Transforming the JSON Output of SPARQL Queries for Linked Data Clients

Pasquale Lisena
EURECOM
Sophia Antipolis, France
pasquale.lisena@eurecom.fr

Raphaël Troncy
EURECOM
Sophia Antipolis, France
raphael.troncy@eurecom.fr

ABSTRACT

SPARQL endpoints are one possible access method to linked data. The results of SPARQL queries serialized in JSON are, however, not suitable to be directly used by web developers in end-user applications who often need to merge the values resulting from variable bindings. In this work, we propose a generic approach implemented in a JavaScript module that takes as input a JSON file describing both the SPARQL query and the shape of the expected output at the same time.

CCS CONCEPTS

• **Information systems → Data access methods**; **Semantic web description languages**; *Query languages*; *Extraction, transformation and loading*;

KEYWORDS

SPARQL, JSON, JSON-LD, transformer, JavaScript

ACM Reference Format:
Pasquale Lisena and Raphaël Troncy. 2018. Transforming the JSON Output of SPARQL Queries for Linked Data Clients. In *WWW '18 Companion: The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3184558.3188739

1 INTRODUCTION

The SPARQL W3C recommendation [5] defines not only a query language for data represented in RDF, but also the protocol for retrieving the data, including a standardised format for the answers in XML, CSV/TSV and in JSON [9]. The latter one, in particular, has been introduced with the purpose of easing the consumption of the data by web (and not only) applications, through its complete compatibility with JavaScript. However, the nature of data models between the Web of Data (RDF, graph-oriented) and the Web of Applications (JSON, document-oriented) is very different. In practice, the output of SPARQL endpoints is a set of all possible bindings (of the form <variable, value>) that satisfies the query, which is not handy for efficient client processing. Current habits, expertise and coding technologies for web applications make the manipulation of nested objects easier for developers than the manipulation of triples.

WWW '18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3188739

As an example, let's retrieve a list of Italian cities, with their name and an illustration from DBpedia[1], limited to the first 100 results.

```
SELECT *
WHERE {
 ?city a dbo:City ;
       dbo:country dbr:Italy ;
       foaf:depiction ?image ;
       rdfs:label ?name .
} LIMIT 100
```

The output – of which Figure 1 contains an extract – is hard to read and manipulate.

The output described in Figure 2, in which the number of results and the different values belonging to each result are clearly visible, is much compact and ready to be consumed.

1.1 Web developer requirements

We describe four different needs that developers face when manipulating SPARQL JSON output:

1. Skip irrelevant metadata. A typical SPARQL output contains a lot of metadata that are usually not useful for web developers. This is the case of the `head` part that contains the list of variables that one might find in the results. In practice, the developer ignores completely this part and he checks for the availability of a certain property directly in the JSON tree.

2. Reducing and parsing. The value of a property is always wrapped in an object with at least the attributes *type* (uri or literal) and *value*, that contain the information. As a consequence, this information is bounded at a deeper level in the JSON structure than the one the developer expects. This makes the manipulation more clumsy. In addition, each literal is expressed as a string value with a datatype. The consequence is that numbers and booleans need to be casted for being used.

3. Merging. Taking as example a visualisation interface, displaying the various names of Bologna in different languages means merging the results of the first two elements of the bindings, being careful to keep all the different names of the city and the unique but repeated value for the image. A merge becomes mandatory especially when the number of properties that have multiple values grows (i.e. multilingual names, multilingual descriptions, a set of images) and the endpoint output contains a result for each combination of values. This task becomes even harder when the required output foresees a deeper structure – i.e. the names and surnames of the most famous citizens for each city can not be merged in two distinct arrays, but do better fit in an array of objects.

[1] http://dbpedia.org/sparql

Figure 1: Extract of a JSON output from a SPARQL query. It includes 3 results, among which 2 (A, B) refers to the same entity (`dbr:Bologna`).

