Easy Web API Development with SPARQL Transformer

Pasquale Lisena[1](✉), Albert Meroño-Peñuela[2], Tobias Kuhn[2], and Raphaël Troncy[1]

[1] EURECOM, Sophia Antipolis, France
`{pasquale.lisena,raphael.troncy}@eurecom.fr`
[2] Vrije Universiteit, Amsterdam, The Netherlands
`{t.kuhn,albert.merono}@vu.nl`

Abstract. In a document-based world as the one of Web APIs, the triple-based output of SPARQL endpoints can be a barrier for developers who want to integrate Linked Data in their applications. A different JSON output can be obtained with SPARQL Transformer, which relies on a single JSON object for defining which data should be extracted from the endpoint and which shape should they assume. We propose a new approach that amounts to merge SPARQL bindings on the base of identifiers and the integration in the `grlc` API framework to create new bridges between the Web of Data and the Web of applications.

Keywords: SPARQL · JSON · JSON-LD · API

1 Introduction

The Semantic Web is a valuable resource of data and technologies, which is having a crucial role in realising the initial idea of Web. RDF can potentially represent any kind of knowledge, enabling reasoning, interlinking between datasets, and graph-based artificial intelligence. Nevertheless, a structural gap exists that is limiting a broader consumption of RDF data by the community of Web developers. Recent initiatives such as EasierRDF[1] are strongly pushing the proposal of new solutions for making Semantic data on the Web *developer friendly* [3,10].

We focus here on the output format of SPARQL endpoints, and in particular, query results in the JSON format [24]. This standard is part of the SPARQL W3C recommendation [12], introduced with the purpose of easing the consumption of the data by Web (and non-Web) applications. The format consists of a set of all possible bindings (of the form `<variable, value>`) that satisfies the query. This is not handy for efficient processing by clients, which would prefer nested objects (document-based data structures) rather than this representation of triples (graph-oriented data structures). An example of this is shown in Fig. 1.

[1] https://github.com/w3c/EasierRDF.

```
SELECT DISTINCT *
WHERE {
    ?id a dbo:City ;
          dbo:country dbr:Italy ;
          rdfs:label ?label .

    OPTIONAL { ?id foaf:depiction ?image }.

    ?id dbo:region ?region .
    ?region rdfs:label ?region_name .
    FILTER(lang(?region_name) = 'it')

} LIMIT 100
```

(a)

```
[{
    "id": "http://dbpedia.org/resource/Siena",
    "name": [{
        "language": "fr",
        "value": "Sienne"
      },
      {
        "language": "it",
        "value": "Siena"
      },
    ],
    "image": "./PiazzadelCampoSiena.jpg",
    "region": {
      "id": "http://dbpedia.org/resource/Tuscany",
      "name": {
        "language": "it",
        "value": "Toscana"
      }
    }
  },
  {
    "id": "http://dbpedia.org/resource/Milan",
    "name": {
      "language": "en",
      "value": "Milan"
    },
    "image": "./Flag_of_Milan.svg",
    "region": {
      "id": "http://dbpedia.org/resource/Lombardy",
      "name": {
        "language": "it",
        "value": "Lombardia"
      }
    }
  }]
```

(b)

```
{
  "head": {
    "link": [],
    "vars": [ "id", "label", "image", "region", "region_name" ]
  },
  "results": {
    "distinct": false,
    "ordered": true,
    "bindings": [{
        "id": {
          "type": "uri",
          "value": "http://dbpedia.org/resource/Siena"
        },
        "label": {
          "type": "literal",
          "xml:lang": "it",
          "value": "Siena"                                          A
        },
        "image": {
          "type": "uri",
          "value": "./PiazzadelCampoSiena.jpg"
        },
        "region": { ... },
        "region_name": { ... }
      },
      {
        "id": {
          "type": "uri",
          "value": "http://dbpedia.org/resource/Siena"
        },
        "label": {
          "type": "literal",
          "xml:lang": "fr",
          "value": "Sienne"                                         B
        },
        "image": {
          "type": "uri",
          "value": "./PiazzadelCampoSiena.jpg"
        },
        "region": { ... },
        "region_name": { ... }
      },
      {
        "id": {
          "type": "uri",
          "value": "http://dbpedia.org/resource/Milan"
        },
        "label": {
          "type": "literal",
          "xml:lang": "en",                                         C
          "value": "Milan"
        },
        "image": {
          "type": "uri",
          "value": "./Flag_of_Milan.svg"
        },
        "region": { ... },
        "region_name": { ... }
      }]
  }
}
```

(c)

Fig. 1. A SPARQL query (a) extracting a list of Italian cities with picture, label and belonging region, of which the URI and the Italian name are also requested. In the standard output of the endpoint (c), the city of Siena is represented by both object A and B, while the transformed output (b) offers a more compact structure.

Given this situation, we identify four tasks that developers have to fulfil:

1. **Skip irrelevant metadata.** A typical SPARQL output contains a lot of metadata that are often not useful for Web developers. This is the case of the `head` field, which contains the list of variables that one might find in the results. In practice, developers may ignore completely this part and check 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*, containing

the information. As a consequence, this information is bounded at a deeper level in the JSON structure than the one the developer expects. In addition, each literal is expressed as a string value with a datatype, so that numbers and booleans need to be casted.
3. **Merging.** As the query results represent all the valid solutions of the query, it is possible that two bindings differ only by a single field. When the number of properties that have multiple values grows (i.e. multilingual names, multilingual descriptions, a set of images), the endpoint returns even more results, one for each combination of values. The consumption of such data requires often to identify all the bindings which represent a given entity, merging the objects on the URI. The presence of more variables on which the merging can be performed can further complicate the merging process.
4. **Mapping.** The Web developer may want to map the results to another structure – i.e. for using them as input to a library – or vocabulary such as *schema.org*.

