JSON-LD Primer

Thursday, February 2, 2017 11:38 AM

@ http://json-ld.org/primer/latest/

JSON
JSON (JavaScript Object Notation) [*RFC4627*] is a simple way to express objects in a syntax compatible with JavaScript.

Linked Data
Linked Data is a term used to describe data relationships through interconnected documents. The basic tenets are described in the [*LINKED-DATA*] design note by Tim Berners-Lee:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information.
4. Include links to other URIs so that they can discover more things.

Structured Data
Structured Data describes general means of describing inter-relationships between entities in a way that can be described using a graph or tree structure. JSON is an example of a grammar for describing Structured Data. In the context of JSON-LD, Structured Data refers to JSON objects in which the subject may not have an IRI, or where the IRI is not dereferencable, or does not retrieve a representation of the object to which it refers.

JSON Object
From [*RFC4627*]:

> An object structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members). A name is a string. A single colon comes after each name, separating the name from the value. A single comma separates a value from a following name. The names within an object SHOULD be unique.

1. ***Linked Data*** is a set of documents, each containing a representation of a linked data graph.
2. A ***linked data graph*** is an unordered labeled directed graph, where nodes are subjects or objects, and edges are properties.
3. A ***subject*** is any node in a linked data graph with at least one outgoing edge.
4. A subject SHOULD be labeled with an IRI.
5. A ***property*** is an edge of the linked data graph.
6. A property SHOULD be labeled with an IRI.
7. An ***object*** is a node in a linked data graph with at least one incoming edge.
8. An object MAY be labeled with an IRI.
9. An IRI that is a label in a linked data graph SHOULD be dereferencable to a Linked Data document describing the labeled subject, object or property.
10. A ***literal*** is an object with a label that is not an IRI

The following are taken to be requirements and principles for creating Linked Data in JSON.

1. A JSON-LD document MUST be able to express a linked data graph.
2. A JSON-LD document MUST be a valid JSON document.
3. All JSON constructs MUST have semantic meaning in a JSON-LD document: JSON objects, arrays, numbers, strings and the literal names false, and true.
4. A subject is defined using a designated name/value pair of a JSON object.
5. There MUST be a way to label a JSON object with an IRI.
6. There MAY be a way to reference an un-labeled JSON object that does not have a direct child relationship.
7. JSON object name/value SHOULD be used to describe property-object relationships.
8. A property SHOULD resolve to an absolute IRI.
9. Un-coerced strings represent literal objects.
10. Coerced strings MAY represent IRIs.
11. There SHOULD be a way to associate a datatype IRI with a literal JSON value.
12. JSON boolean and numbers values represent specific datatyped literals.
13. The literal value of JSON null is undefined.
14. A JSON array MAY be used to associate multiple objects with a subject through a common property.
15. Without explicit syntactic support, JSON arrays MUST NOT be interpreted as defining an object ordering.
16. A JSON-LD document SHOULD be able to express an ordered list of objects.

Applications processing linked data can do so in a number of ways. One is to access the raw triples. This is very flexible but often is cumbersome. Another is to use some form of graph processing API, of which many variations exist. Another is to create a tree view from a portion of the graph.

JSON-LD combines these features. At a low-level it provides a standardized way to represent linked data in JSON. However, it has been found that much of the lined data that is practically processed in JSON can be converted into tree structures that are more natural to handle. Many programming languages even offer native natural access to tree-like structures and no special APIs are required.

Converting linked data graphs to easily accessible tree structures solves one problem of linked data processing. Another is that in many cases the components of a triple are represented as IRIs. Using long IRIs everywhere to access data is very flexible and has many benefits but from a programming view point is rather unwieldy.

JSON-LD provides the ability to add "context" to the data and "coerce" values into forms that are easier to process. In fact, the end result of good JSON-LD usage is data structures that look like simple JSON but are in fact full linked data graphs. This provides a good deal of power for application developers to convert linked data the are processing into easily manipulatable JSON data.

Example : https://www.1and1.com/digitalguide/websites/website-creation/tutorial-json-ld-with-schemaorg/

4. Framing

Many applications wish to access data as if it is a simple tree structure. This is natural in many programming languages and often comes for free without a specialized API. The flexibility of the JSON-LD format and linked data it represents does come with a problem.

5. Normalization

Normalization is the process of converting a data structure into a standard format using a common algorithm. Any implementation of a normalization algorithm should produce the exact same output for the same input. The normalized form has a number of uses:

- regular structure for simple applications
- graph comparison
- hashing
- cryptographic digital signatures

The JSON-LD normalization process will convert an arbitrary JSON-LD data structure to a standard normalized JSON-LD data structure. The structure is only normalized at the data structure level. Many applications that use normalization will also have the need to serialize the output. This step will often also add the step of lexicographically sorting the JSON members by name. Serialized output will then be exactly the same across implementations for a given input.