4. Mapping. The web developer may want to map the results to another structure or vocabulary such as *schema.org*.

1.2 Transforming the output

In this paper, we present a JavaScript module called *SPARQL Transformer*, with the goal of transforming the JSON output to a chosen structure. The approach relies on a novel query language – based on JSON – that is able to specify both the query and the result structure at the same time.

```
{
  "@context": "http://schema.org",
  "@graph": [{
    "@type": "City",
    "@id" : "http://dbpedia.org/resource/Bologna",
    "name": [
       {"@value":"Bologna","@language":"it"},
       {"@value":"Bologne","@language":"fr"}
     ],
    "image": "...FilePath/Bologna_postcard.jpg"
  }, {
    "@type": "City",
    "@id" : "http://dbpedia.org/resource/Siena",
    "name": { "@value":"Siena", "@language":"en" },
    "image": ".../PiazzadelCampoSiena.jpg"
  }]
}
```

Figure 2: Desired JSON-LD output of a SPARQL query.

The implementation is strongly inspired by the JSON-LD syntax. This version of JSON defines in its standard some interesting Linked Data possibilities, such as the ability of assigning an URI that identifies an entity (`@id`), defining its class (`@type`), declaring the language of a literal (with the couple `@language`/`@value`). Nevertheless, *SPARQL Transformer* is able to generate JSON with any shape, among which JSON-LD occupies a privileged position.

Among the two available query syntax (plain JSON and JSON-LD), we will use examples using the JSON-LD one, while the repository README fully explained how to use the JSON one. The module, with the full documentation and more examples is available at https://github.com/D2KLab/sparql-transformer.

2 RELATED WORK

Issues about the usage of SPARQL output in real-life applications have been explored in [1], where the proposed solution is specific to the generation of HTML reports.

Various works have provided bridges between the Web of Data and the developers. Among these, we can cite `grlc`, a software for the automatic generation of web API from SPARQL query contained in GitHub repositories [7].

Wikidata SDK [6] addresses the problem of the complexity of the SPARQL JSON output through a precise function[2] that transform the JSON output to a simplified version by reading the variable names. The implementation takes care of the reduction and parsing tasks, but it does not address the problem of merging.

The provision to JavaScript developers of an easier way to deal with RDF is also capturing the effort of the W3C RDFJS Community Group[3], that produced a low level interface specification for the interoperability of RDF data in JavaScript environments [2]. Their approach tries to preserve the triple format of the information and the graph-oriented model of RDF.

[2] https://github.com/maxlath/wikidata-sdk/blob/master/docs/simplify_sparql_results.md
[3] https://www.w3.org/community/rdfjs/

The attempt of converting and mapping RDF sources has led to *SPARQL Template Transformation Language (STTL)* [3]. This language allows to define transformation templates (as strings) in which the results of the SPARQL query will find place. Moreover, it exposes a significant number of functions, especially when combined with LDScript [4]. Despite transformation being a goal of STTL, it does neither focus on the conversion to JSON-LD, nor on the merging binding results as exposed in Section 1.

The W3C SPARQL Specification itself includes a query format called `CONSTRUCT` for extracting from the endpoint a set of triples, which is eventually different from its actual content and can also consist in a mapping [5]. It returns in output a graph, following one of the standard SPARQL outputs, including a JSON-LD version. As a limit, literals are not parsed, and they are always represented as objects. This format is less popular than the `SELECT` one among developers, who are consequently less used to it. Moreover, it does not allow the use of aggregate functions. For this reason, it is no possible to rely on the sole `CONSTRUCT` for shaping JSON with predefined structure. This is demonstrated also by `sparql-to-jsonld`, a command-line library that aims to pass from SPARQL to JSON-LD requiring on 3 different inputs: a `SELECT` query, a `CONSTRUCT` or `DESCRIBE` query and a JSON-LD frame [8].

Finally, *JSON Schema* is a format for defining the structure of a JSON object. Although it is a powerful tool for validation – for example – of forms and APIs, there are not evident benefits for JSON reshaping purposes [10].

