "rtl", or be null.

The unaliased @direction *MAY* be used as a key in a context definition.

See § 4.2.4.1 Base Direction for a further discussion.

@graph

The @graph keyword *MAY* be aliased and *MAY* be used as a key in a node object or a graph object, where its value *MUST* be a value object, node object, or an array of either value objects or node objects.

The unaliased @graph *MAY* be used as the value of the @container key within an expanded term definition.

See § 4.9 Named Graphs.

@id

The @id keyword *MAY* be aliased and *MAY* be used as a key in a node object or a graph object.

The unaliased @id *MAY* be used as a key in an expanded term definition, or as the value of the @container key within an expanded term definition.

The value of the @id key *MUST* be an IRI reference, or a compact IRI (including blank node identifiers).

See § 3.3 Node Identifiers, § 4.1.5 Compact IRIs, and § 4.5.1 Identifying Blank Nodes for further discussion on @id values.

@import

The unaliased @import keyword *MAY* be used in a context definition. Its value *MUST* be an IRI reference. See § 4.1.10 Imported Contexts for a further discussion.

@included

The @included keyword *MAY* be aliased and its value *MUST* be an included block. This keyword is described further in § 4.7 Included Nodes, and § 9.13 Included Blocks.

@index

The @index keyword *MAY* be aliased and *MAY* be used as a key in a node object, value object, graph object, set object, or list object. Its value *MUST* be a string.

The unaliased @index *MAY* be used as the value of the @container key within an expanded term definition.

See § 9.9 Index Maps for a further discussion.

@json

The @json keyword *MAY* be aliased and *MAY* be used as the value of the @type key within a value object or an expanded term definition.

See § 4.2.2 JSON Literals.

@language

The @language keyword *MAY* be aliased and *MAY* be used as a key in a value object. Its value *MUST* be a string with the lexical form described in [BCP47] or be null.

The unaliased @language *MAY* be used as a key in a context definition, or as the value of the @container key within an expanded term definition.

See § 4.2.4 String Internationalization, § 9.8 Language Maps.

@list

The `@list` keyword *MAY* be aliased and *MUST* be used as a key in a list object. The unaliased `@list` *MAY* be used as the value of the `@container` key within an expanded term definition. Its value *MUST* be one of the following:

- string,
- number,
- `true`,
- `false`,
- null,
- node object,
- value object, or
- an array of zero or more of the above possibilities

See § 4.3 Value Ordering for further discussion on sets and lists.

@nest

The `@nest` keyword *MAY* be aliased and *MAY* be used as a key in a node object, where its value must be a map.

The unaliased `@nest` *MAY* be used as the value of a simple term definition, or as a key in an expanded term definition, where its value *MUST* be a string expanding to `@nest`.

See § 9.14 Property Nesting for a further discussion.

@none

The `@none` keyword *MAY* be aliased and *MAY* be used as a key in an index map, id map, language map, type map. See § 4.6.1 Data Indexing, § 4.6.2 Language Indexing, § 4.6.3 Node Identifier Indexing, § 4.6.4 Node Type Indexing, § 4.9.3 Named Graph Indexing, or § 4.9.2 Named Graph Data Indexing for a further discussion.

@prefix

The unaliased `@prefix` keyword *MAY* be used as a key in an expanded term definition. Its value *MUST* be `true` or `false`. See § 4.1.5 Compact IRIs and § 9.15 Context Definitions for a further discussion.

@propagate

The unaliased `@propagate` keyword *MAY* be used in a context definition. Its value *MUST* be `true` or `false`. See § 4.1.9 Context Propagation for a further discussion.

@protected

The unaliased `@protected` keyword *MAY* be used in a context definition, or an expanded term definition. Its value *MUST* be `true` or `false`. See § 4.1.11 Protected Term Definitions for a further discussion.

`@reverse`

The `@reverse` keyword *MAY* be aliased and *MAY* be used as a key in a node object.

The unaliased `@reverse` *MAY* be used as a key in an expanded term definition.

The value of the `@reverse` key *MUST* be an IRI reference, or a compact IRI (including blank node identifiers).

See § 4.8 Reverse Properties and § 9.15 Context Definitions for further discussion.

`@set`

The `@set` keyword *MAY* be aliased and *MUST* be used as a key in a set object. Its value *MUST* be one of the following:

- string,
- number,
- `true`,
- `false`,
- null,
- node object,
- value object, or
- an array of zero or more of the above possibilities

The unaliased `@set` *MAY* be used as the value of the `@container` key within an expanded term definition.

See § 4.3 Value Ordering for further discussion on sets and lists.

`@type`

The `@type` keyword *MAY* be aliased and *MAY* be used as a key in a node object or a value object, where its value *MUST* be a term, IRI reference, or a compact IRI (including blank node identifiers).

The unaliased `@type` *MAY* be used as a key in an expanded term definition, where its value may also be either `@id` or `@vocab`, or as the value of the `@container` key within an expanded term definition.

Within a context, `@type` may be used as the key for an expanded term definition, whose entries are limited to `@container` and `@protected`.

This keyword is described further in § 3.5 Specifying the Type and § 4.2.1 Typed Values.

`@value`

The `@value` keyword *MAY* be aliased and *MUST* be used as a key in a value object. Its value key *MUST* be either a string, a number, `true`, `false` or null. This keyword is described further in § 9.5 Value Objects.

`@version`

The unaliased `@version` keyword *MAY* be used as a key in a context definition. Its value *MUST* be a number with the value `1.1`. This keyword is described further in § 9.15 Context Definitions.

`@vocab`

The unaliased `@vocab` keyword *MAY* be used as a key in a context definition or as the value of `@type` in an expanded term definition. Its value *MUST* be an IRI reference, a compact IRI, a blank node identifier, a term, or null. This keyword is described further in § 9.15 Context Definitions, and § 4.1.2 Default Vocabulary.

