SPARQL Micro-Services: Lightweight Integration of Web APIs and Linked Data

Franck Michel
University Côte d'Azur,
CNRS, Inria, I3S, France
franck.michel@cnrs.fr

Catherine Faron-Zucker
University Côte d'Azur,
Inria, CNRS, I3S, France
faron@i3s.unice.fr

Fabien Gandon
University Côte d'Azur,
Inria, CNRS, I3S, France
fabien.gandon@inria.fr

ABSTRACT

Web APIs are a prominent source of machine-readable information. We hypothesize that harnessing the Semantic Web standards to enable automatic combination of Linked Data and non-RDF Web APIs data could trigger novel cross-fertilization scenarios. To achieve this goal, we define the SPARQL Micro-Service architecture. A SPARQL micro-service is a lightweight, task-specific SPARQL endpoint that provides access to a small, resource-centric, virtual graph, while dynamically assigning dereferenceable URIs to Web API resources that do not have URIs beforehand. The graph is delineated by the Web API service being wrapped, the arguments passed to this service, and the restricted types of RDF triples that this SPARQL micro-service is designed to spawn. In this context, we argue that full SPARQL expressiveness can be supported efficiently without jeopardizing servers availability. Eventually, we believe that an ecosystem of SPARQL micro-services could emerge from independent service providers, enabling Linked Data-based applications to glean pieces of data from a wealth of distributed, scalable and reliable services. We describe an experimentation where we dynamically augment biodiversity-related Linked Data with data from Flickr, MusicBrainz and the Macauley scientific media library.

KEYWORDS

Web API, SPARQL, micro-service, Data Integration, Linked Data, JSON-LD

ACM Reference Format:
Franck Michel, Catherine Faron-Zucker, and Fabien Gandon. 2018. SPARQL Micro-Services: Lightweight Integration of Web APIs and Linked Data. In *Proceedings of Linked Data on the Web.* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Web APIs are a common means for Web portals and data producers to enable HTTP-based, machine-processable access to their data. They are a prominent source of information[1] pertaining to topics as diverse as entertainment, finance, social networks, government or scientific information. Many of them follow a REST or "REST-like" architecture[2] ensuring a relatively uniform access to resource descriptions. In a similar perspective, Linked Data [9] seek the publication of machine-readable data on the Web, but go one step beyond. Connecting related RDF datasets brings about the ability to query and make sense of distributed knowledge graphs, provided that these be accessible through standard interfaces ranging from sheer RDF dumps to full-fledged SPARQL endpoints. Here too, large amounts of data about all sorts of topics are openly available[3].

We hypothesize that harnessing the Semantic Web standards to enable automatic combination of disparate resources representations coming from both Linked Data interfaces and Web APIs could trigger novel cross-fertilization scenarios. Major initiatives such as Google's Knowledge Graph[4] or Facebook's Open Graph[5] leverage these two worlds (alongside other types of data sources) to come up with vast knowledge graphs. Strikingly enough though, standard approaches still lack in this domain, as several issues must be overcome:

- **Vocabularies**: Web APIs typically produce resource descriptions using negotiated media types such as JSON or XML, that however often rely on proprietary vocabularies. These vocabularies are commonly documented in Web pages meant for application developers, but they are hardly machine-readable. Conversely, Linked Data best practices [10] advocate the use of common, well adopted vocabularies described in machine-readable format. Consequently, consuming Web API data as RDF triples often leads to the development of wrappers implementing bespoke vocabulary alignment.
- **Parsimony**: Web APIs commonly consist of many different services (search by name/tag/group, organize content, interact with contacts, etc.). Providing a Linked Data interface for all of these services may require a significant effort, although a tiny fraction of them may fulfill the needs of most use cases. Therefore, a more parsimonious, on-demand approach may be more relevant.
- **Trades-offs**: each type of Linked Data interface has its own benefits and concerns. RDF dumps allow in-house consumption but do not fit in when data change at a high pace; URI dereferencing provides subject-centric documents hence lacking query expressiveness; SPARQL [8] is more expressive but puts the query processing cost solely on the server, and it has been shown that allowing clients to run arbitrary SPARQL queries against public endpoints leads to unsatisfactory availability [2]. Besides, on-the-fly SPARQL querying of non-RDF legacy data proves to be difficult, as attested by the many works on SPARQL-based access to relational [20, 25] or NoSQL [18, 22] databases.