In addition to this, the support for curating and reusing SPARQL queries is sub-optimal, these queries typically end up being hard-written in the application code. A specifically unsettling case of these Linked Data (LD) APIs, which refer to those APIs that just wrap underlying SPARQL functionality. To solve this problem, various works have provided bridges between the Web of Data and the developers. `grlc` is a software for the automatic generation of Web APIs from SPARQL queries contained in GitHub repositories [16]. **SPARQL Transformer**[2] is a library that gives a chosen structure to the SPARQL output. The library is able to perform all the above mentioned tasks, helping Web developers in the manipulation of data from the Web.

This paper largely extends [15] with a more organic description of the module, the integration of SPARQL Transformer in `grlc` and Tapas, a playground application for testing the query outcome and an evaluation on performance and usability. Moreover, the library has been ported to Python, and a set of new features have been included, most importantly the support of OFFSET (allowing pagination, e.g. in `grlc`) and language filtering for the management of multi-language APIs. The remainder of this paper is structured as follows: we propose a thorough review of other works which aim to ease the consumption of RDF data and their limitations in Sect. 2. We introduce the new JSON format for queries in Sect. 3, which feeds the SPARQL Transformer library detailed in Sect. 4. The work is finally evaluated in Sect. 5, while some conclusions and future work are presented in Sect. 6.

2 Related Work

The need for overcoming the issues about the usage of SPARQL output in real-life applications has inspired different works. One of the first proposed solutions

[2] SPARQL Transformer is available at https://github.com/D2KLab/sparql-transformer as a JavaScript library, while a Python implementation is available at https://github.com/D2KLab/py-sparql-transformer.

consists in a strategy for representing the SPARQL output in a tabular structure, to address the creation of HTML reports [1].

Wikidata SDK [14] takes care of the reduction and parsing tasks through a precise function[3] that transforms the JSON output to a simplified version by reading the variable names. However this implementation does not address the problem of merging.

The conversion of RDF data can rely on the *SPARQL Template Transformation Language (STTL)* [4]. Those transformation templates (as strings) are exploited for shaping the results of the SPARQL query. Moreover, STTL exposes a significant number of functions, especially when combined with LDScript [5]. Among the limits of this approach is the absence of any support for converting the results to JSON-LD. No merging strategy is also studied in this approach.

The W3C RDFJS Community Group[4] is heavily contributing to the effort of offering a tool to JavaScript developers for using RDF data. The major outcome of the initiative is a low-level interface specification for the interoperability of RDF data in JavaScript environments [2]. RDFJS brings the graph-oriented model of RDF into the browser, allowing developers to directly manipulate triples.

The `CONSTRUCT` query format – included in the W3C SPARQL Specification [12] – can be seen as a way for mapping the SPARQL results into a chosen structure, following one of the standard SPARQL output formats, including JSON-LD. An attempt has been realised by the command-line library `sparql-to-jsonld` [17]. The need for three different inputs – a `SELECT` query, a `CONSTRUCT` or `DESCRIBE` query, and a JSON-LD frame – indirectly proves that a sole `CONSTRUCT` for shaping JSON with non predefined structure is not sufficient. Indeed, the `CONSTRUCT` keyword can only generate triplesets, from which the generation of JSON tree-like documents is ambiguous. This is inconvenient for developers, and leads to the problem of how to change the structure of the query result. JSON-LD Framing[5] overcomes this problem, but, in our opinion, the combination is not easier for developers who would have to write and keep in sync the two parts (query and result shape). The complexity of writing a `CONSTRUCT` query – i.e. with respect to a `SELECT` one – can be an additional deterrent for its usage. Furthermore, literals are not parsed and they are always represented as objects, and aggregate functions are not supported.

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 no evident benefits for JSON reshaping purposes [29].

The development of *SOLID* framework for decentralised LD applications [28] gives popularity to its module *LDflex*[6] for retrieving and manipulating Linked Data. LDflex allows the user to browse nodes in the graph by accessing to JS

[3] https://github.com/maxlath/wikidata-sdk/blob/master/docs/simplify_sparql_results.md.
[4] https://www.w3.org/community/rdfjs/.
[5] https://www.w3.org/TR/json-ld11-framing/.
[6] https://github.com/RubenVerborgh/LDflex.

properties. Thus, the paradigm of this module is different, consisting in navigating the graph following the links, rather than finding solutions to structured queries.

There is abundant work in SPARQL query repositories, which are typically used to study the efficiency and reusability of querying. For example, in [21] authors use SPARQL query logs to study differences between human and machine executed queries; in [13], these logs are used to understand the semantic relations between queried entities. Saleem et al. [23] propose to "create a Linked Dataset describing the SPARQL queries issued to various public SPARQL endpoints".

There is also a large body of Semantic Web literature on Linked Data and Web Services [9,20]. In [25] and the smartAPI [30], the authors propose to expose REST APIs as Linked Data, and enumerate the advantages of using Linked Data technology on top of Web services. In the opposite direction, the Linked Data API specification[7] and the W3C Linked Data Platform 1.0 specification, describe "the use of HTTP for accessing, updating, creating and deleting resources from servers that expose their resources as Linked Data"[8]. Our work follows this direction, and is more related to providing APIs that facilitate Linked Data access and query results consumption. The OpenPHACTS Discovery Platform for pharmacological data [11], LDtogo [19] and the BASIL server [6] use SPARQL as an underlying mechanism to implement APIs and provide Linked Data query results. Influenced by these works, `grlc` [16], a technology we extend in this paper, decouples query storage from API implementations by leveraging queries uniquely and globally identified by stable and de-referenceable URIs, automating the query construction process.