3 TEMPLATING THE RESULT WITH THE QUERY

The strategy of this work relies in the use of a single JSON object for defining the query and the expected structure (or *prototype*) of the output. Different from SPARQL `CONSTRUCT`, the query and the final structure are not two distinct parts of the query, but they are expressed together at the same time. This union of intent is strengthen by the adoption of the expecting result format – JSON – as query dialect itself.

Two formats (and two types of output) are supported: plain JSON and JSON-LD. When not differently specified, the former use the same keys of the JSON-LD one, without prepending them with the `@` sign. The syntax of both formats is composed by two main parts: the prototype definition and the root `$`-properties.

For the JSON-LD version, a `@context` property is foreseen for specifying the context.

3.1 The prototype definition

The `@graph` property (or `proto`, for plain JSON syntax) contains the prototype of the result as expected by the user. Values in the prototype can start with:

- an interrogative point "?" (like `?id` or `?city`), suggesting that the value should be replaced by the one of the homonym SPARQL variable;
- a "$" sign, that identifies the parts that requires to be processed by the software;
- any valid value, that will be present in the output as is and which do not depend from the query result.

```
{
  "@context": "http://schema.org/",
  "@graph": [{
    "@type": "City",
    "@id": "?id",
    "name": "$rdfs:label$required$lang:it",
    "image": "$foaf:depiction$required"
  }],
  "$where": [
    "?id a dbo:City",
    "?id dbo:country dbr:Italy"
  ],
  "$limit": 100
}
```

Listing 1: A query in the JSON-LD format

```
{
  "@context": "http://schema.org/",
  "@graph": [{
    "@id": "?city",
    "name": "$rdfs:label",
    "containedInPlace":{
      "@id" : "?region",
      "name": "$rdfs:label$lang:it"
    }
  }],
  "$where" : "?id dbo:region ?region"
}
```

Listing 2: The query can contain nested objects. The two `rdfs:labels` refer respectively to `?city` and `?region`.

When the value should be taken from the query result, it is declared using the following syntax:

```
$<SPARQL PREDICATE>[$modifier[:option]...]
```

The main part is the SPARQL predicate (a property or a path, e.g. `rdfs:label`, `foaf:depiction`, *etc.*). The object of the predicate is automatically assigned, unless the presence of the `$var` modifier.

The subject of the predicate is the variable of the closest `@id` in the structure. If this variable does not exist, it is set to `?id` by default. This allows to go beyond a flat set of primitive properties, and have objects at different levels in the JSON structure: each object, as soon as it owns an `@id`, behaves as a nested prototype.

Listing 2 shows a practical example, by representing the relative region of each city in a `containedIn` property. This property is valorised by an object that contains the id and the label. In this case, the subject of the predicate `rdfs:label` is not `?city` (root `@id`) but `?region` (closer `@id`), and this would happen to any other eventual property of the object. This allows to avoid to write paths of properties longer the what is necessary.[4]

Some modifiers can be present after the predicate, separated by an equal number of `$` sign. These modifiers will be used in the

[4]See the full example at https://github.com/D2KLab/sparql-transformer/blob/master/query.examples.md

PROPERTY	INPUT	DESCRIPTION
`$where`	string, array	Add where clause in the triple format.
`$values`	object	Set VALUES for specified variables as a map.
`$limit`	number	LIMIT the SPARQL results
`$distinct`	boolean (default `true`)	Set the DISTINCT in the select
`$orderby`	string, array	Build an ORDER BY on the variables in the input.
`$groupby`	string, array	Build an ORDER BY on the variables in the input.
`$having`	string, array	Allows to declare the content of HAVING.
`$filter`	string, array	Add the content as a FILTER.
`$prefixes`	object	set the prefixes in the format `"foaf": "http://xmlns.com/foaf/0.1/"`.

Table 1: Supported root $-properties

query step and have a default values chosen in order to make the developer to retrieve the maximum information. For example, all the values are optional, unless specified as `$required`.

A special role is given to the `$var` modifier, because it allows to assign a specific SPARQL variable as object (e.g. `$var:?myVar`), so that it can be addressed in other modifiers. Other possibilities include filtering by language (`$lang:it`) or sample those values (`$sample`).

