JSON-LD 1.0

A JSON-based Serialization for Linked Data

W3C Recommendation 16 January 2014

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

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

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

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

Copyright © 2010-2014 W3C® (MIT, ERCIM, Keio, Beihang), All Rights Reserved. W3C liability, trademark and document use rules apply.

Abstract

JSON is a useful data serialization and messaging format. This specification defines JSON-LD, 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.

Status of This Document

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

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

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

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

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

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

Table of Contents

1. Introduction
  1.1 How to Read this Document
2. Design Goals and Rationale
3. Terminology
  3.1 General Terminology
  3.2 Data Model Overview

  3.3 Syntax Tokens and Keywords
4. Conformance
5. Basic Concepts
  5.1 The Context
  5.2 IRIs
  5.3 Node Identifiers
  5.4 Specifying the Type
6. Advanced Concepts
  6.1 Base IRI
  6.2 Default Vocabulary
  6.3 Compact IRIs
  6.4 Typed Values
  6.5 Type Coercion
  6.6 Embedding
  6.7 Advanced Context Usage
  6.8 Interpreting JSON as JSON-LD
  6.9 String Internationalization
  6.10 IRI Expansion within a Context
  6.11 Sets and Lists
  6.12 Reverse Properties
  6.13 Named Graphs
  6.14 Identifying Blank Nodes
  6.15 Aliasing Keywords
  6.16 Data Indexing
  6.17 Expanded Document Form
  6.18 Compacted Document Form
  6.19 Flattened Document Form
  6.20 Embedding JSON-LD in HTML Documents
7. Data Model
8. JSON-LD Grammar
  8.1 Terms
  8.2 Node Objects
  8.3 Value Objects
  8.4 Lists and Sets
  8.5 Language Maps
  8.6 Index Maps
  8.7 Context Definitions
9. Relationship to RDF
  9.1 Serializing/Deserializing RDF
A. Relationship to Other Linked Data Formats
  A.1 Turtle
  A.2 RDFa
  A.3 Microformats
  A.4 Microdata
B. IANA Considerations
C. Acknowledgements
D. References
  D.1 Normative references
  D.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 [RFC4627]. 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 JSON object 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, if desired, for use with other Linked Data technologies like SPARQL. Developers who require any of the facilities listed above or need to serialize an RDF Graph or RDF 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, like Turtle [TURTLE]. Complete details of how JSON-LD relates to RDF are in section 9. 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 Processing Algorithms and API specification [JSON-LD-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 [RFC4627].

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.

2. 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 only need to know JSON and two keywords (`@context` and `@id`) 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 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 section 6.8 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 9. Relationship to RDF.

3. Terminology

3.1 General Terminology

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

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

array
: An array structure is represented as square brackets surrounding zero or more values. Values are separated by commas. In JSON, an array is an *ordered* sequence of zero or more values. While JSON-LD uses the same array representation as JSON, the collection is *unordered* by default. While order is preserved in regular JSON arrays, it is not in regular JSON-LD arrays unless specifically defined (see section 6.11 Sets and Lists).

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

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

true* and *false
: Values that are used to express one of two possible boolean states.

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

3.2 Data Model Overview

This section is non-normative.

Generally speaking, the data model used for JSON-LD is a labeled, directed graph. The graph contains nodes, which are connected by edges. A node is typically data such as a string, number, typed values (like dates and times) or an IRI. There is also a special class of node called a blank node, which is typically used to express data that does not have a global identifier like an IRI. Blank nodes are identified using a blank node identifier. 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 7. 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 9. Relationship to RDF.

3.3 Syntax Tokens and Keywords

JSON-LD specifies a number of syntax tokens and ***keywords*** that are a core part of the language:

`@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 section 5.1 The Context.

`@id`
: Used to uniquely identify *things* that are being described in the document with IRIs or blank node identifiers. This keyword is described in section 5.3 Node Identifiers.

`@value`
: Used to specify the data that is associated with a particular property in the graph. This keyword is described in section 6.9 String Internationalization and section 6.4 Typed Values.

`@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 section 6.9 String Internationalization.

`@type`
: Used to set the data type of a node or typed value. This keyword is described in section 6.4 Typed Values.

`@container`
: Used to set the default container type for a term. This keyword is described in section 6.11 Sets and Lists.

`@list`
: Used to express an ordered set of data. This keyword is described in section 6.11 Sets and Lists.

`@set`
: Used to express an unordered set of data and to ensure that values are always represented as arrays. This keyword is described in section 6.11 Sets and Lists.

`@reverse`
: Used to express reverse properties. This keyword is described in section 6.12 Reverse Properties.

`@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 section 6.16 Data Indexing.

`@base`
: Used to set the base IRI against which relative IRIs are resolved. This keyword is described in section 6.1 Base IRI.

`@vocab`
: Used to expand properties and values in `@type` with a common prefix IRI. This keyword is described in section 6.2 Default Vocabulary.

`@graph`
: Used to express a graph. This keyword is described in section 6.13 Named Graphs.

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

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

4. Conformance

This specification describes the conformance criteria for JSON-LD documents. This criteria is relevant to authors and authoring tool implementers. 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.

A JSON-LD document complies with this specification if it follows the normative statements in appendix 8. JSON-LD Grammar. JSON documents can be interpreted as JSON-LD by following the normative statements in section 6.8 Interpreting JSON as JSON-LD. For convenience, normative statements for documents are often phrased as statements on the properties of the document.