Recent works realised an interoperability between the GraphQL language[9] and RDF, performing in this way a conversion in JSON of the data in an endpoint [27]. The same syntax of GraphQL allows to produce a JSON object with different levels of nested nodes. Some of these solutions rely on automatic mappings of variables to property names (Stardog[10]), while others rely on a schema (HyperGraphQL[11]) or a context (GraphQL-LD [26]) which the developer is in charge to provide. None of those approaches implements any strategy for detecting and merging bindings referring to the same entity.

3 The JSON Query Syntax

As seen in the experiences reported in Sect. 2, the natural choice of format for defining and developing a transformation template involves JSON or its JSON-LD serialisation, which is usually added to the SPARQL query. The names of

[7] https://github.com/UKGovLD/linked-data-api.
[8] https://www.w3.org/TR/2015/REC-ldp-20150226/.
[9] https://graphql.github.io/.
[10] https://www.stardog.com/.
[11] https://www.hypergraphql.org.

```
1  {
2    "proto": {
3      "id" : "?id",
4      "name": "$rdfs:label$required",
5      "image": "$foaf:depiction",
6      "region": {
7        "id" : "$dbo:region$required",
8        "name": "$rdfs:label$lang:it"
9      }
10   },
11   "$where": [
12     "?id a dbo:City",
13     "?id dbo:country dbr:Italy"
14   ],
15   "$limit": 100
16 }
```

Listing 1.1. The JSON version of the SPARQL query in Fig. 1

```
SELECT DISTINCT ?id ?v1 ?v2 ?v3r ?v31 WHERE {
    ?id a dbo:City.                               # 12
    ?id dbo:country dbr:Italy.                    # 13
    ?id rdfs:label ?v1.                           # 4
    OPTIONAL { ?id foaf:depiction ?v2  }.         # 5
    ?id dbo:region ?v3r .                         # 7
    OPTIONAL { ?v3r rdfs:label ?v31 .
        FILTER(lang(?v31) = "it") }               # 8
}
LIMIT 100                                         # 15
```

Listing 1.2. The intermediate SPARQL query. The comments contain line numbers which identify which part of the JSON query in Listing 1.1 generates the statement.

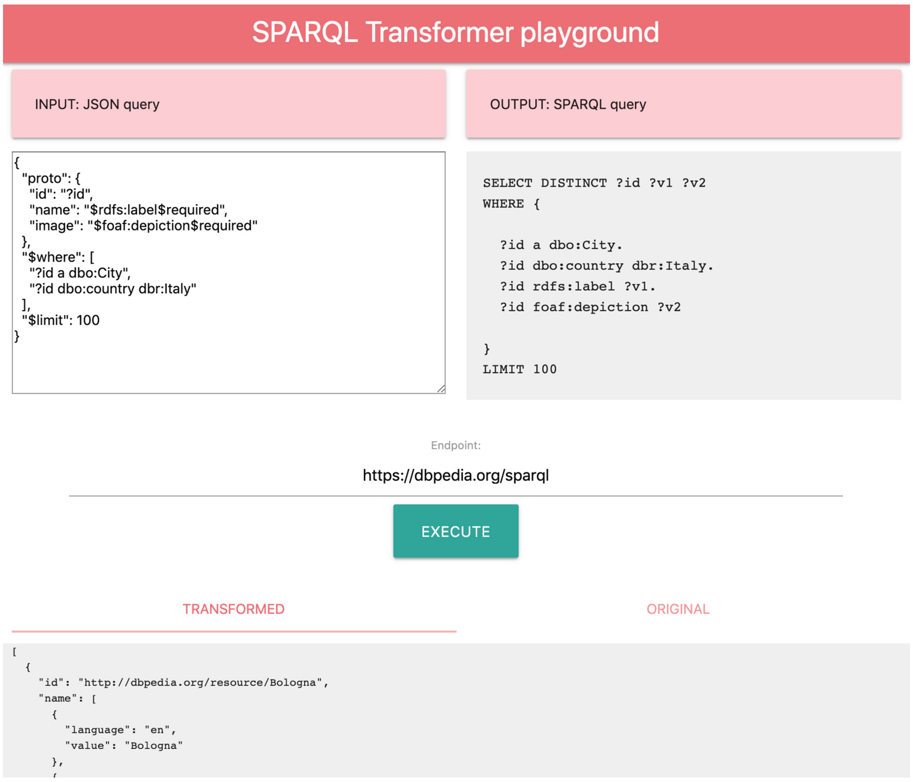
the variables used should match between the template and the query, making the developing process error-prone.

Our proposal is to use a single JSON object, called *JSON query*, with the double role of declaring how to find the information (query) and which structure is expected in its output (template). These properties put the JSON query at a certain distance also from SPARQL `CONSTRUCT`, in which the query and the final structure are two distinct parts of the query.

The syntax of JSON queries consists of two main parts (Listing 1.1):

– the prototype definition, which describes the output structure, expressed as an object and introduced by the `proto` property;
– a set of rules to be included in the SPARQL query, defined through a set of properties starting with the `$` sign, e.g. `$where` and `$limit`.