As stated before, some properties can have as value a SPARQL variable (e.g. `?id`), that will be transferred to the output replaced by its query result value. Also these properties can have modifiers as the previous ones, for example for making them required.

3.2 The root $-properties

Some root properties – identified by a key starting with a $ – give access to different SPARQL features. They will not be transferred to the output, being considered only at query level. The names of those properties maps exactly to the reserved keywords in SPARQL (`$limit`, `$filter`, `$groupby`, `$orderby`, *etc.*), so that their interpretation is immediate for developers who are familiar with the query language. They often support both a single value or an array. For example, this is the case of `$where`, that enables to specify `WHERE` clauses – in the triple format – which will be added to the ones automatically generated by the prototype.

Table 1 shows the currently implemented $-properties.

4 IMPLEMENTATION

SPARQL Transformer is a library that receives as input the query in JSON described in Section 3, translates it in a SPARQL query, performs the request to the endpoint and returns a JSON output with the desired shape. Implemented as a JavaScript module, it can run both in the browser and in Node.JS environments. It is made of three parts that work in sequence, as shown in Figure 3.

The input is read by the **Parser**, that has the goal of extracting from it the SPARQL `SELECT` query and the prototype. The reason for which we rely on `SELECT` instead of `CONSTRUCT` is detailed in Section 2.

The root $-properties are collected and transformed to match the SPARQL syntax. For the prototype, it requires that for each property:

- It reads the assigned SPARQL variable or generates one automatically;

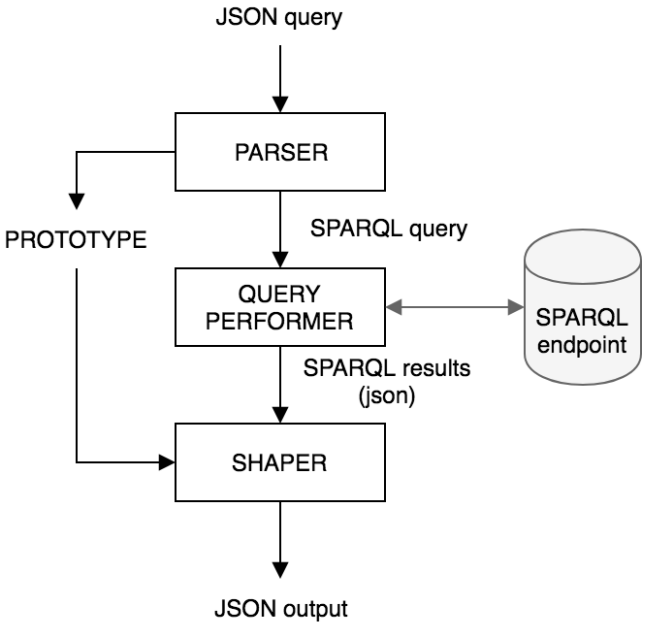

Figure 3: The application schema of *SPARQL Transformer*.

- It adds it to the list or variable in the `SELECT`;
- It reads and interprets the modifiers;
- If a predicate is declared, it assigns to it the object and the subject, taken from the closest `@id`;
- Considering the previous points, it generates `WHERE` clauses, `FILTER`s, *etc.*.

The two outputs of this part are a SPARQL query and a cleaned prototype, in which all the values are SPARQL variables (e.g. `"@id": "?id"`, `"name":"?v0"`), that will be used as placeholders later.

The SPARQL query is passed to the **Query Performer** that is in charge of making an HTTP request to the SPARQL endpoint (that can be specified in input) in order to collect the results[5]. In case of particular needs, this part can be entirely replaced by the end developer with a custom function, that receives as input the generated SPARQL query and return the results through a `Promise`.

[5]It internally relies on Virtuoso SPARQL Client, https://github.com/crs4/virtuoso-sparql-client.