§ 10. Relationship to RDF

JSON-LD is a ***concrete RDF syntax*** as described in [RDF11-CONCEPTS]. Hence, a JSON-LD document is both an RDF document *and* a JSON document and correspondingly represents an instance of an RDF data model. However, JSON-LD also extends the RDF data model to optionally allow JSON-LD to serialize ***generalized RDF Datasets***. The JSON-LD extensions to the RDF data model are:

- In JSON-LD properties can be IRIs or blank nodes whereas in RDF properties (predicates) have to be IRIs. This means that JSON-LD serializes generalized RDF Datasets.
- In JSON-LD lists use native JSON syntax, either contained in a list object, or described as such within a context. Consequently, developers using the JSON representation can access list elements directly rather than using the vocabulary for collections described in [RDF-SCHEMA].
- RDF values are either typed *literals* (typed values) or language-tagged strings whereas JSON-LD also supports JSON's native data types, i.e., number, strings, and the boolean values `true` and `false`. The JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] defines the

conversion rules between JSON's native data types and RDF's counterparts to allow round-tripping.

- As an extension to the RDF data model, literals without an explicit datatype *MAY* include a base direction. As there is currently no standardized mechanism for representing the base direction of RDF literals, the JSON-LD to standard RDF transformation loses the base direction. However, the Deserialize JSON-LD to RDF Algorithm provides a means of representing base direction using mechanisms which will preserve round-tripping through non-standard RDF.

> NOTE
>
> The use of blank node identifiers to label properties is obsolete, and may be removed in a future version of JSON-LD, as is the support for generalized RDF Datasets.

Summarized, these differences mean that JSON-LD is capable of serializing any RDF graph or dataset and most, but not all, JSON-LD documents can be directly interpreted as RDF as described in RDF 1.1 Concepts [RDF11-CONCEPTS].

Authors are strongly encouraged to avoid labeling properties using blank node identifiers, instead, consider one of the following mechanisms:

- a relative IRI reference, either relative to the document or the vocabulary (see § 4.1.4 Using the Document Base for the Default Vocabulary for a discussion on using the document base as part of the vocabulary mapping),
- a URN such as `urn:example:1`, see [URN], or
- a "Skolem IRI" as per Replacing Blank Nodes with IRIs of [RDF11-CONCEPTS].

The normative algorithms for interpreting JSON-LD as RDF and serializing RDF as JSON-LD are specified in the JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API].

Even though JSON-LD serializes RDF Datasets, it can also be used as a ***graph source***. In that case, a consumer *MUST* only use the default graph and ignore all named graphs. This allows servers to expose data in languages such as Turtle and JSON-LD using HTTP content negotiation.

NOTE

Publishers supporting both dataset and graph syntaxes have to ensure that the primary data is stored in the default graph to enable consumers that do not support datasets to process the information.

§ 10.1 Serializing/Deserializing RDF

This section is non-normative.

The process of serializing RDF as JSON-LD and deserializing JSON-LD to RDF depends on executing the algorithms defined in RDF Serialization-Deserialization Algorithms in the JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API]. It is beyond the scope of this document to detail these algorithms any further, but a summary of the necessary operations is provided to illustrate the process.

The procedure to deserialize a JSON-LD document to RDF involves the following steps:

1. Expand the JSON-LD document, removing any context; this ensures that properties, types, and values are given their full representation as IRIs and expanded values. Expansion is discussed further in § 5.1 Expanded Document Form.
2. Flatten the document, which turns the document into an array of node objects. Flattening is discussed further in § 5.3 Flattened Document Form.
3. Turn each node object into a series of triples.

For example, consider the following JSON-LD document in compact form:

EXAMPLE 151: 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": [
    {
      "@id": "http://manu.sporny.org/about#manu",
      "name": "Manu Sporny"
    }, {
      "name": "Dave Longley"
    }
  ]
}
```

Running the JSON-LD Expansion and Flattening algorithms against the JSON-LD input document in the example above would result in the following output:

EXAMPLE 152: Flattened and expanded form for the previous example

```
[
  {
    "@id": "_:b0",
    "http://xmlns.com/foaf/0.1/name": "Dave Longley"
  }, {
    "@id": "http://manu.sporny.org/about#manu",
    "http://xmlns.com/foaf/0.1/name": "Manu Sporny"
  }, {
    "@id": "http://me.markus-lanthaler.com/",
    "http://xmlns.com/foaf/0.1/name": "Markus Lanthaler",
    "http://xmlns.com/foaf/0.1/knows": [
      { "@id": "http://manu.sporny.org/about#manu" },
      { "@id": "_:b0" }
    ]
  }
]
```

Deserializing this to RDF now is a straightforward process of turning each

node object into one or more triples. This can be expressed in Turtle as follows:

EXAMPLE 153: Turtle representation of expanded/flattened document

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

_:b0 foaf:name "Dave Longley" .

<http://manu.sporny.org/about#manu> foaf:name "Manu Sporny" .

<http://me.markus-lanthaler.com/> foaf:name "Markus Lanthaler" ;
    foaf:knows <http://manu.sporny.org/about#manu>, _:b0 .
```

The process of serializing RDF as JSON-LD can be thought of as the inverse of this last step, creating an expanded JSON-LD document closely matching the triples from RDF, using a single node object for all triples having a common subject, and a single property for those triples also having a common predicate. The result may then be framed by using the Framing Algorithm described in [JSON-LD11-FRAMING] to create the desired object embedding.

§ 10.2 The `rdf:JSON` Datatype

RDF provides for JSON content as a possible literal value. This allows markup in literal values. Such content is indicated in a graph using a literal whose datatype is set to `rdf:JSON`.

The `rdf:JSON` datatype is defined as follows:

The IRI denoting this datatype
is `http://www.w3.org/1999/02/22-rdf-syntax-ns#JSON`.

The lexical space
is the set of UNICODE [UNICODE] strings which conform to the JSON Grammar as described in Section 2 JSON Grammar of [RFC8259].

The value space
is the union of the four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays) from [ECMASCRIPT]. Two JSON values A and B are considered equal if and only if the following is true:

1. If A and B are both objects, both A and B have the same number of

entries, and each entry in A is equal to a corresponding entry in B where

- the keys are equal (as defined in Section 7.2.12, step 5.a in [ECMASCRIPT]), and
- the values are considered equal through applying this comparison recursively.