Verborgh et al. defined a framework to analyze and compare different Linked Data query interfaces [28]: a response to a query

[1]19,000+ Web APIs are registered on ProgrammableWeb.com as of January 2018.
[2]REST-like refers to loosely-defined architectures complying with some REST principles but relaxing others.

[3]Almost 10,000 knowledge graphs are inventoried by LODStats as of January 2018 (http://lodstats.aksw.org/).
[4]https://goo.gl/BqMC21
[5]http://ogp.me/

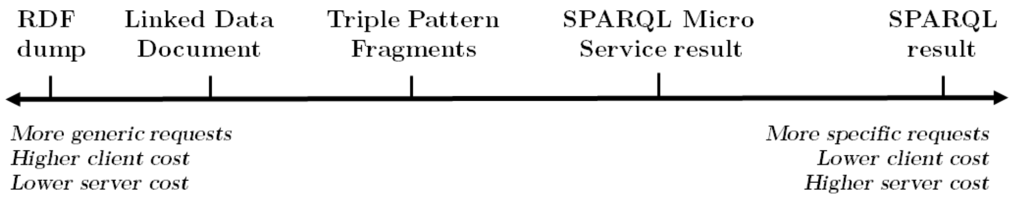


Figure 1: Granularity of different types of *Linked Data Fragments* returned by different Linked Data query interfaces

against such an interface is called a *Linked Data Fragment* (LDF). Different LDF types can be sorted by the granularity of their querying mechanism, as depicted on Figure 1: on the left hand-side, querying consists of a mere download operation and clients bear the full cost of evaluating queries against RDF dumps; on the right hand-side, SPARQL enables expressing specific queries but endpoints fully bear the evaluation cost. Between these two extremes lies a spectrum of intermediate approaches: a Linked Data document results from a URI dereferencing; a *Triple Pattern Fragment* [28] (TPF) results from evaluating a single triple pattern against an RDF dataset. TPFs succeed in balancing the query evaluation cost between the client and the server, thereby enabling better server availability and efficiency than full-fledged SPARQL endpoints. Still, we can think of solutions wherein servers can compute more expressive queries (more than just a triple pattern) without jeopardizing their sustainability.

In this paper, we address the specific goal of combining Web APIs data and Linked Data. We propose an additional type of Linked Data Fragment interface called *SPARQL Micro-Services*, by analogy with the popular micro-services architecture that structures an application as a collection of lightweight, loosely-coupled services designed for a specific task [23]. A SPARQL micro-service is a lightweight method to query a Web API using SPARQL. It provides access to a small, resource-centric, virtual graph, while dynamically assigning dereferenceable URIs to Web API resources that do not have URIs beforehand. This graph corresponds to a fragment of the whole dataset served by the Web API, delineated by (i) the Web API service that is bridged by this SPARQL micro-service; (ii) the arguments passed to this Web API service; and (iii) the restricted types of RDF triples that this SPARQL micro-service is designed to spawn. Since a query is evaluated against a somewhat limited graph, its processing cost can be kept low. Hence, unlike Triple Pattern Fragments, SPARQL micro-services allow full SPARQL expressiveness without risking server overloading. They are depicted on Figure 1, half-way between TPF and SPARQL.

The rest of this article is organized as follows. Section 2 first reminds the micro-service architectural principles and sketches how these could apply to the development of Linked Data-based applications. Then, sections 3 and 4 define in more details the SPARQL micro-service architecture. In sections 5 and 6 we present our implementation and the experimentation we have conducted. Related works are discussed in section 7 while the last section sums up our approach and suggests future leads.