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL in this specification have the meaning defined in [RFC2119].

5. Basic Concepts

This section is non-normative.

JSON [RFC4627] 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 1: 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 2: Sample JSON-LD document using full IRIs instead of terms

```
{
  "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
  "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.

5.1 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 any valid string that is not a reserved JSON-LD keyword can be used as a term.

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

EXAMPLE 3: Context for the sample document in the previous section

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

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 JSON object.

When a JSON object is associated with a term, it 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 absolute or compact IRIs as keys, which is mainly used to associate type or language information with an absolute or compact IRI.

Contexts can either be directly embedded into the document or be referenced. Assuming the context document in the previous example can be retrieved at `http://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 4: Referencing a JSON-LD context

```
{
  "@context": "http://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 section 5.2 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.

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 section 6.8 Interpreting JSON as JSON-LD. It is also possible to apply a custom context using the JSON-LD API [JSON-LD-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.

EXAMPLE 5: In-line context definition

```
{
  "@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. More advanced features related to the JSON-LD Context are covered in section 6. Advanced Concepts.

5.2 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 absolute IRI or a relative IRI. An ***absolute IRI*** is defined in [RFC3987] as containing a *scheme* along with *path* and optional *query* and *fragment* segments. A ***relative IRI*** is an IRI that is relative to some other absolute IRI. In JSON-LD all relative IRIs are resolved relative to the ***base IRI***.

A string is interpreted as an IRI when it is the value of an `@id` member:

EXAMPLE 6: Values of @id are interpreted as IRI

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

Values that are interpreted as IRIs, can also be expressed as relative IRIs. For example, assuming that the following document is located at `http://example.com/about/`, the relative IRI `../` would expand to `http://example.com/` (for more information on where relative IRIs can be used, please refer to section 8. JSON-LD Grammar).

EXAMPLE 7: IRIs can be relative

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

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

EXAMPLE 8: IRI as a key

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

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

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

EXAMPLE 9: Term expansion from context definition

```
{
  "@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 10: Type coercion

```
{
  "@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 section 6.5 Type Coercion for more details about this feature.

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

1. JSON object keys that have a term mapping 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 6. Advanced Concepts.

5.3 Node Identifiers

This section is non-normative.

To be able to externally reference nodes in a 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 11: Identifying a node

```
{
  "@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 6. Advanced Concepts.

5.4 Specifying the Type

This section is non-normative.

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

EXAMPLE 12: Specifying the type for a node

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

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

EXAMPLE 13: Specifying multiple types for a node

```
{
...
  "@id": "http://example.org/places#BrewEats",
  "@type": [ "http://schema.org/Restaurant", "http://schema.org/Brewery" ],
...
}
```

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

EXAMPLE 14: Using a term to specify the type

```
{
  "@context": {
    ...
    "Restaurant": "http://schema.org/Restaurant",
    "Brewery": "http://schema.org/Brewery"
  }
  "@id": "http://example.org/places#BrewEats",
  "@type": [ "Restaurant", "Brewery" ],
  ...
}
```

NOTE

This section only covers the most basic features associated with types in JSON-LD. It is worth noting that the `@type` keyword is not only used to specify the type of a node but also to express typed values (as described in section 6.4 Typed Values) and to type coerce values (as described in section 6.5 Type Coercion). Specifically, `@type` cannot be used in a context to define a node's type. For a detailed description of the differences, please refer to section 6.4 Typed Values.

6. Advanced Concepts

JSON-LD has a number of features that provide functionality above and beyond the core functionality described above. The following section describes this advanced functionality in more detail.

6.1 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 IRIs would resolve against that IRI:

EXAMPLE 15: Use a relative IRI 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 absolute IRI, a context may define an `@base` mapping, to overwrite the base IRI for the document.

EXAMPLE 16: Setting the document base in a document

```
{
  "@context": {
    "@base": "http://example.com/document.jsonld"
  },
  "@id": "",
  "label": "Just a simple document"
}
```

Setting `@base` to null will prevent relative IRIs to be expanded to absolute IRIs.

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

6.2 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 to be used for all properties and types that do not match a term and are neither a compact IRI nor an absolute IRI (i.e., they do not contain a colon).

EXAMPLE 17: Using a common vocabulary prefix

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

If `@vocab` is used but certain keys in an object 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` member would not expand to an IRI.

EXAMPLE 18: Using the null keyword to ignore data

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

6.3 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 short hand 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 absolute IRI for the vocabulary term. For example, `foaf:name` would be expanded to the IRI `http://xmlns.com/foaf/0.1/name`.