The last part is the **Shaper**. First, it creates as many instances of the prototype as the result items are, replacing the placeholders with the real data. If some results does not contain a certain value – which happens when the variable is `OPTIONAL` –, it is removed from the instance. Then, it merges the object with the same `@id`, transforming the property values in array when needed and appending all the distinct values. Nested objects are also involved in this step: they are merged in the same way if they share the `@id` or alternatively the value of all properties, otherwise they are considered distinct and aggregated in an array.

If the query object used the JSON-LD syntax, the module packs the results in a JSON-LD structure, by wrapping it in a `@graph` and adding the desired `@context`.

Finally the result is returned in output.

5 CONCLUSION AND FUTURE WORK

The need of easing the manipulation of SPARQL results in web environments led us to define a new strategy of querying the endpoints based on the use of JSON. This strategy found an implementation in a JavaScript module called *SPARQL Transformer*.

This work aims to give a solution to some of the problems that developers are used to face with SPARQL json results. Among them:

- Writing a query and the expected output at the same time;
- Parsing the values according to the datatype;
- Give to the results a different structure than the flat set of <variable, value> items retrieved by default;
- Merging the results that describe the same entity;
- Map the results to a different vocabulary;
- Realise a JSON-LD middleware for RDF data.

Considering that it makes use only of JS and JSON, its use cases easily include Node.JS environments (e.g., for realising a web API) or directly the browser (for directly retrieving and consuming data coming from the endpoint).

SPARQL Transformer is designed to fulfil the most common needs of developers, with the clear limit of the difficulty in getting the same expressiveness of a mature query language as SPARQL. Its implementation covers different features of SPARQL specification, while further ones are foreseen as future work.

Further studies will investigate if it would be eventually possible to cover other kind of SPARQL operation, like `ASK`, `INSERT` and `DELETE`.

Finally, we plan to better evaluate the library. This goal will be reached in two different ways. On one side, we will try to apply *SPARQL Transformer* to existing collection of queries, for evaluating its coverage. On the other side, we will realise a web interface that enables final developers to test the library, making also use of examples coming from the documentation.

ACKNOWLEDGMENTS

This work has been partially supported by the French National Research Agency (ANR) within the DOREMUS Project, under grant number ANR-14-CE24-0020.

REFERENCES

[1] Sunitha Abburu and G Suresh Babu. 2013. Format SPARQL Query Results into HTML Report. *International Journal of Advanced Computer Science and Applications (IJACSA)* 4, 6 (2013), 144–148.

[2] Thomas Bergwinkl, Michael Luggen, elf Pavlik, Blake Regalia, Piero Savastano, and Ruben Verborgh. 2017. *Interface Specification: RDF Representation, Draft Report.* Technical Report. W3C.

[3] Olivier Corby, Catherine Faron-Zucker, and Fabien Gandon. 2015. A generic RDF transformation software and its application to an online translation service for common languages of linked data. In *14th International Semantic Web Conference (ISWC)*. Bethlehem, Pennsylvania, USA, 150–165.

[4] Olivier Corby, Catherine Faron-Zucker, and Fabien Gandon. 2017. LDScript: a Linked Data Script Language. In *16th International Semantic Web Conference (ISWC)*. Vienna, Austria, 208–224.

[5] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language – W3C Recommendation.* Technical Report. W3C.

[6] Maxime Lathuilière. 2015. `Wikidata SDK`. GitHub repository, https://github.com/maxlath/wikidata-sdk. (2015).

[7] Albert Meroño-Peñuela and Rinke Hoekstra. 2016. grlc Makes GitHub Taste Like Linked Data APIs. In *The Semantic Web – ESWC 2016 Satellite Events*. 342–353.

[8] Jindřich Mynarz. 2016. `sparql-to-jsonld`. GitHub repository, https://github.com/jindrichmynarz/sparql-to-jsonld. (2016).

[9] Andy Seaborne. 2013. *SPARQL 1.1 Query Results JSON Format – W3C Recommendation.* Technical Report. W3C.

[10] Austin Wright and Henry Andrews. 2017. *JSON Schema: A Media Type for Describing JSON Documents.* Technical Report. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-handrews-json-schema/