RDF/JSON: A Specification for serialising RDF in JSON

Keith Alexander

Talis Information Ltd, Knights Court, Solihull Parkway, Birmingham Business Park, B37 7YB, United Kingdom
keith.alexander@talis.com

Abstract. This paper outlines the design of a data structure for RDF serialisable in JSON, and describes why it is significant for developing with RDF and scripting languages.

What is JSON

JSON (JavaScript Object Notation) is billed as a "lightweight data-interchange format"[1]. It is a subset of Javascript, using its "'object literal notation"' and as such, can simply be evaled in javascript, and parsed with little effort in most other languages. As a data-interchange format, it has become increasingly popular as an alternative to XML. Its advocates generally cite its human-readability and the ease of parsing as the main advantages. However, the key feature, I believe, is that JSON data structures translate transparently into the native data structures universal to almost all programming languages used today.

RDF in JSON?

At Talis, we provide a Semantic Web Technologies Platform, of which RDF is a core component. We want to be able to provide the option of requesting RDF output from our services (for example, from SPARQL CONSTRUCTs and DESCRIBEs) as JSON in order to make the data accessible in scripting language environments without the overhead of an RDF/XML parser.

There are of course, other 'light-weight' serialisations, such as Turtle[2], for which parsers exist in the most of the popular web scripting languages. Turtle, in fact, probably has the advantage of greater human-readability over any JSON or XML serialisation. However, consumers of Turtle still need to parse it into triples, and represent those triples in some kind of data structure. This is the niche we see for RDF in JSON: a well-defined and commonly understood data structure for representing RDF data that can be transparently serialised in JSON and passed easily between server and client.

We found several existing approaches to the problem of representing RDF data in JSON:

Approaches to Representing RDF in JSON

Flat Triples Approach

These approaches represent the components of RDF triples in a readily understandable, and easily serialisable fashion. The disadvantage is that it is difficult to access the data without an RDF model API of some kind to iterate over the triples and pick out a property for you.

ARC[3] v.1's RDF Parser Output

```
{
  "data" : [
        {
          "s" : { "type" : "uri" , "uri" : "http://example.org/about" } ,
          "p" : "http://purl.org/dc/elements/1.1/creator",
          "o" : { "type" : "literal", "val" : "Anna Wilder" }
        } ,
        {
          "s" : { "type" : "uri" , "uri" : "http://example.org/about" } ,
          "p" : "http://purl.org/dc/elements/1.1/title",
          "o" : { "type" : "literal", "val" : "Anna's Homepage", "lang" :
"en" }
        }
     ]
}
```

Resource-oriented Approach

This approach groups properties of resources together.
JDIL[4]

```
       {
         "@namespaces": {
            "dc":"http://purl.org/dc/elements/1.1/",
            "rss":"http://purl.org/rss/1.0/",
            "georss":"http://www.georss.org/georss/"
         },
         "@type":"rss:channel",
         "rss:items": [
            { "@type":"rss:item",
              "rss:title":"A visit to Astoria",
              "rss:description":"sample description",
              "dc:coverage":{
                 "@id":"a0",
                 "dc:title":"Astoria, Oregon, US",
                 "georss:point":"46.18806  -123.83"
               }
             },
          ]
        }
```

This is a more transparently RDF-like format, and it looks fairly natural to read and write by hand. However, several things make it difficult for scripting with:

- Namespace prefixes are used. This makes it pretty to read and write for humans, but difficult for scripting consumption, since the script will in most cases not be able to know beforehand which prefixes will be used for which namespaces, and hence will have to resolve all 'Qnames'[5] to full URIs.
- Resources can be nested - this makes it hard to find individual resources, as you do not know where they are stored in the hierarchy
- Resource identifiers are JSON object properties - so to find a particular resource, you must iterate over all the resources, checking for an "@id" key

Talis's RDF/JSON specification

RDF/JSON represents a set of RDF triples as a series of nested data structures. Each unique subject in the set of triples is represented as a key in JSON object (also known as associative array, dictionary or hash table). The value of each key is a object whose keys are the URIs of the properties associated with each subject. The value of each property key is an array of objects representing the value of each property.

