# Simplified SPARQL REST API
## CRUD on JSON Object Graphs via URI Paths

Markus Schröder[1], Jörn Hees[1], Ansgar Bernardi[1],
Daniel Ewert[2], Peter Klotz[2], and Steffen Stadtmüller[2]

[1] German Research Center for Artificial Intelligence GmbH (DFKI)
{markus.schroeder, joern.hees, ansgar.bernardi}@dfki.de
[2] Robert Bosch GmbH
{daniel.ewert, peter.klotz, steffen.stadtmueller}@de.bosch.com

**Abstract.** Within the Semantic Web community, SPARQL is one of the predominant languages to query and update RDF knowledge. However, the complexity of SPARQL, the underlying graph structure and various encodings are common sources of confusion for Semantic Web novices. In this paper we present a general purpose approach to convert any given SPARQL endpoint into a simple to use REST API. To lower the initial hurdle, we represent the underlying graph as an interlinked view of nested JSON objects that can be traversed by the API path.

**Keywords:** SPARQL, REST API, URI, JSON, CRUD, Query, Update

## 1 Introduction

Nowadays, the majority of developers already know how to use web technologies such as REST APIs and JSON. However, in order to use Semantic Web technologies, they typically still need extensive additional training. Before being able to perform simplistic CRUD (create, read, update, delete) workflows, they first need to learn about RDF basics, URIs, Literals, BNodes and how they're used to model knowledge as a graph of triples (as opposed to more often used JSON representations). Further, in the process newcomers are overwhelmed with a multitude of encodings, serialization and result formats, before finally being able to interact with a triple store via SPARQL Update (and to understand what they are doing).

In this paper, we present an approach that aims to reduce this initial hurdle to use semantic technologies: We would like to allow Semantic Web newcomers to interact with a SPARQL endpoint without requiring them to go through extensive training first. To reach this goal, our approach transforms and simplifies a given SPARQL endpoint into a generic path based JSON REST API. During the design, we focused on simple CRUD workflows. To reduce complexity, we decided against attempting to cover all SPARQL capabilities, but instead provide a trade-off between simplicity and expressivity. We use an easy to understand path metaphor to translate REST calls into corresponding SPARQL queries. Users can conveniently follow connections between the returned object views by

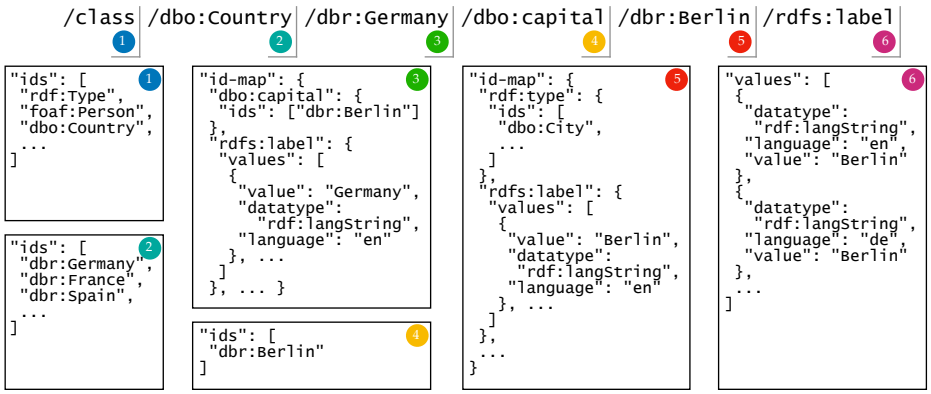


**Fig. 1.** Illustration of basic REST API usage. Shown are excerpts of the JSON results for the different paths. Returned ids can conveniently be chained to walk the graph.

iteratively extending the path of their requests. The resulting SPARQL response is translated back into an easy to understand and possibly nested JSON format.

A simple example can be found in Figure 1. To get a list of all countries, users can access `/class/dbo:Country`. Chaining one of the returned ids, they can then access `/class/dbo:Country/dbr:Germany` and subsequently `/class/dbo:Country/dbr:Germany/dbo:capital` to arrive at `/class/dbo:Country/dbr:Germany/dbo:capital/dbr:Berlin` (or alternatively `/resource/dbr:Berlin`).

An online demo can be found at `http://purl.com/sparql-rest-api`.

## 2 Related Work

Similarly to Battle and Benson [1] our approach offers simple access based on class and resource entry points. However, our approach significantly extends the expressiveness by allowing arbitrary length paths to walk the graph and using wildcards and property paths. Further our serialization format consequently abstracts away from triples, as will be detailed in the following section.