2. Otherwise, if A and B are both arrays, both A and B have the same number of elements, and each element A_i is considered equal to the corresponding element B_i through applying this comparison recursively.
3. Otherwise, if A and B satisfy the Strict Equality Comparison defined in Section 7.2.15 in [ECMASCRIPT].
4. Otherwise, A and B are not equal.

The lexical-to-value mapping

is the result of parsing the lexical representation into an internal representation consistent with [ECMASCRIPT] representation created by using the `JSON.parse` function as defined in Section 24.5 The JSON Object of [ECMASCRIPT].

The canonical mapping

is non-normative, as a normative recommendation for JSON canonicalization is not yet defined. Implementations *SHOULD* use the following guidelines when creating the lexical representation of a JSON literal:

- Serialize JSON using no unnecessary whitespace,
- Keys in objects *SHOULD* be ordered lexicographically,
- Native Numeric values *SHOULD* be serialized according to Section 7.1.12.1 of [ECMASCRIPT],
- Strings *SHOULD* be serialized with Unicode codepoints from `U+0000` through `U+001F` using lower case hexadecimal Unicode notation (`\uhhhh`) unless in the set of predefined JSON control characters `U+0008`, `U+0009`, `U+000A`, `U+000C` or `U+000D` which *SHOULD* be serialized as `\b`, `\t`, `\n`, `\f` and `\r` respectively. All other Unicode characters *SHOULD* be serialized "as is", other than `U+005C` (`\`) and `U+0022` (`"`) which *SHOULD* be serialized as `\\` and `\"` respectively.

> ISSUE
> The JSON Canonicalization Scheme [JCS] is an emerging standard for JSON canonicalization not yet ready to be referenced. When a JSON canonicalization standard becomes available, this specification will likely be updated to require such a canonical representation. Users are cautioned from depending on the JSON literal lexical representation as an RDF literal, as the specifics of serialization may change in a future revision of this document.

§ 10.3 The `i18n` Namespace

This section is non-normative.

The `i18n` namespace is used for describing combinations of language tag and base direction in RDF literals. It is used as an alternative mechanism for describing the [BCP47] language tag and base direction of RDF literals that would otherwise use the `xsd:string` or `rdf:langString` datatypes.

Datatypes based on this namespace allow round-tripping of JSON-LD documents using base direction, although the mechanism is not otherwise standardized.

The Deserialize JSON-LD to RDF Algorithm can be used with the ***rdfDirection*** option set to `i18n-datatype` to generate RDF literals using the `i18n` base to create an IRI encoding the base direction along with optional language tag from value objects containing `@direction` by appending to `https://www.w3.org/ns/i18n#` the value of `@language`, if any, followed by an underscore (`"_"`) followed by the value of `@direction`.

> (FEATURE AT RISK) ISSUE
>
> This feature is experimental, as RDF does not have a standard way to represent base direction in RDF literals. A future RDF Working Group may support base direction differently. The JSON-LD Working Group solicits feedback from the community on the usefulness of these transformations.

The following example shows two statements with literal values of `i18n:ar-EG_rtl`, which encodes the language tag `ar-EG` and the base direction `rtl`.

```
@prefix ex: <http://example.org/> .
@prefix i18n: <https://www.w3.org/ns/i18n#> .

# Note that this version preserves the base direction using a non-standarc
[
  ex:title "تصميم و إنشاء مواقع الويب: HTML و CSS"^^i18n:ar-EG_rtl;
  ex:publisher "مكتبة"^^i18n:ar-EG_rtl
] .
```

See § 4.2.4.1 Base Direction for more details on using base direction for strings.

§ 10.4 The `rdf:CompoundLiteral` class and the `rdf:language` and `rdf:direction` properties

This section is non-normative.

This specification defines the `rdf:CompoundLiteral` class, which is in the domain of `rdf:language` and `rdf:direction` to be used for describing RDF literal values containing base direction and a possible language tag to be associated with the string value of `rdf:value` on the same subject.

`rdf:CompoundLiteral`
: A class representing a compound literal.

`rdf:language`
: An RDF property. The range of the property is an `rdfs:Literal`, whose value *MUST* be a well-formed [BCP47] language tag. The domain of the property is `rdf:CompoundLiteral`.

`rdf:direction`
: An RDF property. The range of the property is an `rdfs:Literal`, whose value *MUST* be either `"ltr"` or `"rtl"`. The domain of the property is `rdf:CompoundLiteral`.

The Deserialize JSON-LD to RDF Algorithm can be used with the rdfDirection option set to `compound-literal` to generate RDF literals using these properties to describe the base direction and optional language tag from value objects containing `@direction` and optionally `@language`.

(FEATURE AT RISK) ISSUE

This feature is experimental, as RDF does not have a standard way to represent base direction in RDF literals. A future RDF Working Group may support base direction differently. The JSON-LD Working Group solicits feedback from the community on the usefulness of these transformations.

The following example shows two statements with compoud literals representing strings with the language tag `ar-EG` and base direction `rtl`.

```
@prefix ex: <http://example.org/> .

# Note that this version preserves the base direction using a bnode struct
[
  ex:title [
    rdf:value "HTML و CSS: تصميم و إنشاء مواقع الويب",
    rdf:language "ar-EG",
    rdf:direction "rtl"
  ];
  ex:publisher [
    rdf:value "مكتبة",
    rdf:language "ar-EG",
    rdf:direction "rtl"
  ]
] .
```

See § 4.2.4.1 Base Direction for more details on using base direction for strings.

§ 11. Security Considerations

See, Security Considerations in § C. IANA Considerations.

NOTE

Future versions of this specification may incorporate subresource integrity [SRI] as a means of ensuring that cached and retrieved content matches data retrieved from remote servers; see issue 86.