2 MICRO-SERVICE AND LINKED DATA-BASED APPLICATIONS

The term *micro-service* refers to an architectural style, also called *fine-grained SOA*, where an application consists of a collection of services that are loosely coupled, fine-grained (designed to fulfill a single function) and independently deployable [23]. There is no standard definition of micro-services at this point, however a consensus is emerging about commonly agreed upon principles [5, 30]. Compared to traditional monolithic applications, the micro-service architectural style improves modularity by making applications easier to develop, maintain and test. Development teams are typically geared towards continuous refactoring, delivery and deployment of the services they are responsible for, independently of other teams. More complex services are achieved by composition on top of these fine-grained services.

Micro-services have been increasingly adopted during the last six to seven years, not only by major Web companies who inspired this architecture, but also by many other companies that need to speed up the development and deployment of their services. Like any architecture style, the experience shows that micro-services have pitfalls of their own[6]. In particular, figuring out the appropriate size of a micro-service (its functional scope) is critical in many aspects. Nevertheless, leveraging these principles may help in the design of distributed, modular Linked Data-based applications structured as a collection of lightweight, loosely-coupled services. These services would typically be RDF stores, URI dereferencing services, SPARQL endpoints, Linked Data Platform [26] compliant services, etc. Lightweight container technologies, such as the popular Docker[7], could underpin the quick and elastic deployment of such Linked Data-based applications, enabling on-demand scaling up or down by simply starting or stopping instances of these services.

The SPARQL micro-service architecture is a proposition towards this goal. We think of SPARQL micro-services as independent software units being developed along with the arising of needs. A micro-service development team focuses on one Web API at a time, defines how to wrap a few services, tests and publishes them. Instead of being constrained to use specific technologies within a monolithic application, the team picks the programming language(s), Semantic Web stack and other third-party technologies that it deems most appropriate for a specific service.

[6]https://www.infoq.com/news/2014/08/failing-microservices
[7]https://www.docker.com.

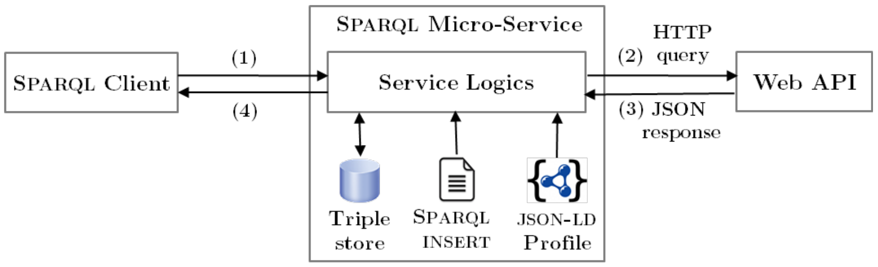


Figure 2: Example implementation of a SPARQL micro-service. A JSON-LD profile interprets the Web API JSON response as a temporary graph G stored in the local RDF triple store. An INSERT query optionally augments G with RDF triples that JSON-LD cannot yield. Lastly, the client's query is evaluated against G.

3 THE SPARQL MICRO-SERVICE ARCHITECTURE

3.1 Definition

A SPARQL micro-service S_μ is a wrapper of a service S_w of a Web API. It behaves as a regular SPARQL endpoint insofar as it supports the SPARQL Query Language [8] (including all query forms: SELECT, ASK, CONSTRUCT, DESCRIBE) and the SPARQL Protocol [6]. It is thus invoked over HTTP/HTTPS by passing a SPARQL query (argument *query* in the GET or URL-encoded POST methods, or HTTP payload in the direct POST method) and optional default and named graph URIs (arguments *default-graph-uri* and *named-graph-uri*).