Anticipating SPARQL Update, Wilde and Hausenblas [7] discuss application of REST to SPARQL, but users would still have to master SPARQL and write RDF statements. The BASILar approach [2] builds Web APIs on top of SPARQL endpoints by generating predefined REST resources. Similarly, grlc [6] builds Web APIs from SPARQL queries stored on GitHub. [3] describes a mapping approach from CRUD HTTP requests to SPARQL queries while [4] maps predefined REST calls to SPARQL queries, too. However, to work, these approaches need manual, up front definitions of mappings or SPARQL queries. Our API path to SPARQL mapping can be compared with RDF Path languages[3].

Summarizing, in contrast to the mentioned works our approach provides a zero-configuration REST to SPARQL conversion (evaluated during runtime) and uses easily understandable JSON objects (instead of triples).

[3] `https://www.w3.org/wiki/RdfPath`

## 3 Approach

As mentioned in the introduction, our approach aims to reduce complexity for Semantic Web newcomers by providing a simple to use, generic path based JSON REST API interface for a given SPARQL endpoint[4]. To achieve this, we (bidirectionally) translate[5] between the RDF predominant way of modelling knowledge in graph form and object oriented knowledge representations. The latter allows an easy to understand, nested serialization as JSON, which plays well together with our path walking metaphor.

**Entry Points and Path Structure.** Focusing on common use-cases, our API interface provides two entry points: `/class` and `/resource`. The former is used to browse instances of a known class, while the latter queries a resource by its known CURIE. An excerpt of the API's path grammar is shown in Listing 1.1. We allow arbitrary length traversal of the underlying graph by using the easy to understand folder metaphor: By alternating resources (RES) and properties (PROP) using the PATH rule we permit users to navigate the graph.

**Listing 1.1.** Excerpt of the API Path Grammar

```
API_PATH = "/api" (CLASS | RESOURCE)
CLASS =    "/class" RES PATH RQL?
RESOURCE = "/resource" PATH RQL?
PATH =     ("/" RES "/" PROP)* ("/" RES)?
```

**Resulting JSON Syntax.** As result format, we refrain from using JSON-LD, as it is an RDF (triple) serialization format driven by RDF specifics and often confuses novices expecting a simple, nested, object oriented format. Instead, we use simple JSON objects, which are modeled along the outgoing edges of a node `x` (so all triples of the form `x ?p ?o`). We refer to nodes as "ids" using their CURIEs, to stress that users are not required to actually know that these are CURIEs or URIs (to them it is just an identifier). In simple use-cases the JSON object response contains one of two JSON array fields (`ids` or `values`) or a JSON object `id-map`, as can be seen in Figure 1. For consistency reasons, single id or value results are represented as arrays as well.

Resources are listed in `ids` using their CURIEs. This reduces the need for escaping in paths, and generates more readable paths allowing easy manual entry. Literals are listed in `values`. To allow correct round-tripping, each of them is represented as object containing its value, language and datatype, inspired by the SPARQL JSON result representation. `id-map`s are used to map ids (resources or properties) to further components, for example to list a resource's properties. As we allow unbounded nested results, `id-map` naturally contain `ids`, `values`, `id-map` and `value-map` (see further below).

**HTTP CRUD Methods.** Besides GET, our API supports POST, PUT and DELETE requests. The payload of POST and PUT is expected to be of the same structure as the GET results, allowing seamless round-trips. In general, all

[4] Our prototype only needs the endpoint's URI and a list of predefined namespaces.
[5] Transforming the API path to SPARQL and the result sets back to our JSON format.

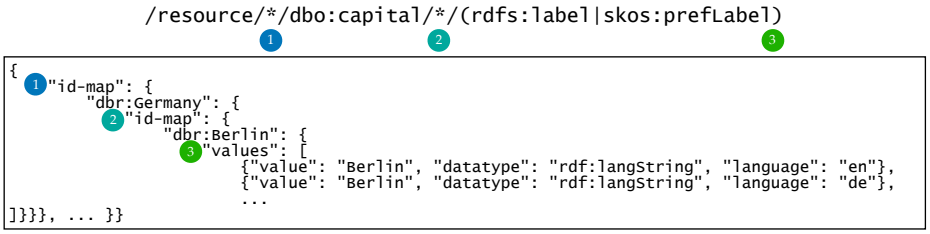


**Fig. 2.** Example of path extensions using wildcards and a property path: The example lists all object ids with an outgoing `dbo:capital` edge, their corresponding linked object ids and their corresponding `rdfs:label` or `skos:prefLabel` as values. As can be seen each `*` introduces an extra nesting level, while property paths (as in SPARQL) are transparently collapsed in the result.

modifying requests extend information from the body with that encoded in the path (e.g., `/class/X/x` implicitly adds the triple `x a X.`). POST requests create a new resource and generate (and return) a new random identifier for the object specified in the body, while PUT requests create or update an object identified by the path. Depending on the path depth, DELETE requests delete a full resource (depth: 1, all in- and outgoing edges), the specified outgoing properties (depth: 2) or one specific triple (depth: 3).