EXAMPLE 19: Prefix expansion

```
{
  "@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 absolute 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 20: Using vocabularies

```
{
  "@context":
  {
    "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"
}
```

6.4 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 21: Expanded term definition with type coercion

```
{
  "@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 type coerced to a *dateTime* value because of the information specified in the `@context`. A JSON-LD processor will interpret the example above as follows:

Subject	Property	Value	Value Type
http://example.com/docs/1	http://purl.org/dc/terms/modified	2010-05-29T14:17:39+02:00	http://www.w3.org/2001/XMLSchema#dateTime

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

EXAMPLE 22: Expanded value with type

```
{
  "@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 different.

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 23: Example demonstrating the context-sensitivity for @type

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

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 JSON object, the `@type` keyword is expressing a value type. Otherwise, the `@type` keyword is expressing a node type. The example above expresses the following data:

Subject	Property	Value	Value Type
http://example.org/posts#TripToWestVirginia	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://schema.org/BlogPosting	-
http://example.org/posts#TripToWestVirginia	http://purl.org/dc/terms/modified	2010-05-29T14:17:39+02:00	http://www.w3.org/2001/XMLSchema

6.5 Type Coercion

This section is non-normative.

JSON-LD supports the coercion of values to particular data types. Type ***coercion*** allows someone deploying JSON-LD to coerce the incoming or outgoing values to the proper data type based on a mapping of data type IRIs to terms. Using type coercion, value representation is preserved without requiring the data type to be specified 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 keywords `@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 absolute 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 compact IRI or absolute IRI if there's a colon in the value; otherwise, it will expand the value using the active context's vocabulary mapping, if present, or by interpreting it as relative IRI. Values coerced to `@id` in contrast are expanded as compact IRI or absolute IRI if a colon is present; otherwise, they are interpreted as relative IRI.

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 24: Expanded term definition with types

```
{
  "@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/"
  ]
}
```

The example shown above would generate the following data.

Subject	Property	Value	Value Type
http://example.com/people#john	http://xmlns.com/foaf/0.1/name	John Smith	
http://example.com/people#john	http://xmlns.com/foaf/0.1/age	41	http://www.w3.org/2001/XMLSchema#integ
http://example.com/people#john	http://xmlns.com/foaf/0.1/homepage	http://personal.example.org/	IRI
		http://work.example.com/jsmith/	IRI

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

EXAMPLE 25: Term definitions using compact and absolute IRIs

```
{
  "@context":
  {
    "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 compact IRI or 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. It also allows a compact IRI (or even an absolute IRI) to be defined as something else entirely. For example, one could specify that the term `http://example.org/zoo` should expand to `http://example.org/river`, but this usage is discouraged because it would lead to a great deal of confusion among developers attempting to understand the JSON-LD document.

6.6 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.

The example shows two nodes related by a property from the first node:

EXAMPLE 26: Embedding a node object as property value of another node object

```
{
...
  "name": "Manu Sporny",
  "knows":
  {
    "@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.

6.7 Advanced Context Usage

This section is non-normative.

Section 5.1 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 at any time a JSON object is defined. The only time that one cannot express a context is inside a context definition itself. For example, a JSON-LD document may use more than one context at different points in a document:

EXAMPLE 27: Using multiple contexts

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

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

EXAMPLE 28: Scoped contexts within node objects

```
{
```

```
  "@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. 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 JSON object. 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 JSON object 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. The following example specifies an external context and then layers an embedded context on top of the external context:

EXAMPLE 29: Combining external and local contexts

```
{
  "@context": [
    "http://json-ld.org/contexts/person.jsonld",
    {
      "pic": "http://xmlns.com/foaf/0.1/depiction"
    }
  ],
  "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 are to be avoided as they might be used as keywords in future versions of JSON-LD. Terms starting with an `@` character that are not JSON-LD 1.0 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.

6.8 Interpreting JSON as JSON-LD

Ordinary JSON documents can be interpreted as JSON-LD by 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, an author MUST specify an IRI to a valid JSON-LD document in an HTTP Link Header [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation. The referenced document MUST have a top-level JSON object. The `@context` subtree 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 [RFC5988] using the `http://www.w3.org/ns/json-ld#context` link relation.

The following example demonstrates the use of an external context with an ordinary JSON document:

EXAMPLE 30: 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
```


JSON-LD 1.0 07/01/2018, 15:12

```
Link: <http://json-ld.org/contexts/person.jsonld>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"

{
  "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.

6.9 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 31: Setting the default language of a JSON-LD document

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

The example above would associate the `ja` language code with the two strings *花澄* and *科学者*. Languages codes 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 a local context as follows:

EXAMPLE 32: Clearing default language

```
{
  "@context": {
    ...
    "@language": "ja"
  },
  "name": "花澄",
  "details": {
    "@context": {
      "@language": null
    },
    "occupation": "Ninja"
  }
}
```

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

EXAMPLE 33: 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 code `ja`, *Ninja* with the language code `en`, and *Nindža* with the language code `cs`. The value of `name`, *Yagyū Muneyoshi* wouldn't be associated with any language code 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 34: 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. 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 35: 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 36: Removing language information using an expanded value

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

6.10 IRI Expansion within a Context

This section is non-normative.

In general, normal IRI expansion rules apply anywhere an IRI is expected (see section 5.2 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 37: 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 38: 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 39: 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":
    {
      "@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.

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

EXAMPLE 40: Associating context definitions with absolute IRIs

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

In order for the absolute IRI to match above, the absolute 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.

NOTE

While it is possible to define a compact IRI, or an absolute IRI to expand to some other unrelated IRI (for example, `foaf:name` expanding to `http://example.org/unrelated#species`), such usage is strongly discouraged.

The only 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 41: Illegal circular definition of terms within a context

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

6.11 Sets and Lists

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 provide an 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 42: Multiple values with no inherent order

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

The example shown above would result in the following data being generated, each relating the node to an individual value, with no inherent order:

Subject	Property	Value
http://example.org/people#joebob	http://xmlns.com/foaf/0.1/nick	joe
http://example.org/people#joebob	http://xmlns.com/foaf/0.1/nick	bob
http://example.org/people#joebob	http://xmlns.com/foaf/0.1/nick	JB

Multiple values may also be expressed using the expanded form:

EXAMPLE 43: Using an expanded form to set multiple values

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

The example shown above would generate the following data, again with no inherent order:

Subject	Property	Value	Language
http://example.org/articles/8	http://purl.org/dc/terms/title	Das Kapital	de
http://example.org/articles/8	http://purl.org/dc/terms/title	Capital	en

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 44: An ordered collection of values in JSON-LD

```
{
...
  "@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 45: Specifying that a collection is ordered in the context

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

NOTE

List of lists in the form of list objects are not allowed in this version of JSON-LD. This decision was made due to the extreme amount of added complexity when processing lists of lists.