§ 12. Privacy Considerations

The retrieval of external contexts can expose the operation of a JSON-LD processor, allow intermediate nodes to fingerprint the client application through introspection of retrieved resources (see [fingerprinting-guidance]), and provide an opportunity for a man-in-the-middle attack. To protect against this, publishers should consider caching remote contexts for future use, or use the documentLoader to maintain a local version of such contexts.

§ 13. Internationalization Considerations

As JSON-LD uses the RDF data model, it is restricted by design in its ability to properly record JSON-LD Values which are strings with left-to-right or right-to-left direction indicators. Both JSON-LD and RDF provide a mechanism for specifying the language associated with a string (language-tagged string), but do not provide a means of indicating the base direction of the string.

Unicode provides a mechanism for signaling direction within a string (see *Unicode Bidirectional Algorithm* [UAX9]), however, when a string has an overall base direction which cannot be determined by the beginning of the string, an external indicator is required, such as the [HTML] dir attribute, which currently has no counterpart for RDF literals.

The issue of properly representing base direction in RDF is not something that this Working Group can handle, as it is a limitation or the core RDF data model. This Working Group expects that a future RDF Working Group will consider the matter and add the ability to specify the base direction of language-tagged strings.

Until a more comprehensive solution can be addressed in a future version of this specification, publishers should consider this issue when representing strings where the base direction of the string cannot otherwise be correctly inferred based on the content of the string. See [string-meta] for a discussion best practices for identifying language and base direction for strings used on the Web.

§ A. Image Descriptions

This section is non-normative.

§ A.1 Linked Data Dataset

This section is non-normative.

This section describes the Linked Data Dataset figure in § 8. Data Model.

The image consists of three dashed boxes, each describing a different linked data graph. Each box consists of shapes linked with arrows describing the linked data relationships.

The first box is titled "default graph: <no name>" describes two resources: `http://example.com/people/alice` and `http://example.com/people/bob` (denoting "Alice" and "Bob" respectively), which are connected by an arrow labeled `schema:knows` which describes the knows relationship between the two resources. Additionally, the "Alice" resource is related to three different literals:

Alice
: an RDF literal with no datatype or language.

weiblich | de
: an language-tagged string with the value "weiblich" and language tag "de".

female | en
: an language-tagged string with the value "female" and language tag "en".

The second and third boxes describe two named graphs, with the graph names "http://example.com/graphs/1" and "http://example.com/graphs/1", respectively.

The second box consists of two resources: `http://example.com/people/alice` and `http://example.com/people/bob` related by the `schema:parent` relationship, and names the `http://example.com/people/bob` "Bob".

The third box consists of two resources, one named `http://example.com/people/bob` and the other unnamed. The two resources related to each other using `schema:sibling` relationship with the second named "Mary".

§ B. Relationship to Other Linked Data Formats

This section is non-normative.

The JSON-LD examples below demonstrate how JSON-LD can be used to express semantic data marked up in other linked data formats such as Turtle, RDFa, and Microdata. These sections are merely provided as evidence that JSON-LD is very flexible in what it can express across different Linked Data

approaches.

§ B.1 Turtle

This section is non-normative.

The following are examples of transforming RDF expressed in [Turtle] into JSON-LD.

§ B.1.1 Prefix definitions

The JSON-LD context has direct equivalents for the Turtle `@prefix` declaration:

EXAMPLE 154: A set of statements serialized in Turtle

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://manu.sporny.org/about#manu> a foaf:Person;
  foaf:name "Manu Sporny";
  foaf:homepage <http://manu.sporny.org/> .
```

EXAMPLE 155: The same set of statements serialized in JSON-LD

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/"
  },
  "@id": "http://manu.sporny.org/about#manu",
  "@type": "foaf:Person",
  "foaf:name": "Manu Sporny",
  "foaf:homepage": { "@id": "http://manu.sporny.org/" }
}
```

§ B.1.2 Embedding

Both [Turtle] and JSON-LD allow embedding, although [Turtle] only allows embedding of blank nodes.

EXAMPLE 156: Embedding in Turtle

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://manu.sporny.org/about#manu>
  a foaf:Person;
  foaf:name "Manu Sporny";
  foaf:knows [ a foaf:Person; foaf:name "Gregg Kellogg" ] .
```

EXAMPLE 157: Same embedding example in JSON-LD

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/"
  },
  "@id": "http://manu.sporny.org/about#manu",
  "@type": "foaf:Person",
  "foaf:name": "Manu Sporny",
  "foaf:knows": {
    "@type": "foaf:Person",
    "foaf:name": "Gregg Kellogg"
  }
}
```

§ B.1.3 Conversion of native data types

In JSON-LD numbers and boolean values are native data types. While [Turtle] has a shorthand syntax to express such values, RDF's abstract syntax requires that numbers and boolean values are represented as typed literals. Thus, to allow full round-tripping, the JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API] defines conversion rules between JSON-LD's native data types and RDF's counterparts. Numbers without fractions are converted to `xsd:integer`-typed literals, numbers with fractions to `xsd:double`-typed literals and the two boolean values `true` and `false` to a `xsd:boolean`-typed literal. All typed literals are in canonical lexical form.

EXAMPLE 158: JSON-LD using native data types for numbers and boolean values

```
{
  "@context": {
    "ex": "http://example.com/vocab#"
  },
  "@id": "http://example.com/",
  "ex:numbers": [ 14, 2.78 ],
  "ex:booleans": [ true, false ]
}
```

EXAMPLE 159: Same example in Turtle using typed literals

```
@prefix ex: <http://example.com/vocab#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/>
  ex:numbers "14"^^xsd:integer, "2.78E0"^^xsd:double ;
  ex:booleans "true"^^xsd:boolean, "false"^^xsd:boolean .
```

§ B.1.4 Lists

Both JSON-LD and [Turtle] can represent sequential lists of values.