JSON queries can be expressed in two different formats, producing coherently the output: plain JSON and JSON-LD. The latter foresees a slightly different syntax in order to return an output compliant with the JSON-LD specification. This version of the query allows to specify a JSON-LD context, and can be used for mapping the results into a chosen vocabulary. We refer to the documentation[12] for more details.



```
{
  "proto": {
    "id": "?id",
    "name": "$rdfs:label$required",
    "image": "$foaf:depiction$required"
  },
  "$where": [
    "?id a dbo:City",
    "?id dbo:country dbr:Italy"
  ],
  "$limit": 100
}
```

```
SELECT DISTINCT ?id ?v1 ?v2
WHERE {

  ?id a dbo:City.
  ?id dbo:country dbr:Italy.
  ?id rdfs:label ?v1.
  ?id foaf:depiction ?v2

}
LIMIT 100
```

```
[
  {
    "id": "http://dbpedia.org/resource/Bologna",
    "name": [
      {
        "language": "en",
        "value": "Bologna"
      },
```

Fig. 2. User interface of SPARQL Transformer playground

A Web application called **SPARQL Transformer playground**[13] has been developed in order to quickly test JSON queries. The application is live converting the JSON into a corresponding SPARQL query, so that the user can appreciate every single change. In addition, it is possible to execute the query against a given endpoint, and the user interface offers the possibility of comparing the transformed output with the original one (Fig. 2).

[12] https://github.com/D2KLab/sparql-transformer.
[13] https://d2klab.github.io/sparql-transformer/.

3.1 The Prototype Definition

By prototype, we mean the common structure each object in output should respect. It is designed as an ordinary JSON object, in which the leaf nodes will be replaced by incoming data according to specific rules. In particular:

1. **variable nodes**, which start with a question mark "`?`" (like `?id` or `?city`), are replaced by the value of the homonym SPARQL variable;
2. **predicate nodes**, which starts with a "`$`" sign, are replaced by the object of a specific RDF triple;
3. **literal nodes**, which cover all the other contents, are not replaced and will be present as is in the output, regardless of the query results.

In the transforming process, SPARQL triples will be automatically generated from the prototype. Referring to case 2, the following syntax is used:

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

The first parameter is the SPARQL predicate, which can be a property or a property path, e.g. `rdfs:label`, `foaf:depiction`, etc. This kind of node will be replaced by the object of an RDF triple having as predicate the one given inline. As subject, the variable of the sibling *merging anchor* is selected if it exists; otherwise, the closer merging anchor among the parent nodes. The merging anchors are all the fields in the JSON introduced with the `id` property. If this variable does not exist, it is set to `?id` by default. In other words, each level in the JSON tree may declare a specific subject through the merging anchor, which will be the subject of all the predicates in the scope. Listing 1.1 includes two merging anchors at line 3 and 7: the former acts as subject of the name, image, and region; while the region name refers to the latter.

The role of the *merging anchor* is crucial for the following steps. In fact, two result objects having the same id will be considered as the same item and their properties will be merged. This will happen at each level of the JSON tree. This controlled way of aggregating SPARQL results ensures a more compact while not less informative output, ready to be used by Web developers.

Both variable and predicate nodes can accept some modifiers appended at the end of the string, separated by the `$` sign. These elements are taken in account when writing the SPARQL query. For example, `$required` avoids the predicate to be considered optional (the default behaviour), while `$var` assigns a specific SPARQL variable as object (e.g. `$var:?myVar`), so that it can be addressed in other modifiers. Other possibilities include filtering by language (`$lang:it` or `$bestlang:en;q=1, it;q=0.7 *;q=0.1`) or sample those values (`$sample`).

3.2 The Root $-properties

A set of `$`-properties give access to the SPARQL features indicated by their name (`$limit`, `$groupby`, etc). These properties are directly assigned to the root of the JSON query object, and will not appear in the final output. Among them, some additional `WHERE` clauses – in the triple format – can be declared in the `$where` field. The `$lang` modifiers set the language chosen for all the `$bestlang` in the prototype. An exhaustive list of implemented `$`-properties is reported in Table 1.

Table 1. Supported root `$`-properties

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	Set the DISTINCT in the select (default `true`)
`$offset`	number	OFFSET applied to the SPARQL results
`$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 `"prefix": "uri"`
`$lang`	string	Default language in the Accept-Language standard [8]

4 Implementation

The implementation of SPARQL Transformer relies on three main blocks, each one having a specific function (Fig. 3).

The **Parser** reads the input JSON query and parses its content. The prototype is extracted and a SPARQL variable – which here acts as a placeholder – is assigned to all the predicate nodes. Contextually, the SPARQL `SELECT` query (Listing 1.2) is generated: the predicate nodes are translated into `WHERE` clauses according to the rules defined in Sect. 3.1 and taking into account the modifiers. The root `$`-properties are parsed and inserted in the query, which is then passed to the **Query Performer**. This module is in charge of performing the request to the SPARQL endpoint and returning the results in the SPARQL JSON output format. The query performer can be replaced by the user with a custom one, for fulfilling different requirements for accessing the endpoint (e.g. authentication) or for integration into more complex environments (as done during the integration with `grlc`).

Finally, the **Shaper** accesses the results, discarding the side information included in the `head` field and directly accessing the bindings. The latter ones are applied to the prototype in sequence, matching the SPARQL variables to the placeholders separately for each binding. In this phase, the data-type of the binding is checked, eventually parsing the value to Boolean, integer or float.

When a result binding does not contain a certain value – which happens when the variable is `OPTIONAL` –, the property is removed from the instance. Then, the instances which have a common value for the merging anchor are identified and their properties are compared, in order to keep all the distinct values without repetition. Recursively, the same merging strategy is applied to the nested objects. Finally, they are serialised in JSON and returned as output.