**Extended Expressiveness: Wildcards, Property Paths & RQL.** Apart from these basic features, we extended our API with a couple of noteworthy features that seamlessly integrate into the path metaphor. A very powerful feature is the well known Bash wildcard `*`, which we allow in any RES and PROP position in the path. The asterisk is interpreted in an "all of them" way, introducing an additional nesting level for each asterisk in the resulting JSON. It allows to quickly create a partial view of nested objects, as can be seen in Figure 2.

As also shown in Figure 2, we additionally permit the use of SPARQL property paths[6] in every PROP position by using surrounding brackets (e.g. `/:x/(foaf:name|rdfs:label)/`). Reminding of regular expressions, this enables queries containing alternatives, inverse directions and multiple hops. Similarly to SPARQL, the followed property path is not shown in the result (collapsed).

Combining wildcards and inverse property paths also allows us to step over literals: For example, `/resource/*/foaf:name/*/(^rdfs:label)` will list all object ids that link to values via `foaf:name`, the corresponding Literals, and (other) object ids which use the same literal as `rdfs:label`. As Literals are complex objects they cannot appear as keys in JSON syntax. Hence, we introduce a last additional keyword to our JSON result format: `value-map`, which represents mappings of values to further components as a list of pairs.

Apart from wildcards and property paths, we allow the API paths to be extended with Resource Query Language (RQL)[7] methods, such as `regex`, `sort`, `limit` and aggregations like `count`, `sum` and `avg`.

[6] `https://www.w3.org/TR/sparql11-query/#propertypaths`
[7] `https://github.com/persvr/rql`

**Further Features: Batch & BNode Handling.** We additionally implemented batch processing in order to bundle many similar requests into one and to reduce connection overhead. To avoid URI length restrictions and because processing a batch usually is a procedure, we implement it via JSON-RPC[8]. Moreover, a `/namespace` entry point can be used to to resolve prefixes, e.g. `/namespace/rdfs,owl`. Our API handles BNodes via their fixed (skolem) URIs as mentioned in [5] and supported by many triplestores (e.g., CURIE `_:b1` ⇔ URI `<_:b1>`)). This allows users to use and traverse BNodes like normal URIs.

## 4 Conclusion and Outlook

In this paper we presented an approach to turn any given SPARQL endpoint into a simple to use JSON REST API. To achieve this, our approach translates between CRUD API requests and SPARQL (Update) queries. The API paths allow users to simply navigate the underlying graph to their point of interest. The paths further allows wildcards and SPARQL property path components, seamlessly integrated in the API as deeper nestings of the resulting JSON.

While the development of our approach already embeds a lot of user feedback, in the future we would like to enhance our approach by adding further ideas. Apart from improvements in the areas of error messages and content negotiation, we especially would like to focus on supporting named graphs, introducing path based permissions and easy to use CRUD for simplistic TBox management.

A live demo showing various examples and their corresponding generated SPARQL queries, the source code, API docs and further information are available online: `http://purl.com/sparql-rest-api`

## References


1. Battle, R., Benson, E.: Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST): Semantic Web and Web 2.0. Web Semantics: Science, Services and Agents on the World Wide Web pp. 61–69 (2008)
2. Daga, E., Panziera, L., Pedrinaci, C.: A BASILar approach for building web APIs on top of SPARQL endpoints. CEUR Workshop Proceedings 1359, 22–32 (2015)
3. Garrote, A., García, M.N.: RESTful writable APIs for the web of linked data using relational storage solutions. CEUR Workshop Proceedings 813 (2011)
4. Hopkinson, I., Maude, S., Rospocher, M.: A simple API to the KnowledgeStore. CEUR Workshop Proceedings 1268, 7–12 (2014)
5. Mallea, A., Arenas, M., Hogan, A., Polleres, A.: On Blank Nodes. In: The Semantic Web - ISWC 2011. LNCS, vol. 7031, pp. 421–437. Bonn (2011)
6. Meroño-Peñuela, A., Hoekstra, R.: Grlc makes github taste like linked data APIs. Lecture Notes in Computer Science 9989, p. 342–353 (2016)
7. Wilde, E., Hausenblas, M.: RESTful SPARQL? You Name It! — Aligning SPARQL with REST and Resource Orientation. WEWST 2009 pp. 39–43


[8] `http://www.jsonrpc.org/specification`