EXAMPLE 160: A list of values in Turtle

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://example.org/people#joebob> a foaf:Person;
  foaf:name "Joe Bob";
  foaf:nick ( "joe" "bob" "jaybee" ) .
```

EXAMPLE 161: Same example with a list of values in JSON-LD

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/"
  },
  "@id": "http://example.org/people#joebob",
  "@type": "foaf:Person",
  "foaf:name": "Joe Bob",
  "foaf:nick": {
    "@list": [ "joe", "bob", "jaybee" ]
  }
}
```

§ B.2 RDFa

This section is non-normative.

The following example describes three people with their respective names and homepages in RDFa [RDFA-CORE].

EXAMPLE 162: RDFa fragment that describes three people

```
<div prefix="foaf: http://xmlns.com/foaf/0.1/">
   <ul>
      <li typeof="foaf:Person">
        <a property="foaf:homepage" href="http://example.com/bob/">
          <span property="foaf:name">Bob</span>
        </a>
      </li>
      <li typeof="foaf:Person">
        <a property="foaf:homepage" href="http://example.com/eve/">
         <span property="foaf:name">Eve</span>
        </a>
      </li>
      <li typeof="foaf:Person">
        <a property="foaf:homepage" href="http://example.com/manu/">
          <span property="foaf:name">Manu</span>
        </a>
      </li>
   </ul>
</div>
```

An example JSON-LD implementation using a single context is described below.

EXAMPLE 163: Same description in JSON-LD (context shared among node objects)

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "foaf:homepage": {"@type": "@id"}
  },
  "@graph": [
    {
      "@type": "foaf:Person",
      "foaf:homepage": "http://example.com/bob/",
      "foaf:name": "Bob"
    }, {
      "@type": "foaf:Person",
      "foaf:homepage": "http://example.com/eve/",
      "foaf:name": "Eve"
    }, {
      "@type": "foaf:Person",
      "foaf:homepage": "http://example.com/manu/",
      "foaf:name": "Manu"
    }
  ]
}
```

§ B.3 Microdata

This section is non-normative.

The HTML Microdata [MICRODATA] example below expresses book information as a Microdata Work item.

EXAMPLE 164: HTML that describes a book using microdata

```
<dl itemscope
    itemtype="http://purl.org/vocab/frbr/core#Work"
    itemid="http://purl.oreilly.com/works/45U8QJGZSQKDH8N">
 <dt>Title</dt>
 <dd><cite itemprop="http://purl.org/dc/elements/1.1/title">Just a Gee
 <dt>By</dt>
 <dd><span itemprop="http://purl.org/dc/elements/1.1/creator">Wil Whea
 <dt>Format</dt>
 <dd itemprop="http://purl.org/vocab/frbr/core#realization"
     itemscope
     itemtype="http://purl.org/vocab/frbr/core#Expression"
     itemid="http://purl.oreilly.com/products/9780596007683.BOOK">
  <link itemprop="http://purl.org/dc/elements/1.1/type" href="http://p
  Print
 </dd>
 <dd itemprop="http://purl.org/vocab/frbr/core#realization"
     itemscope
     itemtype="http://purl.org/vocab/frbr/core#Expression"
     itemid="http://purl.oreilly.com/products/9780596802189.EBOOK">
  <link itemprop="http://purl.org/dc/elements/1.1/type" href="http://p
  Ebook
 </dd>
</dl>
```

Note that the JSON-LD representation of the Microdata information stays true to the desires of the Microdata community to avoid contexts and instead refer to items by their full IRI.

EXAMPLE 165: Same book description in JSON-LD (avoiding contexts)