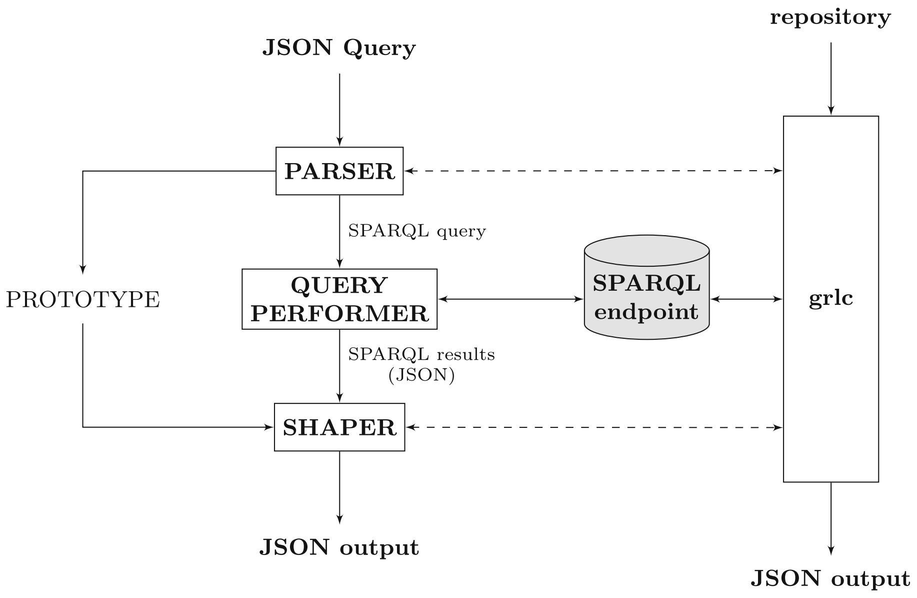
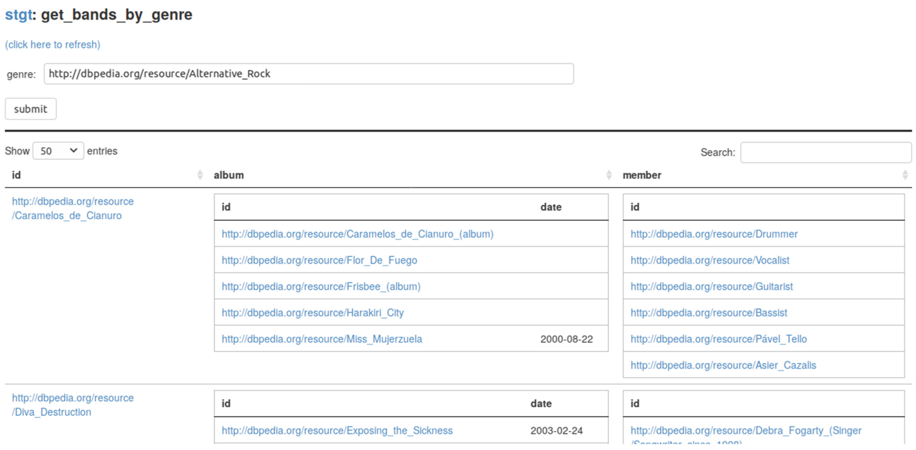


Fig. 3. The application schema of SPARQL Transformer

The SPARQL Transformer library is available in two different implementations in JavaScript and Python, published respectively on the NPM Package Manager[14] and the Python Package Index[15] (PyPI). The JavaScript version has been recently converted in an ECMAScript Module [7] and it is designed to both work in Node.js and in the browser. The Python version return a `dict` object, which can be directly manipulated by a script or serialised in JSON.

Since version 1.3, SPARQL Transformer is included in the `grlc`[16] framework, which is now able to generate Web APIs from the JSON queries contained in a given GitHub repository. The integration involved the Parser and the Shaper: the former is executed before each access to the SPARQL query, keeping in memory the prototype for being shaped once SPARQL results are back. The JSON query file can include the configuration options for `grlc` in an homonym

[14] https://www.npmjs.com/package/sparql-transformer.
[15] https://pypi.org/project/SPARQLTransformer/.
[16] http://grlc.io/.



stgt: get_bands_by_genre

(click here to refresh)

genre: http://dbpedia.org/resource/Alternative_Rock

submit

Show 50 entries Search:

id	album	member
http://dbpedia.org/resource/Caramelos_de_Cianuro	(see nested table 1a)	(see nested table 1b)
http://dbpedia.org/resource/Diva_Destruction	(see nested table 2a)	(see nested table 2b)

id	date
http://dbpedia.org/resource/Caramelos_de_Cianuro_(album)	
http://dbpedia.org/resource/Flor_De_Fuego	
http://dbpedia.org/resource/Frisbee_(album)	
http://dbpedia.org/resource/Harakiri_City	
http://dbpedia.org/resource/Miss_Mujerzuela	2000-08-22

id
http://dbpedia.org/resource/Drummer
http://dbpedia.org/resource/Vocalist
http://dbpedia.org/resource/Guitarist
http://dbpedia.org/resource/Bassist
http://dbpedia.org/resource/Pável_Tello
http://dbpedia.org/resource/Asier_Cazalis

id	date
http://dbpedia.org/resource/Exposing_the_Sickness	2003-02-24

id
http://dbpedia.org/resource/Debra_Fogarty_(Singer

Fig. 4. Screenshot of the Tapas interface

field. For maximising the compatibility, the options can be specified as a YAML string or in JSON. The support to JSON queries includes all the features of `grlc`, such as the pagination and the selection of query parameters. In addition, a `lang` query parameter can change the value of the `$lang` property of the query, allowing the development of multi-language APIs. Further development involved the upgrade of `grlc` to the latest Python version.