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` or `@list` 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 section 6.18 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.

6.12 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 46: A document with children linking to their parent

```
[
  {
    "@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 47: A person and its children using a reverse property

```
{
  "@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 48: Using @reverse to define reverse properties

```
{
  "@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"
    }
  ]
}
```

6.13 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 49: Identifying and making statements about a graph

```
{
  "@context": {
    "generatedAt": {
      "@id": "http://www.w3.org/ns/prov#generatedAtTime",
      "@type": "http://www.w3.org/2001/XMLSchema#date"
    },
    "Person": "http://xmlns.com/foaf/0.1/Person",
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@id": "http://example.org/graphs/73",
  "generatedAt": "2012-04-09",
  "@graph":
  [
    {
      "@id": "http://manu.sporny.org/about#manu",
      "@type": "Person",
      "name": "Manu Sporny",
      "knows": "http://greggkellogg.net/foaf#me"
    },
    {
      "@id": "http://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/graphs/73`. 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. An alternative view of the information above is represented in table form below:

Graph	Subject	Property	Value
	http://example.org/graphs/73	http://www.w3.org/ns/prov#generatedAtTime	2012-04-09
http://example.org/graphs/73	http://manu.sporny.org/about#manu	http://www.w3.org/2001/XMLSchema#type	http://xmlns.com/foaf/0.1/Person
http://example.org/graphs/73	http://manu.sporny.org/about#manu	http://xmlns.com/foaf/0.1/name	Manu Sporny
http://example.org/graphs/73	http://manu.sporny.org/about#manu	http://xmlns.com/foaf/0.1/knows	http://greggkellogg.net/foaf#me
http://example.org/graphs/73	http://greggkellogg.net/foaf#me	http://www.w3.org/2001/XMLSchema#type	http://xmlns.com/foaf/0.1/Person
http://example.org/graphs/73	http://greggkellogg.net/foaf#me	http://xmlns.com/foaf/0.1/name	Gregg Kellogg
http://example.org/graphs/73	http://greggkellogg.net/foaf#me	http://xmlns.com/foaf/0.1/knows	http://manu.sporny.org/about#mar

When a JSON-LD document's top-level structure is an object that contains no other properties 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 50: Using @graph to explicitly express the default graph


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

In this case, embedding doesn't work as each node object references the other. This is equivalent to using multiple node objects in array and defining the `@context` within each node object:

EXAMPLE 51: Context needs to be duplicated if @graph is not used

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

6.14 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 52: Specifying a local blank node identifier

```
{
   ...
   "@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 nothing 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.

6.15 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 53: Aliasing keywords

```
{
  "@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.

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

6.16 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. Often this data does not have any 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 54: Indexing data in JSON-LD

```
{
  "@context":
  {
     "schema": "http://schema.org/",
     "name": "schema:name",
     "body": "schema:articleBody",
     "words": "schema:wordCount",
     "post": {
       "@id": "schema:blogPost",
       "@container": "@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 crude oil...",
       "words": 1539
     },
     "de": {
       "@id": "http://example.com/posts/1/de",
       "body": "Die Werte an Warenbörsen stiegen im Sog eines starken Handels von Rohöl...",
       "words": 1204
     }
  }
}
```

In the example above, the **post** term has been marked as an index map. The **en** and **de** keys will be ignored semantically, but preserved syntactically, by the JSON-LD Processor. This allows a developer to access the German version of the **post** using the following code snippet: `obj.post.de`.

The interpretation of the data above is expressed in the table below. Note how the index keys do not appear in the Linked Data below, but would continue to exist if the document were compacted or expanded (see section 6.18 Compacted Document Form and section 6.17 Expanded Document Form) using a JSON-LD processor:

Subject	Property	Value
http://example.com/	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://schema.org/Blog
http://example.com/	http://schema.org/name	World Financial News
http://example.com/	http://schema.org/blogPost	http://example.com/posts/1/en
http://example.com/	http://schema.org/blogPost	http://example.com/posts/1/de
http://example.com/posts/1/en	http://schema.org/articleBody	World commodities were up today with heavy trading of crude oil...
http://example.com/posts/1/en	http://schema.org/wordCount	1539
http://example.com/posts/1/de	http://schema.org/articleBody	Die Werte an Warenbörsen stiegen im Sog eines starken Handels von Rohöl...
http://example.com/posts/1/de	http://schema.org/wordCount	1204

6.17 Expanded Document Form

This section is non-normative.

The JSON-LD Processing Algorithms and API specification [JSON-LD-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 55: Sample JSON-LD document

```
{
   "@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 56: Expanded form for the previous example

```
[
  {
    "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`.

6.18 Compacted Document Form

This section is non-normative.

The JSON-LD Processing Algorithms and API specification [JSON-LD-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 57: Sample expanded JSON-LD document

```
[
  {
    "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 58: Sample 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 59: 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`.