Blank node subjects are named using the Turtle a string conforming to the nodeID production in Turtle. For example: `_:A1`

A triple (subject S, predicate P, object O) is encoded in the following structure:

```
{ "S" : { "P" : [ O ] } }
```

The object of the triple O is represented as a further JSON object with the following keys:

type
: one of 'uri', 'literal' or 'bnode' (**required** and must be lowercase)

value
: the lexical value of the object (**required**, full URIs should be used, not qnames)

lang
: the language of a literal value (optional but if supplied it must not be empty)

datatype
: the datatype URI of the literal value (optional)

The 'lang' and 'datatype' keys should only be used if the value of the 'type' key is "literal".

For example, the following triple:

```
<http://example.org/about> <http://purl.org/dc/elements/1.1/title>
"Anna's Homepage" .
```

can be encoded in RDF/JSON as:

```
{
  "http://example.org/about" :
    {
       "http://purl.org/dc/elements/1.1/title": [ { "type" : "literal" ,
"value" : "Anna's Homepage." } ]
    }
}
```

Usage Examples

Accessing a Resource in a Graph

```
var resource = data['http://example.org/about'];
// resource is an object containing
// all the properties belonging to http://example.org/about
```

Accessing the title of a Resource

```
var title = data['http://example.org/about']['http://purl.org/dc/
elements/1.1/title'][0]['value'];
```

Iterating over a Graph

```
for(var uri in data){

  for(var property in data[uri]){

     for(var i=0; i<data[uri][property].length; i++ ){
          var s = uri;
          var p = property;
          var o = data[uri][property][i]['value'];
          var o_type = data[uri][property][i]['type'];
          var o_lang = data[uri][property][i]['lang'];
          var o_datatype = data[uri][property][i]['datatype'];
     }

  }

}
```

Design Constraints

We decided what we needed was a JSON structure for RDF that:

- expresses the whole RDF model (we didn't want information loss just because a consumer had requested JSON rather than XML or turtle)
- requires as little processing / control structures for common tasks, as possible.

As I mentioned above, the most compelling advantage of JSON is that it translates directly into universal data structures. So what we tried to do is come up with the most useful default structure for RDF data.

Design Decisions

An important goal was that the structure should be as consistent and predictable as possible, making it easier for scripts to handle any data encoded as RDF/JSON.

- **Resource-centric structure**. There are various options for how to arrange RDF data into a structure. The most common, from those that we found, are "triple" structures, where the basic unit is the triple, and "resource" structures, where a resource's properties are gathered up in one unit. Technically, JDIL's structure does not strictly enforce this unification of a resource's properties: since the resource identifier is a property of, rather than a key to, the resource object, it is possible to spread the resource description across multiple object structures. It would, of course, also be possible to structure the data according to one of the other basic components of RDF - by property URI, or Object, for example - or to introduce named graphs as the top level grouping for the data, and there are certainly use cases for which such structures would be better suited. However a resource-centric structure makes things easier for most common tasks (such as rendering descriptions in HTML, and accessing particular properties of a given resource.
- **No nesting of resources**. Although nesting resources can sometimes seem intuitive when authoring by hand, it opens the serialisation up to a huge amount of variability, making it more tedious for a script consuming the data to find a particular resource in the graph.
- **Resource Descriptions should be identified by their URIs**. In contrast to the JDIL format, URIs are not "properties" of a resource description, but the key to the resource description (which consists of a hash object of properties and their values). In conjunction with no nesting, this means that you access the properties of any given resource in the rdf/json object simply by

using the URI (or node ID) as a key. eg: `data['http://example.org/#foo']`. It also enforces that all known properties of a resource are contained in the same

- **No prefixes**. While declaring prefixes to stand for namespace URIs seems natural enough in a hand-authoring environment, and it can make the resulting document easier to read, we realised it would also make it more difficult for consumers to use, since, in most situations the consumer will not be able to know which prefixes will be used for which
- **Property URIs are keys to an array of values** - even if there is only one 'object' in the array. If you knew in advance that a particular property only had one value, it might be simpler to access it directly, rather than as the single item of an array. However, the (we think) more common case is not knowing whether there is one or more than one values of a property, in which case it is simpler to always deal with an array of objects.