Additionally, S_μ accepts a possibly empty set Arg_w of arguments that are specific to the service being wrapped. They are passed to S_μ as parameters on the HTTP query string; in turn, S_μ passes them on to S_w. Accordingly, while the URL of a regular SPARQL endpoint is typically of the form "http://hostname/sparql", a SPARQL micro-service is invoked with additional parameters, like "http://hostname/sparql?param1=value1&param2=value2".

The semantics of a SPARQL micro-service differs from that of a standard SPARQL endpoint insofar as the SPARQL protocol treats a service URL as a black box, *i.e.* it does not identify nor interprets URL parameters in any way. By contrast, a SPARQL micro-service is a configurable SPARQL endpoint whose set Arg_w of arguments delineates the scope of the virtual graph that is being queried. Thence, each pair (S_μ, Arg_w) is a standard SPARQL endpoint. As we see it, the graph accessed through (S_μ, Arg_w) is typically resource-centric and corresponds to a small fragment of the dataset served by the Web API.

Figure 2 illustrates the SPARQL micro-service architecture in the case of the implementation we describe further on in section 5. S_μ evaluates a SPARQL query against an RDF graph that it builds at run-time following Algorithm 1 (the algorithm's steps are depicted on Figure 2). Note that how the transformation at step (3) is carried out is implementation-dependent. In the case of JSON-based Web APIs, our implementation first applies a JSON-LD profile to the Web API response and optionally executes a SPARQL INSERT query to yield additional triples. However, various other methods may be used at this point, involving *e.g.* different flavors of mapping, reasoning or rule processing.

Algorithm 1 Evaluation of a SPARQL query by a SPARQL micro-service S_μ.

```
(1) $S_\mu$ receives a SPARQL query $Q$ along with arguments $Arg_w$.
(2) $S_\mu$ invokes the Web API service $S_w$ with the arguments in
    $Arg_w$. The response is a fragment $F_w$ of the dataset served
    by the Web API.
(3) $S_\mu$ translates $F_w$ into a Linked Data Fragment $F_{LD}$ and loads
    it into a triple store as a temporary graph $G$.
(4) $S_\mu$ evaluates $Q$ against $G$ and returns the result to the client.
```

3.2 Caching Strategy

Querying Web APIs is often a time-consuming task, as suggested by the measures we report in section 6. Thus, when possible, defining caching strategies may be necessary to achieve the performance expected by some applications. There typically exist many syntactical variants of the same SPARQL query. Such that classic HTTP proxy servers set up between SPARQL clients and servers fail to reach efficient cache reuse. By contrast, Web API queries allow a lesser syntactical variability. Therefore, in the context of SPARQL micro-services, enforcing a cache strategy on the Web API side should ensure better cache reuse. Typically, Web API queries could be used to index either the Web API responses or the temporary graphs produced from them.

Furthermore, some Web APIs provide data expiration information (such as the *Expires*, *Cache-Control* and/or *Last-Modified* HTTP headers) on which a caching system can rely to figure out a caching strategy. When no such information is provided, a SPARQL micro-service may authoritatively decide on an appropriate expiration period depending on the type of service being wrapped.

3.3 Errors Management

A typical client's SPARQL query may invoke several SPARQL micro-services simultaneously using multiple SERVICE clauses. The failure of a non-critical micro-service (due *e.g.* to a network error or a Web API error) should not hinder processing the other ones and returning partial results. Hence, a client may use the SILENT keyword to ignore errors from a SPARQL micro-service, and possibly embed SERVICES clauses within OPTIONAL clauses to allow for the production of partial results (this is exemplified later on in Listing 4).

```
{ "photos": {
    "page": 1, "pages": "22689",
    "photo": [
      { "id": "38427227466", "title": "Brooklyn Bridge sunset",
        "owner": "33634811@N07",  "ownername": "Franck Michel",
        "secret": "100fa110d0", "server": "4542", "farm": 5 }
    ]
} }
```

```
prefix api: <http://sms.i3s.unice.fr/schema/>
[] api:photos [
    api:page "1"; api:pages "22689";
    api:photo [
        api:id "38427227466"; api:title "Brooklyn Bridge sunset";
        api:owner "33634811@N07"; api:ownername "Franck Michel";
        api:secret "100fa110d0"; api:server "4542"; api:farm "5".
] ].
```

Listing 1: Snippet from an example Flickr Web API response (top), and equivalent RDF triples in the Turtle syntax [1] produced by applying a simple JSON-LD profile (bottom).