6.19 Flattened Document Form

This section is non-normative.

The JSON-LD Processing Algorithms and API specification [JSON-LD-API] defines a method for *flattening* a JSON-LD document. Flattening collects all properties of a node in a single JSON object 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 60: 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 Flattening algorithm against the JSON-LD input document in the example above and using the same context would result in the following output:

EXAMPLE 61: Flattened and compacted form for the previous example

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@graph": [
    {
      "@id": "_:b0",
      "name": "Dave Longley"
    },
    {
      "@id": "http://manu.sporny.org/about#manu",
      "name": "Manu Sporny"
    },
    {
      "@id": "http://me.markus-lanthaler.com/",
      "name": "Markus Lanthaler",
      "knows": [
        { "@id": "http://manu.sporny.org/about#manu" },
        { "@id": "_:b0" }
      ]
    }
  ]
}
```

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 from.

6.20 Embedding JSON-LD in HTML Documents

This section is non-normative.

HTML script tags can be used to embed blocks of data in documents. This way, JSON-LD content can be easily embedded in HTML by placing it in a script element with the `type` attribute set to `application/ld+json`.

EXAMPLE 62: Embedding JSON-LD in HTML

```
<script type="application/ld+json">
```



```
{
  "@context": "http://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>
```

Depending on how the HTML document is served, certain strings may need to be escaped.

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 should be expanded into an RDF Dataset using the *Deserialize JSON-LD to RDF Algorithm* [JSON-LD-API].

7. 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 [RFC4627], 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 section 9. Relationship to RDF.

To ease understanding for developers unfamiliar with the RDF model, the following summary is provided:

- A ***JSON-LD document*** serializes a generalized 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 edges.
- Every ***edge*** has a direction associated with it and is labeled with an IRI or a blank node identifier. Within the JSON-LD syntax these edge labels are called ***properties***. Whenever practical, an edge SHOULD be labeled with an IRI.
- Every ***node*** is an IRI, a blank node, a JSON-LD value, or a list.
- 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 edge to any other node.
- 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 MAY be 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 typed value with type `xsd:string`), a number (numbers with a non-zero fractional part, i.e., the result of a modulo-1 operation, 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].
- 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.

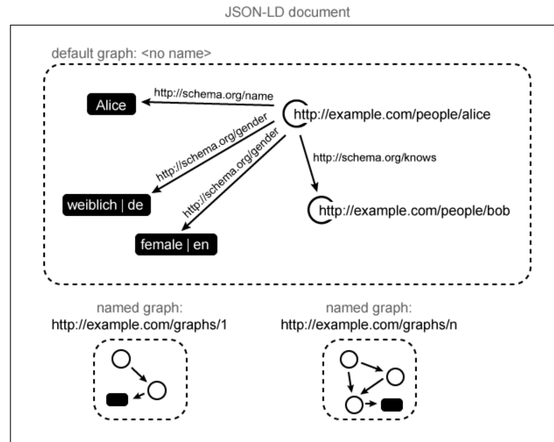

Figure 1: An illustration of the data model.

8. JSON-LD Grammar

This appendix restates the syntactic conventions described in the previous sections more formally.

A JSON-LD document MUST be a valid JSON document as described in [RFC4627].

A JSON-LD document MUST be a single node object or an array whose elements are each node objects at the top level.

In contrast to JSON, in JSON-LD the keys in objects MUST be unique.

> NOTE
>
> JSON-LD allows keywords to be aliased (see section 6.15 Aliasing Keywords for details). Whenever a keyword is discussed in this grammar, the statements also apply to an alias for that keyword. 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.

8.1 Terms

A ***term*** is a short-hand string that expands to an IRI or a blank node identifier.

A term MUST NOT equal any of the JSON-LD keywords.

To avoid forward-compatibility issues, a term SHOULD NOT start with an `@` character 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 section 5.1 The Context and section 5.2 IRIs for further discussion on mapping terms to IRIs.

8.2 Node Objects

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

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

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 JSON object. All keys which are not IRIs, compact IRIs, terms valid in the active context, or one of the following keywords MUST be ignored when processed:

- `@context`,
- `@id`,
- `@graph`,
- `@type`,
- `@reverse`, or
- `@index`

If the node object contains the `@context` key, its value MUST be null, an absolute IRI, a relative IRI, a context definition, or an array composed of any of these.

If the node object contains the `@id` key, its value MUST be an absolute IRI, a relative IRI, or a compact IRI (including blank node identifiers). See section 5.3 Node Identifiers, section 6.3 Compact IRIs, and section 6.14 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 contains an `@id` keyword, its value is used as the label of a named graph. See section 6.13 Named Graphs for further discussion on `@graph` values. As a special case, if a JSON object contains no keys other than `@graph` and `@context`, and the JSON object is the root of the JSON-LD document, the JSON object is not treated as a node object; this is used as a way of defining node definitions 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 absolute IRI, a relative IRI, a compact IRI (including blank node identifiers), a term defined in the active context expanding into an absolute IRI, or an array of any of these. See section 5.4 Specifying the Type for further discussion on `@type` values.

If the node object contains the `@reverse` key, its value MUST be a JSON object containing members representing reverse properties. Each value of such a reverse property MUST be an absolute IRI, a relative IRI, a compact IRI, a blank node identifier, a node object or an array containing a combination of these.

If the node object contains the `@index` key, its value MUST be a string. See section 6.16 Data Indexing for further discussion on `@index` values.

Keys in a node object that are not keywords MAY expand to an absolute IRI using the active context. The values associated with keys that expand to an absolute IRI MUST be one of the following:

- string,
- number,
- true,
- false,
- null,
- node object,
- value object,
- list object,
- set object,
- an array of zero or more of the possibilities above,
- a language map, or
- an index map