Implementations

Our RDF/JSON specification already has several implementations, including, at the time of writing:

- ARC PHP RDF toolkit http://arc.semsol.org/ (2008)
- Drupal RDF Module http://drupal.org/handbook/modules/rdf (2008)
- Raptor http://librdf.org/raptor/RELEASE.html#rel1_4_17 (2008)
- Triplr http://triplr.org (2008)

Benefits of Wider Adoption

Since JSON is not just a text format, but a "serialised data structure", RDF libraries can support it, not just as format to parse from or output to, but also internally, as a data structure that can be passed to, and returned by functions and methods, as ARC, and the Drupal RDF module do.

There are a few advantages to converging on a common data structure for RDF in code. One is that it becomes easier to pass data between components; if I use library A to spider some RDF from the web, and I need to use another library B to do some OWL reasoning, say, ordinarily, I would have to reserialise the RDF data with library A so that I can pass it to library B, but if both libraries use the same data structure, I can pass the data directly, without the parsing, re-serialising, and parsing again. This in turn may mean that we don't need such monolithic RDF API libraries, and can encourage the development of more modular components.

Of course, our proposed RDF/JSON structure is not optimal for every task, but (given the advantages of interoperability and familiarity) in such cases where the internal structure would mainly be arbitrarily different, I would like to encourage RDF library developers to support our RDF/JSON structure internally as well as externally.

The other big advantage for the RDF developer standardising on a common data structure is that there is less mental overhead once you are familiar with that structure. You can use the same algorithms for processing RDF in PHP, Javascript, Ruby, Perl and Python, and between library components that use that structure too.

By using and supporting RDF/JSON, semantic web developers can encourage a code ecosystem that will make programming with RDF easier across languages and libraries to wider audience of developers.

References

1. JSON http://json.org/ (2008)
2. Turtle http://www.dajobe.org/2004/01/turtle/ (2007)
3. ARC PHP_RDF toolkit http://arc.semsol.org/ (2008)
4. JDIL http://www.jdil.org/ (2007)
5. Namespaces in XML http://www.w3.org/TR/REC-xml-names/#dt-qualname (2006)
6. Drupal RDF Module http://drupal.org/handbook/modules/rdf (2008)
7. Raptor http://librdf.org/raptor/RELEASE.html#rel1_4_17 (2008)
8. Triplr http://triplr.org (2008)
9. RDF/JSON Specification http://n2.talis.com/wiki/RDF_JSON_Specification (2008)
10. RDF/JSON Brainstorming http://n2.talis.com/wiki/RDF_JSON_Brainstorming (2008)
11. Beckett, Dave: <XML/> without the X: The return of Template:Textual markup http://www.dajobe.org/talks/200705-textual/ (2008)
12. Feigenbaum, Lee:Using RDF On the Web - a Vision http://thefigtrees.net/lee/blog/2007/01/using_rdf_on_the_web_a_vision.html (2007)
13. Feigenbaum, Lee: Using RDF on the Web - a Survey http://thefigtrees.net/lee/blog/2007/01/using_rdf_on_the_web_a_survey.html (2007)
14. Willison, Simon: Why JSON isn't just for Javascript http://simonwillison.net/2006/Dec/20/json/ (2006)
15. Cyganiak, Richard: RDF/JSON http://dowhatimean.net/2006/05/rdfjson (2006)
16. Torres, E., Feigenbaum, Lee., Clark, K. G.: Serializing SPARQL Query Results in JSON http://www.w3.org/TR/rdf-sparql-json-res/ (2007)
17. Wilson, G.: URF: A Semantic Framework alternative to RDF, XML, and JSON. http://www.urf.name/ (2008)