3.4 Example

We now illustrate Algorithm 1 with an example based on Flickr's Web API[8]. Let S_w be the Flickr service that retrieves photos posted in a specific group and having a specific tag[9]. Arg_w (the arguments of S_w) comprises two arguments, *group_id* and *tags*. A SPARQL micro-service S_μ wraps S_w. In step 1, a client willing to retrieve the URLs of photos matching some specific group and tags executes a SPARQL query "`SELECT ?img WHERE {?photo foaf:depiction ?img.}`" against S_μ. Using the SPARQL HTTP GET method [6], S_μ is invoked as follows:

```
http://hostname/flickr/getPhotosByGroupByTag?
  group_id=35034350743@N01&tags=bridge&
  query=SELECT\%20\%3Fimg\%20WHERE\%20\%7B\%3Fphoto \
        \%20foaf\%3Adepiction\%20\%3Fimg.\%7D
```

Interestingly enough, this is precisely the invocation that would be performed by the SERVICE clause in the following SPARQL query:

```
SELECT ?img WHERE {
  SERVICE <http://hostname/flickr/getPhotosByGroupByTag? \
         group_id=35034350743@N01&tags=bridge>
  { ?photo foaf:depiction ?img. }
}
```

In step 2, S_μ invokes S_w with the arguments of Arg_w in addition to other arguments possibly required by the Web API (for the sake of clarity we have left over some technical arguments). Listing 1 (top) sketches a snippet of the response to this query:

```
https://api.flickr.com/services/rest/?
  method=flickr.groups.pools.getPhotos&format=json&
  group_id=35034350743@N01&tags=bridge
```

Step 3 is implementation dependent. The prototype implementation presented in section 5 draws on the method proposed by the JSON-LD specification [27] to interpret regular JSON content as a serialized RDF graph. The resulting graph G is stored into a triple store, and an optional SPARQL INSERT query yields additional triples using relevant domain vocabularies.

Finally, in step 4, S_μ evaluates the client's SPARQL query against G and returns the response in one of the media types supported by the SPARQL client (following a regular content negotiation).

[8]https://www.flickr.com/services/api/

[9]https://www.flickr.com/services/api/flickr.groups.pools.getPhotos.html

3.5 Discussion

In this proposition, a design decision is to pass the arguments of Arg_w to S_μ as HTTP query string parameters. Arguably, other solutions may be adopted. Below we discuss the respective benefits and drawbacks of some alternatives we identified.

A first alternative consists in defining one predicate for each argument, *e.g.*:

```
prefix api: <http://sms.i3s.unice.fr/schema/>
SELECT ?img WHERE {
  SERVICE <http://hostname/flickr/getPhotosByGroupByTag>
  { ?photo foaf:depiction ?img;
           api:group_id "806927@N20";
           api:tags "bridge".
  }
}
```

At a first sight, making the arguments explicit can seem compelling as they can be used in other triples of the graph pattern. Besides, such a SPARQL micro-service is a standard SPARQL endpoint since there is no more variable part in the service endpoint URL. Several concerns should be pointed out however. (i) This solution requires that each SPARQL micro-service be defined along with its own bespoke terms, whereas we seek a solution wherein only terms of well-adopted vocabularies would be exposed. (ii) In this example, the group_id and tags arguments are meaningful for the end user. But some services may require more technical arguments that we typically do not want to define as ontological terms. (iii) Furthermore, this solution questions the nature of the subject to which the arguments are associated. Again, in this specific example, declaring the group_id and tags as properties of the photographic resource ?photo is an acceptable choice, but this would be inappropriate with internal or technical service parameters.