Moreover, SPARQL Transformer queries are now also supported by Tapas[17]. Tapas is a small interface module implemented in HTML and JavaScript that reads the specification of an instance of a `grlc` API and turns it into a nice and simple HTML interface. The elements of the API specification are in a straightforward manner transformed into HTML form elements, which the user can fill in to access the service by pressing the *submit* button. Tapas asynchronously calls the API via `grlc` and shows the results at the bottom part of the same page using the YASR component of the YASGUI interface [22] to display the SPARQL query results in a user-friendly manner.We extended Tapas to also support SPARQL Transformer queries and display the results in an equally user-friendly manner. Unlike the flat tables produced by YASR for the common kind of SPARQL results, the nested results of a SPARQL Transformer query are shown as nested tables in Tapas. An example of this can be seen in Fig. 4, showing a screenshot of the query interface and its results for an exemplary SPARQL Transformer query about music bands, with the nested tables derived from the nested structure of the SPARQL Transformer results. Tapas together with `grlc` thereby allow us to automatically generate an intuitive interface for technically-minded end users just from the query file in a completely general and generic manner.

[17] https://github.com/peta-pico/tapas.

5 Evaluation

As evidence of *current* use, we have deployed this tool in two communities driven by H2020 projects which have adopted both SPARQL Transformer and `grlc`. MeMAD[18] uses it to generate automatically an API on top of a knowledge graph describing TV and radio programs which are also automatically annotated. The resulting semantic metadata is hence integrated in the professional Media Asset Management system Flow developed by Limecraft. SILKNOW[19] uses it to generate an API on top of a knowledge graph describing silk-related objects from 10 museums. The generated API is used to empower an exploratory search engine and a virtual assistant.

To provide evidence of *prospective* use of our approach, we carried out two kinds of evaluations:

- an experiment for measuring the compactness of the results and the execution time of SPARQL Transformer;
- a user survey on the preference of users on using a system that presents Linked Data query results through SPARQL Transformer, versus another that does so through traditional SPARQL results rendering.

5.1 Quantitative Evaluation

We test the Python implementation of SPARQL Transformer on a set of five queries detailed in the DBpedia wiki[20] in order to ensure a certain generality. The set involves different SPARQL features (filters, ORDER BY, language filtering, optional triples). Those SELECT queries have been manually converted into JSON queries—with 1 or 2 levels of objects in the JSON tree—, making sure that the transformed query was equal to the original one (variable names apart).

Each query has been resolved against a local instance of the English DBpedia[21], with a traditional SPARQL client for the SPARQL queries and with SPARQL Transformer for the JSON queries. Each execution has been repeated 100 times, with a waiting time of 5 s between consecutive executions, in order to obtain an average result as much as possible not correlated to any workload of the machine.

The results in Table 2 shows that the average execution time of SPARQL Transformer is slightly higher with respect to normal SPARQL queries, never surpassing 0.1 s (limit of the instantaneous feeling according to [18]). The difference in percentage, computed as $100 * (t_{sparql} - t_{json})/avg(t_{sparql}, t_{json})$, do not reveal any regularity in the time increment, even if some patterns suggest that it depends on the number of results and variables for each result. The same dimensions seem to impact also the gap in number of results, smaller in the JSON

[18] https://memad.eu/.

[19] http://silknow.eu/.

[20] https://wiki.dbpedia.org/onlineaccess, Sect. 1.5.

[21] The setup of the endpoint on a local machine relied on *Dockerized-DBpedia*, available at https://github.com/dbpedia/Dockerized-DBpedia.

query responses because of the merging strategy. It is interesting to point out that such difference exists between all valid combinations of values for requested variables and the number of real-world object described. This is evident in the first query, about people born in Berlin, in which the combinations of names in different languages and birth or death date in different formats almost double the number of results. As a consequence, the Prince Adalbert of Prussia[22] appears in 8 distinct (and even non-consecutive) bindings because of its four names and two versions of its death date, correctly merged in the more compact transformed version. The experiment is further detailed in the GitHub repository[23].

Table 2. Differences in number of results and execution time between SPARQL and JSON queries. For each query, is also reported the number of requested variables.

Query name	N. var	N. results			Time (ms)			
		json	sparql	diff %	json	sparql	diff	diff %
1. Born in Berlin	4	573	1132	49%	168	101	67	50%
2. German musicians	4	257	290	11%	61	49	12	22%
3. Musicians born in Berlin	4	109	172	37%	59	51	8	14%
4. Soccer players	5	70	78	10%	210	203	7	3.7%
5. Games	2	981	1020	4%	121	70	51	54%

5.2 User Survey

In order to evaluate the usefulness of the query results as presented by SPARQL Transformer to potential (technically-minded) end-users and developers and to compare them to a more traditional, table-centric provision of SPARQL query results, we conducted a user survey. We hypothesized that the level of nesting would play an important role, as classical SPARQL results are flat tables whereas the JSON structure of SPARQL Transformer allows for nesting.

We therefore constructed a pair of queries in SPARQL Transformer syntax and its corresponding plain SPARQL version for each of three levels of nesting: no nesting (Level 0), one nested structure (Level 1), and two nested structures (Level 2). These queries are all about bands and their albums and members, and they can be run through the DBpedia SPARQL endpoint. An example of two nested structures as found in Level 2 can be seen in Fig. 4 (the two nested structures being *album* and *member*). We then ran each of these six queries and stored the resulting JSON files (i.e. the files generated by SPARQL Transformer and the standard JSON files with the original SPARQL results, respectively). Moreover, we also ran these on Tapas to compare the user interface aspects

[22] http://dbpedia.org/resource/Prince_Adalbert_of_Prussia_(1811-1873).