```
[
  {
    "@id": "http://purl.oreilly.com/works/45U8QJGZSQKDH8N",
    "@type": "http://purl.org/vocab/frbr/core#Work",
    "http://purl.org/dc/elements/1.1/title": "Just a Geek",
    "http://purl.org/dc/elements/1.1/creator": "Wil Wheaton",
    "http://purl.org/vocab/frbr/core#realization":
    [
      {"@id": "http://purl.oreilly.com/products/9780596007683.BOOK"},
      {"@id": "http://purl.oreilly.com/products/9780596802189.EBOOK"}
    ]
  }, {
    "@id": "http://purl.oreilly.com/products/9780596007683.BOOK",
    "@type": "http://purl.org/vocab/frbr/core#Expression",
    "http://purl.org/dc/elements/1.1/type": {"@id": "http://purl.oreil
  }, {
    "@id": "http://purl.oreilly.com/products/9780596802189.EBOOK",
    "@type": "http://purl.org/vocab/frbr/core#Expression",
    "http://purl.org/dc/elements/1.1/type": {"@id": "http://purl.oreil
  }
]
```

§ C. IANA Considerations

This section has been submitted to the Internet Engineering Steering Group (IESG) for review, approval, and registration with IANA.

§ application/ld+json

Type name:
application

Subtype name:
ld+json

Required parameters:
None

Optional parameters:
profile
A non-empty list of space-separated URIs identifying specific

constraints or conventions that apply to a JSON-LD document according to [RFC6906]. A profile does not change the semantics of the resource representation when processed without profile knowledge, so that clients both with and without knowledge of a profiled resource can safely use the same representation. The `profile` parameter *MAY* be used by clients to express their preferences in the content negotiation process. If the profile parameter is given, a server *SHOULD* return a document that honors the profiles in the list which it recognizes, and *MUST* ignore the profiles in the list which it does not recognize. It is *RECOMMENDED* that profile URIs are dereferenceable and provide useful documentation at that URI. For more information and background please refer to [RFC6906].

This specification defines six values for the `profile` parameter.

`http://www.w3.org/ns/json-ld#expanded`
: To request or specify expanded JSON-LD document form.

`http://www.w3.org/ns/json-ld#compacted`
: To request or specify compacted JSON-LD document form.

`http://www.w3.org/ns/json-ld#context`
: To request or specify a JSON-LD context document.

`http://www.w3.org/ns/json-ld#flattened`
: To request or specify flattened JSON-LD document form.

`http://www.w3.org/ns/json-ld#frame`
: To request or specify a JSON-LD frame document.

`http://www.w3.org/ns/json-ld#framed`
: To request or specify framed JSON-LD document form.

All other URIs starting with `http://www.w3.org/ns/json-ld` are reserved for future use by JSON-LD specifications.

Other specifications *MAY* create further structured subtypes by using `+ld+json` as a suffix for a new base subtype, as in `application/example+ld+json`. Unless defined otherwise, such subtypes use the same fragment identifier behavior as `application/ld+json`.

Other specifications may publish additional `profile` parameter URIs with their own defined semantics. This includes the ability to associate a file extension with a `profile` parameter.

When used as a media type parameter [RFC4288] in an HTTP Accept header [RFC7231], the value of the `profile` parameter *MUST* be

enclosed in quotes (") if it contains special characters such as whitespace, which is required when multiple profile URIs are combined.

When processing the "profile" media type parameter, it is important to note that its value contains one or more URIs and not IRIs. In some cases it might therefore be necessary to convert between IRIs and URIs as specified in section 3 Relationship between IRIs and URIs of [RFC3987].

Encoding considerations:
See RFC 8259, section 11.

Security considerations:
See RFC 8259, section 12 [RFC8259]

Since JSON-LD is intended to be a pure data exchange format for directed graphs, the serialization *SHOULD NOT* be passed through a code execution mechanism such as JavaScript's `eval()` function to be parsed. An (invalid) document may contain code that, when executed, could lead to unexpected side effects compromising the security of a system.

When processing JSON-LD documents, links to remote contexts and frames are typically followed automatically, resulting in the transfer of files without the explicit request of the user for each one. If remote contexts are served by third parties, it may allow them to gather usage patterns or similar information leading to privacy concerns. Specific implementations, such as the API defined in the JSON-LD 1.1 Processing Algorithms and API specification [JSON-LD11-API], may provide fine-grained mechanisms to control this behavior.

JSON-LD contexts that are loaded from the Web over non-secure connections, such as HTTP, run the risk of being altered by an attacker such that they may modify the JSON-LD active context in a way that could compromise security. It is advised that any application that depends on a remote context for mission critical purposes vet and cache the remote context before allowing the system to use it.

Given that JSON-LD allows the substitution of long IRIs with short terms, JSON-LD documents may expand considerably when processed and, in the worst case, the resulting data might consume all of the recipient's resources. Applications should treat any data with due skepticism.

As JSON-LD places no limits on the IRI schemes that may be used, and

vocabulary-relative IRIs use string concatenation rather than IRI resolution, it is possible to construct IRIs that may be used maliciously, if dereferenced.

Interoperability considerations:
Not Applicable

Published specification:
http://www.w3.org/TR/json-ld

Applications that use this media type:
Any programming environment that requires the exchange of directed graphs. Implementations of JSON-LD have been created for JavaScript, Python, Ruby, PHP, and C++.

Additional information:
Magic number(s):
Not Applicable

File extension(s):
.jsonld

Macintosh file type code(s):
TEXT

Person & email address to contact for further information:
Ivan Herman <ivan@w3.org>

Intended usage:
Common

Restrictions on usage:
None

Author(s):
Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, Niklas Lindström

Change controller:
W3C

Fragment identifiers used with application/ld+json are treated as in RDF syntaxes, as per RDF 1.1 Concepts and Abstract Syntax [RDF11-CONCEPTS].

§ C.1 Examples

This section is non-normative.

The following examples illustrate different ways in which the profile parameter may be used to describe different acceptable responses.

EXAMPLE 166: HTTP Request with profile requesting an expanded document

```
GET /ordinary-json-document.json HTTP/1.1
Host: example.com
Accept: application/ld+json;profile=http://www.w3.org/ns/json-ld#expan
```

Requests the server to return the requested resource as JSON-LD in expanded document form.

EXAMPLE 167: HTTP Request with profile requesting a compacted document

```
GET /ordinary-json-document.json HTTP/1.1
Host: example.com
Accept: application/ld+json;profile=http://www.w3.org/ns/json-ld#compa
```

Requests the server to return the requested resource as JSON-LD in compacted document form. As no explicit context resource is specified, the server compacts using an application-specific default context.

EXAMPLE 168: HTTP Request with profile requesting a compacted document with a reference to a compaction context