This issue can be solved by associating the arguments to a separate resource depicting the service itself. This is exemplified in the second alternative that, furthermore, defines a vocabulary to pass the arguments in a uniform manner. Note that existing vocabularies may be tapped for that matter, such as Hydra [14] or the HTTP Vocabulary in RDF [12]. In the example below, additional triple patterns define an instance of the hypothetical `api:Service` class, that takes arguments declared with the `api:param` predicate.

```
prefix api: <http://sms.i3s.unice.fr/schema/>
SELECT ?img WHERE {
  SERVICE <http://hostname/flickr/getPhotosByGroupByTag>
  { ?photo foaf:depiction ?img.

    [] a api:Service;
         api:param [ api:name "group_id"; api:value "806927@N20" ];
         api:param [ api:name "tags";     api:value ?tag ].
  }
}
```

A slight variation could state that the service URL itself is an instance of `api:Service`; the arguments would then configure an execution of this service with predicate `api:execution`, *e.g.*:

```
    <http://hostname/flickr/getPhotosByGroupByTag>
       a api:Service;
       api:execution [
         api:param [ api:name "group_id"; api:value "806927@N20" ];
         api:param [ api:name "tags";     api:value ?tag ].
       ].
```

While these alternatives avoid defining new predicates for each micro-service, the additional triples bear a somewhat artificial semantics: they provide the service with information as to how to

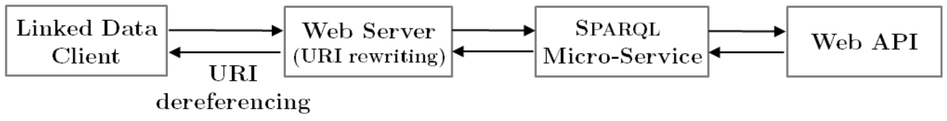


Figure 3: URI dereferencing: a Web server rewrites a URI look-up query into a query to the relevant SPARQL micro-service.

process the other parts of the graph pattern, but they do not actually refer to nor describe the photographic resources that the graph pattern aims to match.

In a third alternative, the service arguments are passed as SPARQL variables with pre-defined names, *e.g.* ?group_id and ?tags in the example below:

```
SELECT ?img WHERE {
  SERVICE <http://hostname/flickr/getPhotosByGroupByTag>
  { ?photo foaf:depiction ?img.

    BIND("806927@N20" AS ?group_id)
    BIND("bridge" AS ?tags)
  }
}
```

Similarly to the previous alternative, variables ?group_id and ?tags are artificial insofar as they provide the service with information as to how to process the other parts of the graph pattern.

The solution proposed in this article is a trade-off meant to satisfy certain goals. Above, we have discussed some alternative solutions, and others may probably be figured out. We believe that further discussions should be engaged to assess the benefits and concerns of these alternatives with respect to the contexts and goals.

4 ASSIGNING URIS TO WEB API RESOURCES

Web APIs often provide resource descriptions that rely on internal, proprietary identifiers. For instance, Listing 1 mentions the photo identifier "38427227466" that has no meaning beyond the scope of Flickr's Web API. One may argue that the URL of this photo's Web page could serve as a URI, but this would just approve a questionable practice that confuses a resource (a photographic work in this case) with an HTML representation thereof.

Therefore, bridging Web APIs and Linked Data not only requires to enable SPARQL querying of Web APIs, but also to dynamically create URIs that identify Web API resources. Furthermore, according to Linked Data best practices [10], it should be possible to look up these URIs in order to retrieve a description of the resources in a negotiated media type. Conventionally, dereferencing a URI returns a set of RDF triples where the URI is either in the subject or object positions. This is typically achieved through a CONSTRUCT or DESCRIBE SPARQL query form.