8.3 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.

A value object MUST be a JSON object containing the `@value` key. It MAY also contain an `@type`, an `@language`, an `@index`, or an `@context` key but MUST NOT contain both an `@type` and an `@language` key at the same time. A value object MUST NOT contain any other keys that expand to an absolute IRI or keyword.

The value associated with the `@value` key MUST be either a string, a number, true, false or null.

The value associated with the `@type` key MUST be a term, a compact IRI, an absolute IRI, a relative IRI, 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 `@index` key MUST be a string.

See section 6.4 Typed Values and section 6.9 String Internationalization for more information on value objects.

8.4 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 compact document form. This simplifies post-processing of the data as the data is always in a deterministic form.

A ***list object*** MUST be a JSON object that contains no keys that expand to an absolute IRI or keyword other than `@list`, `@context`, and `@index`.

A ***set object*** MUST be a JSON object that contains no keys that expand to an absolute IRI or keyword other than `@list`, `@context`, 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 section 6.11 Sets and Lists for further discussion on sets and lists.

8.5 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`. The keys of a language map 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 section 6.9 String Internationalization for further discussion on language maps.

8.6 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`. The values of the members 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 section 6.16 Data Indexing for further information on this topic.

8.7 Context Definitions

A ***context definition*** defines a local context in a node object.

A context definition MUST be a JSON object whose keys MUST either be terms, compact IRIs, absolute IRIs, or the keywords `@language`, `@base`, and `@vocab`.

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 `@base` key, its value MUST be an absolute IRI, a relative IRI, or null.

If the context definition has an `@vocab` key, its value MUST be a absolute IRI, a compact IRI, a blank node identifier, a term, or null.

The value of keys that are not keywords MUST be either an absolute IRI, a compact IRI, a term, a blank node identifier, a keyword, null, or an 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 JSON object composed of zero or more keys from `@id`, `@reverse`, `@type`, `@language` or `@container`. An expanded term definition SHOULD NOT contain any other keys.

If an expanded term definition has an `@reverse` member, it MUST NOT have an `@id` member at the same time. If an `@container` member exists, its value MUST be null, `@set`, or `@index`.

If the term being defined is not a compact IRI or absolute IRI and the active context does not have an `@vocab` mapping, the expanded term definition MUST include the `@id` key.

If the expanded term definition contains the `@id` keyword, its value MUST be null, an absolute IRI, a blank node identifier, a compact IRI, a term, or a keyword.

If the expanded term definition contains the `@type` keyword, its value MUST be an absolute IRI, a compact IRI, a term, null, or the one of the keywords `@id` 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`, or be null. 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.

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 section 5.1 The Context for further discussion on contexts.

9. 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 are part of the data model whereas in RDF they are part of a vocabulary, namely [RDF11-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 Processing Algorithms and API specification [JSON-LD-API] defines the conversion rules between JSON's native data types and RDF's counterparts to allow round-tripping.

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].

For authors and developers working with blank nodes as properties when deserializing to RDF, three potential approaches are suggested:

- If the author is not yet ready to commit to a stable IRI, the property should be mapped to an IRI that is documented as unstable.
- If the developer wishes to use blank nodes as properties and also wishes to interpret the data as a generalized RDF Dataset, there is an option, *produce generalized RDF*, in the Deserialize JSON-LD to RDF algorithm [JSON-LD-API] to do so. Note that a generalized RDF Dataset is an extension of RDF; it does not conform to the RDF standard.
- If the author or developer wishes to use blank nodes as properties and wishes to interpret the data as a standard (non-generalized) RDF Dataset, it is possible to losslessly interpret JSON-LD as RDF by transforming blank nodes used as properties to IRIs, by minting new "Skolem IRIs" 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 Processing Algorithms and API specification [JSON-LD-API].

Even though JSON-LD serializes generalized RDF Datasets, it can also be used as a RDF 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 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.

9.1 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 Processing Algorithms and API specification [JSON-LD-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 section 6.17 Expanded Document Form.
2. Flatten the document, which turns the document into an array of node objects. Flattening is discussed further in section 6.19 Flattened Document Form.
3. Turn each node object into a series of RDF triples.

For example, consider the following JSON-LD document in compact form:

EXAMPLE 63: 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 64: 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 RDF triples. This can be expressed in Turtle as follows:

EXAMPLE 65: Turtle representation of expanded/flattened document

```
_:b0 <http://xmlns.com/foaf/0.1/name> "Dave Longley" .

<http://manu.sporny.org/about#manu> <http://xmlns.com/foaf/0.1/name> "Manu Sporny" .

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

A. 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, Microformats, 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.

A.1 Turtle

This section is non-normative.

The following are examples of transforming RDF expressed in Turtle [TURTLE] into JSON-LD.

Prefix definitions

This section is non-normative.