```
GET /ordinary-json-document.json HTTP/1.1
Host: example.com
Accept: application/ld+json;profile="http://www.w3.org/ns/json-ld#flat
```

Requests the server to return the requested resource as JSON-LD in both compacted document form and flattened document form. Note that as whitespace is used to separate the two URIs, they are enclosed in double quotes (").

§ D. Open Issues

This section is non-normative.

The following is a list of issues open at the time of publication.

ISSUE 108: Consider context by reference with metadata **defer-future-version** **hr:privacy** **hr:security**

Consider context by reference with metadata.

ISSUE 191: Compact IRI expansion support for non-trivial prefix term definitions **defer-future-version** **spec:enhancement**

Compact IRI expansion support for non-trivial prefix term definitions.

ISSUE 280: language-maps don't allow separate base direction **defer-future-version**

Language-maps don't allow separate base direction.

§ E. Changes since 1.0 Recommendation of 16 January 2014

This section is non-normative.

- A context may contain a `@version` entry which is used to set the processing mode.
- An expanded term definition can now have an `@context` property, which defines a context used for values of a property identified with such a term.
- `@container` values within an expanded term definition may now include `@id`, `@graph` and `@type`, corresponding to id maps and type maps.
- An expanded term definition can now have an `@nest` property, which identifies a term expanding to `@nest` which is used for containing properties using the same `@nest` mapping. When expanding, the values of a property expanding to `@nest` are treated as if they were contained within the enclosing node object directly.
- The JSON syntax has been abstracted into an internal representation to allow for other serializations that are functionally equivalent to JSON.
- Added § 4.6.3 Node Identifier Indexing and § 4.6.4 Node Type Indexing.
- Both language maps and index maps may legitimately have an `@none` key,

but *JSON-LD 1.0* only allowed string keys. This has been updated to allow `@none` keys.

- The value for `@container` in an expanded term definition can also be an array containing any appropriate container keyword along with `@set` (other than `@list`). This allows a way to ensure that such property values will always be expressed in array form.
- In JSON-LD 1.1, terms will be chosen as compact IRI prefixes when compacting only if a simple term definition is used where the value ends with a URI gen-delim character, or if their expanded term definition contains a `@prefix` entry with the value `true`. The 1.0 algorithm has been updated to only consider terms that map to a value that ends with a URI gen-delim character.
- Values of properties where the associated term definition has `@container` set to `@graph` are interpreted as implicitly named graphs, where the associated graph name is assigned from a new blank node identifier. Other combinations include `["@container", "@id"]`, `["@container", "@index"]` each also may include `"@set"`, which create maps from the graph identifier or index value similar to index maps and id maps.

Additionally, see § F. Changes since JSON-LD Community Group Final Report.

§ F. Changes since JSON-LD Community Group Final Report

This section is non-normative.

- Lists may now have items which are themselves lists.
- Values of `@type`, or an alias of `@type`, may now have their `@container` set to `@set` to ensure that `@type` entries are always represented as an array. This also allows a term to be defined for `@type`, where the value *MUST* be a map with `@container` set to `@set`.
- The use of blank node identifiers to label properties is obsolete, and may be removed in a future version of JSON-LD, as is the support for generalized RDF Datasets.
- The vocabulary mapping can be a relative IRI reference, which is evaluated either against an existing default vocabulary, or against the document base. This allows vocabulary-relative IRIs, such as the keys of node objects, are expanded or compacted relative to the document base. (See Security Considerations in § C. IANA Considerations for a discussion

on how string vocabulary-relative IRI resolution via concatenation.)

- Added support for `"@type": "@none"` in a term definition to prevent value compaction. Define the `rdf:JSON` datatype.
- Term definitions with keys which are of the form of an IRI reference or a compact IRI *MUST NOT* expand to an IRI other than the expansion of the key itself.
- A frame may also be located within an HTML document, identified using type `application/ld+json;profile=http://www.w3.org/ns/json-ld#frame`.
- Term definitions can now be protected, to limit the ability of other contexts to override them.
- A context defined in an expanded term definition may also be used for values of `@type`, which defines a context to use for node objects including the associated type.
- By default, all contexts are propagated when traversing node objects, other than type-scoped contexts. This can be controlled using the `@propagate` entry in a local context.
- A context may contain an `@import` entry used to reference a remote context within a context, allowing `JSON-LD 1.1` features to be added to contexts originally authored for `JSON-LD 1.0`.
- A node object may include an included block, which is used to contain a set of node objects which are treated exactly as if they were node objects defined in an array including the containing node object. This allows the use of the object form of a JSON-LD document when there is more than one node object being defined, and where those node objects are not embedded as values of the containing node object.
- The `alternate` link relation can be used to supply an alternate location for retrieving a JSON-LD document when the returned document is not JSON.
- Value objects, and associated context and term definitions have been updated to support `@direction` for setting the base direction of strings.
- The processing mode is now implicitly `json-ld-1.1`, unless set explicitly to `json-ld-1.0`.
- Improve notation using IRI, IRI reference, and relative IRI reference.
- Allow further structured subtypes of `application/ld+json` by using `+ld+json` as a suffix for a new base type.
- Warn about forward-compatibility issues for terms of the form (`"@"1*ALPHA`).

§ G. Acknowledgements

This section is non-normative.

The editors would like to specially thank the following individuals for making significant contributions to the authoring and editing of this specification:

- Timothy Cole (University of Illinois at Urbana-Champaign)
- Ivan Herman (W3C Staff)
- Jeff Mixter (OCLC (Online Computer Library Center, Inc.))
- David Lehn (Digital Bazaar)
- David Newbury (J. Paul Getty Trust)
- Robert Sanderson (J. Paul Getty Trust, chair)
- Harold Solbrig (Johns Hopkins Institute for Clinical and Translational Research)
- Simon Steyskal (WU (Wirschaftsuniversität Wien) - Vienna University of Economics and Business)
- A Soroka (Apache Software Foundation)