Implementing URI dereferencing with SPARQL micro-services is straightforward. It only requires two things:
(i) Decide on the domain name and URIs scheme. Following up on the example in section 3, we define the URI scheme of Flickr photo resources as "http://example.org/ld/flickr/photo/<photo_id>".
(ii) Set up a Web server to deal with this URI scheme. When the Web server receives a look-up for a URI that matches the scheme, it dynamically builds a query string to invoke the relevant SPARQL micro-service. Technically, the Web server acts as a reverse proxy: it queries the SPARQL micro-service and transparently proxies the response back to the client. Figure 3 sketches this architecture.

Let us assume the *getPhotoById* SPARQL micro-service retrieves photos by their identifier. If the Web server receives a look-up for the URI "http://example.org/ld/flickr/photo/38427227466", it invokes the *getPhotoById* micro-service with the following query string:

```
http://hostname/flickr/getPhotosById?
  photo_id=38427227466&
  query=DESCRIBE\%20\%3Chttp\%3A\%2F\%2Fexample.org\%2Fld \
        \%2Fflickr\%2Fphoto\%2F38427227466\%3E
```

The value of argument "query" is the URL-encoded string for: `DESCRIBE <http://example.org/ld/flickr/photo/38427227466>`. The response to this query will be proxied back to the client in one of the negotiated media types.

Hence, by smartly designing SPARQL micro-services, we can build a consistent ecosystem where some micro-services respond to SPARQL queries by translating Web API internal identifiers into URIs, and some micro-services (possibly the same) are able to dereference these URIs.

5 IMPLEMENTATION

To evaluate the architecture proposed in section 3, we have developed a prototype implementation in the PHP language, available on Github[10] under the Apache 2.0 license. This prototype complies with the choices pictured in Figure 2: it assumes that Web APIs are able to provide a JSON response, and thence requires each SPARQL micro-service to provide a JSON-LD profile as a first mapping step towards RDF.

SPARQL queries. The processing of SPARQL queries is implemented as depicted in Figure 2. First, a JSON-LD profile is applied to the Web API JSON response. Following up on the example introduced in section 3, S_μ applies to the API response the JSON-LD profile below, that turns each JSON field name into an ad hoc RDF predicate within namespace "http://sms.i3s.unice.fr/schema/" (note that any arbitrary complex profile may be used at this point):

```
{"@context": {"@vocab": "http://sms.i3s.unice.fr/schema/"}}.
```

The resulting graph G, depicted in Listing 1 (bottom), is stored in the Corese-KGRAM in-memory triple store [3]. It exhibits proprietary, technical predicates such as servers and farms identifiers, that are likely irrelevant for a Linked Data representation. The Flickr API documentation describes how to reconstruct photos and user pages URLs from these fields, but this involves the concatenation of values from distinct fields, that JSON-LD is typically not expressive enough to describe. Therefore, in a second step, S_μ executes the INSERT query shown in Listing 2 to augment G with triples based on well-adopted or domain-specific vocabularies (Schema.org, foaf and Dublin Core in our example).

[10]https://github.com/frmichel/sparql-micro-service

```
prefix api:    <http://sms.i3s.unice.fr/schema/>
prefix foaf:   <http://xmlns.com/foaf/0.1/>
prefix schema: <http://schema.org/>
prefix dce:    <http://purl.org/dc/elements/1.1/>

INSERT {
  ?photoUri a schema:Photograph;
      foaf:depiction ?img;
      schema:author ?authorUrl; dce:creator ?authorName;
      dce:title ?title.
} WHERE {
  ?photo api:id ?id; api:secret ?secret;
      api:server ?server; api:farm ?farm;
      api:title ?title;
      api:owner ?owner; api:ownername ?authorName.

  BIND (IRI(concat("http://example.org/ld/flickr/photo/", ?id))
        AS ?photoUri)
  BIND (IRI(concat("https://flickr.com/photos/", ?owner))
        AS ?authorUrl)
  BIND (IRI(concat("https://farm", ?farm, ".staticflickr.com/",
          ?server, "/", ?id, "_", ?secret, "_z.jpg")) AS ?img)
}
```

Listing 2: Insertion of RDF triples based on well-adopted vocabularies (Schema.org, FOAF, Dublin Core).