The JSON-LD context has direct equivalents for the Turtle `@prefix` declaration:

EXAMPLE 66: 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 67: 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/" }
}
```

Embedding

Both Turtle and JSON-LD allow embedding, although Turtle only allows embedding of blank nodes.

EXAMPLE 68: 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 69: 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"
  }
}
```

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 Processing Algorithms and API specification [JSON-LD-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 70: 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 71: 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 .
```

Lists

Both JSON-LD and Turtle can represent sequential lists of values.

EXAMPLE 72: 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 73: 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" ]
  }
}
```

A.2 RDFa

This section is non-normative.

The following example describes three people with their respective names and homepages in RDFa [RDFA-CORE].

EXAMPLE 74: RDFa fragment that describes three people

```
<div prefix="foaf: http://xmlns.com/foaf/0.1/">
   <ul>
      <li typeof="foaf:Person">
        <a rel="foaf:homepage" href="http://example.com/bob/" property="foaf:name">Bob</a>
      </li>
      <li typeof="foaf:Person">
        <a rel="foaf:homepage" href="http://example.com/eve/" property="foaf:name">Eve</a>
      </li>
      <li typeof="foaf:Person">
        <a rel="foaf:homepage" href="http://example.com/manu/" property="foaf:name">Manu</a>
      </li>
   </ul>
</div>
```

An example JSON-LD implementation using a single context is described below.

EXAMPLE 75: Same description in JSON-LD (context shared among node objects)

```
{
  "@context":
  {
    "foaf": "http://xmlns.com/foaf/0.1/"
  },
  "@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"
    }
  ]
}
```

A.3 Microformats

This section is non-normative.

The following example uses a simple Microformats hCard example to express how Microformats [MICROFORMATS] are represented in JSON-LD.

EXAMPLE 76: HTML fragment with a simple Microformats hCard

```
<div class="vcard">
 <a class="url fn" href="http://tantek.com/">Tantek Çelik</a>
</div>
```

The representation of the hCard expresses the Microformat terms in the context and uses them directly for the `url` and `fn` properties. Also note that the Microformat to JSON-LD processor has generated the proper URL type for `http://tantek.com/`.

EXAMPLE 77: Same hCard representation in JSON-LD

```
{
  "@context":
  {
    "vcard": "http://microformats.org/profile/hcard#vcard",
    "url":
    {
      "@id": "http://microformats.org/profile/hcard#url",
      "@type": "@id"
    },
    "fn": "http://microformats.org/profile/hcard#fn"
  },
  "@type": "vcard",
  "url": "http://tantek.com/",
  "fn": "Tantek Çelik"
}
```

A.4 Microdata

This section is non-normative.

The HTML Microdata [MICRODATA] example below expresses book information as a Microdata Work item.