- Ruben Taelman (Imec vzw)
- Benjamin Young (Wiley, chair)

Additionally, the following people were members of the Working Group at the time of publication:

- Steve Blackmon (Apache Software Foundation)
- Dan Brickley (Google, Inc.)
- Newton Calegari (NIC.br - Brazilian Network Information Center)
- Victor Charpenay (Siemens AG)
- Sebastian Käbisch (Siemens AG)
- Axel Polleres (WU (Wirschaftsuniversität Wien) - Vienna University of Economics and Business)
- Leonard Rosenthol (Adobe)
- Jean-Yves ROSSI (CANTON CONSULTING)
- Antoine Roulin (CANTON CONSULTING)
- Manu Sporny (Digital Bazaar)
- Clément Warnier de Wailly (CANTON CONSULTING)

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: Chris Webber, David Wood, Drummond Reed, Eleanor Joslin, Fabien Gandon, Herm Fisher, Jamie Pitts, Kim Hamilton Duffy, Niklas Lindström, Paolo Ciccarese, Paul Frazze, Paul Warren, Reto Gmür, Rob Trainer, Ted Thibodeau Jr., and Victor Charpenay.

§ H. References

§ H.1 Normative references

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

[DOM]
DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMASCRIPT]
ECMAScript Language Specification. Ecma International. URL: https://tc39.es/ecma262/

[HTML]
HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[IANA-URI-SCHEMES]
Uniform Resource Identifier (URI) Schemes. IANA. URL: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

[JSON]
The application/json Media Type for JavaScript Object Notation (JSON). D. Crockford. IETF. July 2006. Informational. URL: https://tools.ietf.org/html/rfc4627

[JSON-LD11-API]
JSON-LD 1.1 Processing Algorithms and API. Gregg Kellogg; Dave Longley; Pierre-Antoine Champin. W3C. 18 October 2019. W3C Working Draft. URL: https://www.w3.org/TR/json-ld11-api/

[JSON-LD11-FRAMING]
JSON-LD 1.1 Framing. Dave Longley; Gregg Kellogg; Pierre-Antoine Champin. W3C. 18 October 2019. W3C Working Draft. URL: https://www.w3.org/TR/json-ld11-framing/

[RDF-SCHEMA]
RDF Schema 1.1. Dan Brickley; Ramanathan Guha. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf-schema/

[RDF11-CONCEPTS]
RDF 1.1 Concepts and Abstract Syntax. Richard Cyganiak; David Wood; Markus Lanthaler. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-concepts/

[RDF11-MT]
RDF 1.1 Semantics. Patrick Hayes; Peter Patel-Schneider. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-mt/

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3986]
Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[RFC3987]
Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF. January 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc3987

[RFC4288]
Media Type Specifications and Registration Procedures. N. Freed; J. Klensin. IETF. December 2005. Best Current Practice. URL: https://tools.ietf.org/html/rfc4288

[RFC5234]
Augmented BNF for Syntax Specifications: ABNF. D. Crocker, Ed.; P. Overell. IETF. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234

[RFC6839]
Additional Media Type Structured Syntax Suffixes. T. Hansen; A. Melnikov. IETF. January 2013. Informational. URL: https://tools.ietf.org/html/rfc6839

[RFC6906]
The 'profile' Link Relation Type. E. Wilde. IETF. March 2013. Informational. URL: https://tools.ietf.org/html/rfc6906

[RFC7231]
Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R.

Fielding, Ed.; J. Reschke, Ed. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7231

[RFC8174]
Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8174

[RFC8259]
The JavaScript Object Notation (JSON) Data Interchange Format. T. Bray, Ed.. IETF. December 2017. Internet Standard. URL: https://tools.ietf.org/html/rfc8259

[RFC8288]
Web Linking. M. Nottingham. October 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8288

[UAX9]
Unicode Bidirectional Algorithm. Mark Davis; Aharon Lanin; Andrew Glass. Unicode Consortium. 4 February 2019. Unicode Standard Annex #9. URL: https://www.unicode.org/reports/tr9/tr9-41.html

[UNICODE]
The Unicode Standard. Unicode Consortium. URL: https://www.unicode.org/versions/latest/

§ H.2 Informative references

[fingerprinting-guidance]
Mitigating Browser Fingerprinting in Web Specifications. Nick Doty. W3C. 28 March 2019. W3C Note. URL: https://www.w3.org/TR/fingerprinting-guidance/

[INFRA]
Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[JCS]
JSON Canonicalization Scheme (JCS). A. Rundgren; B. Jordan; S. Erdtman. Network Working Group. February 16, 2019. Internet-Draft. URL: https://tools.ietf.org/html/draft-rundgren-json-canonicalization-scheme-05

[JSON-LD10]
JSON-LD 1.0. Manu Sporny; Gregg Kellogg; Marcus Langhaler. W3C. 16 January 2014. W3C Recommendation. URL: https://www.w3.org/TR/2014/REC-json-ld-20140116/

[JSON.API]

JSON API. Steve Klabnik; Yehuda Katz; Dan Gebhardt; Tyler Kellen; Ethan Resnick. 29 May 2015. unofficial. URL: https://jsonapi.org/format/

[ld-glossary]
Linked Data Glossary. Bernadette Hyland; Ghislain Auguste Atemezing; Michael Pendleton; Biplav Srivastava. W3C. 27 June 2013. W3C Note. URL: https://www.w3.org/TR/ld-glossary/

[LINKED-DATA]
Linked Data Design Issues. Tim Berners-Lee. W3C. 27 July 2006. W3C-Internal Document. URL: https://www.w3.org/DesignIssues/LinkedData.html

[MICRODATA]
HTML Microdata. Charles 'chaals' (McCathie) Nevile; Dan Brickley; Ian Hickson. W3C. 26 April 2018. W3C Working Draft. URL: https://www.w3.org/TR/microdata/

[RDFA-CORE]
RDFa Core 1.1 - Third Edition. Ben Adida; Mark Birbeck; Shane McCarron; Ivan Herman et al. W3C. 17 March 2015. W3C Recommendation. URL: https://www.w3.org/TR/rdfa-core/

[rfc4122]
A Universally Unique IDentifier (UUID) URN Namespace. P. Leach; M. Mealling; R. Salz. IETF. July 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc4122

[RFC7049]
Concise Binary Object Representation (CBOR). C. Bormann; P. Hoffman. IETF. October 2013. Proposed Standard. URL: https://tools.ietf.org/html/rfc7049

[RFC7946]
The GeoJSON Format. H. Butler; M. Daly; A. Doyle; S. Gillies; S. Hagen; T. Schaub. IETF. August 2016. Proposed Standard. URL: https://tools.ietf.org/html/rfc7946

[SPARQL11-OVERVIEW]
SPARQL 1.1 Overview. The W3C SPARQL Working Group. W3C. 21 March 2013. W3C Recommendation. URL: https://www.w3.org/TR/sparql11-overview/

[SRI]
Subresource Integrity. Devdatta Akhawe; Frederik Braun; Francois Marier; Joel Weinberger. W3C. 23 June 2016. W3C Recommendation. URL: https://www.w3.org/TR/SRI/

[string-meta]
Strings on the Web: Language and Direction Metadata. Addison Phillips;

Richard Ishida. W3C. 11 June 2019. W3C Working Draft. URL: https://www.w3.org/TR/string-meta/

[TriG]
RDF 1.1 TriG. Gavin Carothers; Andy Seaborne. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/trig/

[Turtle]
RDF 1.1 Turtle. Eric Prud'hommeaux; Gavin Carothers. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/turtle/

[URN]
URN Syntax. R. Moats. IETF. May 1997. Proposed Standard. URL: https://tools.ietf.org/html/rfc2141

[WEBIDL]
Web IDL. Boris Zbarsky. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

[YAML]
YAML Ain’t Markup Language (YAML™) Version 1.2. Oren Ben-Kiki; Clark Evans; Ingy döt Net. 1 October 2009. URL: http://yaml.org/spec/1.2/spec.html

↑