[23] A notebook is available at https://github.com/D2KLab/py-sparql-transformer/blob/master/evaluation/test.ipynb.

Table 3. The results of the user survey

Type	Level	Preference for our system					Avg.	p-value	
		-2	-1	0	1	2			
JSON results	0 (no nesting)	6	6	4	13	26	0.85	0.0001980	*
	1 (one nesting)	5	5	3	21	21	0.87	0.000009063	*
	2 (two nestings)	3	9	5	17	21	0.80	0.0003059	*
Tapas interface	0 (no nesting)	4	8	3	19	21	0.82	0.0001275	*
	1 (one nesting)	3	10	2	20	20	0.80	0.0002685	*
	2 (two nestings)	4	7	3	16	25	0.93	0.00003589	*

that come with the different representations and nesting styles, and we made screenshots of the result tables. All these files, including queries, their results, and the Tapas screenshots, can be found online[24].

Based on these query results and screenshots, we then created a questionnaire, where we asked the participants for each of the six cases (JSON files and screenshots for each of the three nesting levels) whether they preferred SPARQL Transformer (referred to as "System A") or the classical SPARQL output (referred to as "System B"), displayed using the YASR component of YASGUI. The possible answers consisted of the five options *Strongly prefer B* (value -2), *Slightly prefer B* (-1), *Indifferent* (0), *Slightly prefer A* (1), and *Strongly prefer A* (2). We also asked the participants whether they consider themselves primarily researchers, developers, or none of these two categories, and we asked about their level of expertise with SPARQL and JSON. The questionnaire can be found online[25].

We then asked people to anonymously participate in this user survey via Linked Data related mailing lists (W3C SemWeb list), and internal group lists of Semantic Web groups at VU Amsterdam and EURECOM, in addition to the SIKS list addressing Dutch universities. The form was accessible for 5 days. In this way, we got responses from 55 participants (40 researchers, 9 developers, 6 others). Their level of expertise on SPARQL and JSON was mixed, with average values of 2.44 and 2.87, respectively, on a scale from 0 to 4. Eight participants had no knowledge of SPARQL at all, while only one participant had no knowledge of JSON.

Table 3 shows the results of the survey (the full table can also be found online[26]). We see that we got the full range of replies for all questions, but also that a clear majority prefers our system slightly (1) or even strongly (2). The average values for both types (JSON and Tapas) and all three nesting levels are between 0.80 and 0.93, i.e. close to the value that stands for a slight preference

[24] https://github.com/tkuhn/stgt/.
[25] https://github.com/tkuhn/stgt/blob/master/eval/questionnaire-form.md.
[26] https://github.com/tkuhn/stgt/raw/master/eval-results/questionnaire-results.ods.

of our system (1) and clearly above the value that stands for an indifference between the two (0).

To test whether the preference towards our system is statistically significant, we used a sign test in the form of a binomial test on the answers that were positive (preference of our system) or negative (preference of the existing system), excluding the zero cases (indifference). This test, therefore, does not take the distinction between slight and strong preference into account, but only which system was preferred. The final column of Table 3 lists the p-values of this test, showing that the effect is highly significant for all six cases.

The results, however, do not support our hypothesis that the level of nesting has an effect on the preference for our system. Throughout all nesting levels, the users expressed clear and significant preference for our system, but this preference did not increase with increased nesting levels.

6 Conclusion and Future Work

SPARQL Transformer offers to Web developers a different way of approaching RDF datasets. The adoption of a novel JSON format for defining both the query and the template makes it possible to realise self-contained files. When collected in a GitHub repository, these files can be easily transformed into Web APIs with `grlc`, completing the decoupling between query, post-processing and consumption in the application, and query results can moreover be presented in a simple and user-friendly manner via Tapas. The evaluation reveals that the restructuring and merging pipeline of SPARQL Transformer has an important impact in making the SPARQL results more usable and understandable by humans.

Differently from other works, SPARQL Transformer allows developers to use one single file for querying and mapping, and even with some limits – i.e. not being as expressive as SPARQL – can be of benefit for fast prototyping of web application.

Further development can improve SPARQL Transformer in order to fulfil a wider range of needs. The query support can be extended to other SPARQL operations, like ASK, INSERT and DELETE, going towards the realisation of full REST APIs on top of SPARQL endpoints. Aggregate functions (e.g. COUNT, SUM) should join the set of available features in the near future. We will further investigate the use of JSON frames, in order to extract the Shaper component from the library and make it available for standalone use.

Currently, the JSON syntax does not foresee any standard way for representing dates, which are therefore represented as plain strings. Alternative representations for dates should be found taking into account developer requirements, even listening and involving them in the final decision. Possibly, the solution should also involve other related data-types, like `xsd:gYear` or `xsd:duration`.

We plan to run another evaluation of this work, this time focused on the creation scenario, consisting in an interview on query writing with SPARQL Transformer and on API management with `grlc`.

Finally, we are currently planning to offer more customisation possibilities to users. Some examples include the choice of a different merging anchor (currently forced to `id` or `@id`); the possibility of ignoring language tags in the results (avoiding the presence of a language-value object); and the chance of distinguishing between IRIs (as resource references) and IRIs in lexical forms.