EXAMPLE 78: HTML fragments 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/terms/title">Just a Geek</cite></dd>
 <dt>By</dt>
 <dd><span itemprop="http://purl.org/dc/terms/creator">Wil Wheaton</span></dd>
 <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/terms/type" href="http://purl.oreilly.com/product-types/BOOK">
  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/terms/type" href="http://purl.oreilly.com/product-types/EBOOK">
  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 79: 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/terms/title": "Just a Geek",
    "http://purl.org/dc/terms/creator": "Whil Wheaton",
    "http://purl.org/vocab/frbr/core#realization":
    [
      "http://purl.oreilly.com/products/9780596007683.BOOK",
      "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/terms/type": "http://purl.oreilly.com/product-types/BOOK"
  },
  {
    "@id": "http://purl.oreilly.com/products/9780596802189.EBOOK",
    "@type": "http://purl.org/vocab/frbr/core#Expression",
    "http://purl.org/dc/terms/type": "http://purl.oreilly.com/product-types/EBOOK"
  }
]
```

B. 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 a non-empty list of space-separated URIs identifying specific constraints or conventions that apply to a JSON-LD document according [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 are recognized by the server. 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 three values for the `profile` parameter. To request or specify expanded JSON-LD document form, the URI `http://www.w3.org/ns/json-ld#expanded` SHOULD be used. To request or specify compacted JSON-LD document form, the URI `http://www.w3.org/ns/json-ld#compacted` SHOULD be used. To request or specify flattened JSON-LD document form, the URI `http://www.w3.org/ns/json-ld#flattened` SHOULD be used. Please note that, according [HTTP11], the value of the `profile` parameter has to be enclosed in quotes (`"`) because it contains special characters and, if multiple profiles are combined, whitespace.

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 6839, section 3.1.
Security considerations:
See [RFC4627]

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 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 Processing Algorithms and API specification [JSON-LD-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.

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:
Manu Sporny <msporny@digitalbazaar.com>
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. Acknowledgements

This section is non-normative.

The authors would like to extend a deep appreciation and the most sincere thanks to Mark Birbeck, who contributed foundational concepts to JSON-LD via his work on RDFj. JSON-LD uses a number of core concepts introduced in RDFj, such as the context as a mechanism to provide an environment for interpreting JSON data. Mark had also been very involved in the work on RDFa as well. RDFj built upon that work. JSON-LD exists because of the work and ideas he started nearly a decade ago in 2004.

A large amount of thanks goes out to the JSON-LD Community Group participants who worked through many of the technical issues on the mailing list and the weekly telecons - of special mention are François Daoust, Stéphane Corlosquet, Lin Clark, and Zdenko 'Denny' Vrandečić.

The work of David I. Lehn and Mike Johnson are appreciated for reviewing, and performing several early implementations of the specification. Thanks also to Ian Davis for this work on RDF/JSON.

Thanks to the following individuals, in order of their first name, for their input on the specification: Adrian Walker, Alexandre Passant, Andy Seaborne, Ben Adida, Blaine Cook, Bradley Allen, Brian Peterson, Bryan Thompson, Conal Tuohy, Dan Brickley, Danny Ayers, Daniel Leja, Dave Reynolds, David Booth, David I. Lehn, David Wood, Dean Landolt, Ed Summers, elf Pavlik, Eric Prud'hommeaux, Erik Wilde, Fabian Christ, Jon A. Frost, Gavin Carothers, Glenn McDonald, Guus Schreiber, Henri Bergius, Jose María Alvarez Rodríguez, Ivan Herman, Jack Moffitt, Josh Mandel, KANZAKI Masahide, Kingsley Idehen, Kuno Woudt, Larry Garfield, Mark Baker, Mark MacGillivray, Marko Rodriguez, Marios Meimaris, Matt Wuerstl, Melvin Carvalho, Nathan Rixham, Olivier Grisel, Paolo Ciccarese, Pat Hayes, Patrick Logan, Paul Kuykendall, Pelle Braendgaard, Peter Patel-Schneider, Peter Williams, Pierre-Antoine Champin, Richard Cyganiak, Roy T. Fielding, Sandro Hawke, Simon Grant, Srecko Joksimovic, Stephane Fellah, Steve Harris, Ted Thibodeau Jr., Thomas Steiner, Tim Bray, Tom Morris, Tristan King, Sergio Fernández, Werner Wilms, and William Waites.

D. References

D.1 Normative references

[BCP47]
A. Phillips; M. Davis. *Tags for Identifying Languages*. September 2009. IETF Best Current Practice. URL: http://tools.ietf.org/html/bcp47
[RDF11-CONCEPTS]
Richard Cyganiak, David Wood, Markus Lanthaler, Editors. *RDF 1.1 Concepts and Abstract Syntax*. 9 January 2014. W3C Proposed Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PR-rdf11-concepts-20140109/. The latest edition is available at http://www.w3.org/TR/rdf11-concepts/
[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt
[RFC3987]
M. Dürst; M. Suignard. *Internationalized Resource Identifiers (IRIs)*. January 2005. RFC. URL: http://www.ietf.org/rfc/rfc3987.txt
[RFC4627]
D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON) (RFC 4627)*. July 2006. RFC. URL: http://www.ietf.org/rfc/rfc4627.txt
[RFC5988]
M. Nottingham. *Web Linking*. October 2010. Internet RFC 5988. URL: http://www.ietf.org/rfc/rfc5988.txt

D.2 Informative references

[HTTP11]
R. Fielding et al. *Hypertext Transfer Protocol - HTTP/1.1*. June 1999. RFC. URL: http://www.ietf.org/rfc/rfc2616.txt
[JSON-LD-API]
Markus Lanthaler, Gregg Kellogg, Manu Sporny, Editors. *JSON-LD 1.0 Processing Algorithms and API*. 16 January 2014. W3C Recommendation. URL: http://www.w3.org/TR/json-ld-api/
[LINKED-DATA]
Tim Berners-Lee. *Linked Data*. Personal View, imperfect but published. URL: http://www.w3.org/DesignIssues/LinkedData.html
[MICRODATA]
Ian Hickson, Editor. *HTML Microdata*. 29 October 2013. W3C Working Group Note. URL: http://www.w3.org/TR/2013/NOTE-microdata-20131029/
[MICROFORMATS]
Microformats. URL: http://microformats.org
[RDF11-MT]
Patrick J. Hayes, Peter F. Patel-Schneider, Editors. *RDF 1.1 Semantics*. 9 January 2014. W3C Proposed Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PR-rdf11-mt-20140109/. The latest edition is available at http://www.w3.org/TR/rdf11-mt/
[RDF11-SCHEMA]
Dan Brickley; R.V. Guha, Editors. *RDF Schema 1.1*. 9 January 2014. W3C Proposed Edited Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PER-rdf-schema-20140109/. The latest edition is available at http://www.w3.org/TR/rdf-schema/
[RDFA-CORE]
Ben Adida; Mark Birbeck; Shane McCarron; Ivan Herman et al. *RDFa Core 1.1 - Second Edition*. 22 August 2013. W3C Recommendation. URL: http://www.w3.org/TR/rdfa-core/
[RFC3986]
T. Berners-Lee; R. Fielding; L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax (RFC 3986)*. January 2005. RFC. URL: http://www.ietf.org/rfc/rfc3986.txt
[RFC6839]
Tony Hansen, Alexey Melnikov. *Additional Media Type Structured Syntax Suffixes*. January 2013. Internet RFC 6839. URL: http://www.ietf.org/rfc/rfc6839.txt
[RFC6906]
Erik Wilde. *The 'profile' Link Relation Type*. March 2013. Internet RFC 6906. URL: http://www.ietf.org/rfc/rfc6906.txt
[TURTLE]
Eric Prud'hommeaux, Gavin Carothers, Editors. *RDF 1.1 Turtle: Terse RDF Triple Language*. 9 January 2014. W3C Proposed Recommendation (work in progress). URL: http://www.w3.org/TR/2014/PR-turtle-20140109/. The latest edition is available at http://www.w3.org/TR/turtle/