Acknowledgements. This work has been partially supported by the European Union's Horizon 2020 research and innovation program within the SILKNOW (grant agreement No. 769504) and MeMAD (grant agreement No. 780069) projects, and by the CLARIAH project of the Dutch Science Foundation (NWO). We want to thank Ilaria Tiddi for her support and suggestions on combining our work.

References

1. Abburu, S., Babu, G.S.: Format SPARQL query results into HTML report. Int. J. Adv. Comput. Sci. Appl. (IJACSA) **4**(6), 144–148 (2013)
2. Bergwinkl, T., Luggen, M., elf Pavlik, Regalia, B., Savastano, P., Verborgh, R.: Interface Specification: RDF Representation, Draft Report. Technical report, W3C (2017)
3. Booth, D., Chute, C.G., Glaser, H., Solbrig, H.: Toward easier RDF. In: W3C Workshop on Web Standardization for Graph Data, Berlin, Germany (2019)
4. Corby, O., Faron-Zucker, C., Gandon, F.: A generic RDF transformation software and its application to an online translation service for common languages of linked data. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 150–165. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_9
5. Corby, O., Faron-Zucker, C., Gandon, F.: LDScript: a linked data script language. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10587, pp. 208–224. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68288-4_13
6. Daga, E., Panziera, L., Pedrinaci, C.: A BASILar approach for building web APIs on top of SPARQL endpoints. In: International Workshop on Services and Applications over Linked APIs and Data (SALAD), vol. 1359. CEUR Workshop Proceedings, Bethlehem (2015)
7. ECMA International: ECMAScript 2015 Language Specification, 6th edn, ECMA-262. Technical report, ECMA International (2015)
8. Fielding, R., et al.: Hypertext transfer protocol (HTTP/1.1): Header Field Definitions, RFC 2616. Technical report, Internet Engineering Task Force (2014)
9. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. Thesis (2000)
10. Gandon, F., et al.: Graph data on the web: extend the pivot don't reinvent the wheel. In: W3C Workshop on Web Standardization for Graph Data, Berlin, Germany (2019)
11. Groth, P., Loizou, A., Gray, A.J., Goble, C., Harland, L., Pettifer, S.: API-centric linked data integration: the open PHACTS discovery platform case study. Web Semant.: Sci. Serv. Agents World Wide Web **29**, 12–18 (2014)
12. Harris, S., Seaborne, A.: SPARQL 1.1 query language - W3C recommendation. Technical report, W3C (2013)
13. Huelss, J., Paulheim, H.: What SPARQL query logs tell and do not tell about semantic relatedness in LOD. In: Gandon, F., Guéret, C., Villata, S., Breslin, J., Faron-Zucker, C., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9341, pp. 297–308. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25639-9_44

14. Lathuilière, M.: `Wikidata SDK` (2015). https://github.com/maxlath/wikidata-sdk
15. Lisena, P., Troncy, R.: Transforming the JSON output of SPARQL queries for linked data clients. In: International Conference Companion on World Wide Web (WWW Companion), pp. 775–780. International World Wide Web Conferences Steering Committee, Lyon (2018). https://doi.org/10.1145/3184558.3188739
16. Meroño-Peñuela, A., Hoekstra, R.: grlc makes GitHub taste like linked data APIs. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 342–353. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47602-5_48
17. Mynarz, J.: `sparql-to-jsonld` (2016). https://github.com/jindrichmynarz/sparql-to-jsonld
18. Nielsen, J.: Usability Engineering. Elsevier, Amsterdam (1994)
19. Ockeloen, N., de Boer, V., Aroyo, L.: LDtogo: a data querying and mapping frameworkfor linked data applications. In: Cimiano, P., Fernández, M., Lopez, V., Schlobach, S., Völker, J. (eds.) ESWC 2013. LNCS, vol. 7955, pp. 199–203. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41242-4_24
20. Pedrinaci, C., Domingue, J.: Toward the next wave of services: linked services for the web of data. J. Univ. Comput. Sci. **16**(13), 1694–1719 (2010)
21. Rietveld, L., Hoekstra, R.: Man vs. machine: differences in SPARQL queries. In: 4th Workshop on Usage Analysis and the Web of Data (USEWOD), Anissaras, Greece (2014)
22. Rietveld, L., Hoekstra, R.: The YASGUI family of SPARQL clients. Semant. Web **8**(3), 373–383 (2017)
23. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_15
24. Seaborne, A.: SPARQL 1.1 query results JSON format - W3C recommendation. Technical report, W3C (2013)
25. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: Antoniou, G., et al. (eds.) ESWC 2011. LNCS, vol. 6643, pp. 170–184. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21034-1_12
26. Taelman, R., Vander Sande, M., Verborgh, R.: GraphQLLD: linked data querying with GraphQL. In: 17th International Semantic Web Conference (ISWC), Poster & Demo Track, Monterey, California, USA (2018)
27. Taelman, R., Vander Sande, M., Verborgh, R.: Bridges between GraphQL and RDF. In: W3C Workshop on Web Standardization for Graph Data, Berlin, Germany (2019)
28. Verborgh, R.: Decentralizing the semantic web through incentivized collaboration. In: 17th International Semantic Web Conference (ISWC), Blue Sky Track, vol. 2189, October 2018
29. Wright, A., Andrews, H.: JSON schema: a media type for describing JSON documents. Technical report, Internet Engineering Task Force (2017). https://datatracker.ietf.org/doc/draft-handrews-json-schema/
30. Zaveri, A., et al.: smartAPI: towards a more intelligent network of web APIs. In: Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O. (eds.) ESWC 2017. LNCS, vol. 10250, pp. 154–169. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58451-5_11