Finally, S_μ evaluates the client's SPARQL query against G and returns the response that, in the example, consists of a solution binding presented below in the SPARQL Results JSON format:

```
{ "head": { "vars": [ "img" ] }, "results": {
    "bindings": [
      { "img": {
          "type": "uri",
          "value": "https://farm5.staticflickr.com/4542/ \
                    38427227466_100fa110d0_z.jpg" }
} ] } }
```

URIs dereferencing. In the URIs dereferencing solution portrayed in section 4, the Web server's URL rewriting engine appends a SPARQL query to the SPARQL micro-service invocation. In our example, this query is simply "DESCRIBE <uri>". Nevertheless, some use cases may require more complex CONSTRUCT queries to produce richer triples. Since the query must be URL-encoded, this method may become cumbersome and difficult to maintain. To avoid this issue, our implementation proposes a more flexible and maintainable alternative: each micro-service may provide a CONSTRUCT query that shall be used to produce the answer to the URI dereferencing query.

Deploying a new SPARQL micro-service. Within this prototype, deploying a new SPARQL micro-service simply consists of provisioning four files among which two are optional:

- *config.ini*: a configuration file that declares the arguments expected by the micro-service and the Web API invocation query string.
- *profile.jsonld*: the JSON-LD profile used to interpret the Web API response as an RDF graph;
- *insert.sparql* (optional): used to augment the graph with additional triples;
- *construct.sparql* (optional): used to produce the response to URI dereferencing queries.

In our experience, deploying a new SPARQL micro-service within our prototype is a matter of one or two hours in simple cases. The most time-consuming task lies in reading the Web API documentation. Thenceforth, a developer defines the API query string and the arguments passed to the SPARQL micro-service. Lastly, she writes the JSON-LD profile and the optional INSERT and CONSTRUCT queries that carry out the mappings toward domain vocabularies.

Our prototype consists of only 350 lines of code, in addition to the RDF and JSON-LD libraries that it relies on. If a Web API requires specific actions that cannot be described using the *config.ini* configuration file (*e.g.* intermediate query, authentication process), the developer can customize a provided script, allowing for more flexibility. The interested reader can look at an example in service *macaulaylibrary/getAudioByTaxon*.

Docker Deployment. In addition to the code available on Github, we have created two Docker images published on Docker hub[11]: one image runs the Corese-KGRAM RDF store and SPARQL endpoint, while the other provides an Apache Web server with the SPARQL micro-services described in section 6. They can be deployed by running a single command, as instructed in the Github README. Note that, for the sake of simplicity, we have defined a single image hosting several micro-services. Nevertheless, more in line with common micro-service practices, it would make sense to define one image per service, enabling the independent deployment of each service.

6 EXPERIMENTATION

To evaluate the effectiveness and efficiency of our approach, we carried out a series of tests related to the use case that initially motivated this work, in the context of biodiversity. In a joint initiative with the French National Museum of Natural History, we have produced a dataset called TAXREF-LD, a Linked Data representation of the French taxonomic register of living beings [19]. TAXREF-LD models over 236.000 taxa as OWL classes while the scientific names used to refer to taxa are modeled as SKOS concepts. It is accessible through a public SPARQL endpoint[12], and all taxa and scientific names URIs are dereferenceable.

To dynamically enrich the description of a taxon in TAXREF-LD with data from various Web APIs, we developed three SPARQL micro-services based on the prototype presented in section 5 (the code is available along with the rest of the project code on Github):

1. *flickr/getPhotosByGroupByTag*, already described in section 3, is used to search the *Encyclopedia of Life* Flickr group[13] for photos of a given taxon. Photos of this group are tagged with the scientific name of the taxon they represent, formatted as "taxonomy:binomial=<scientific name>".
2. *macaulaylibrary/getAudioByTaxon* retrieves audio recordings for a given taxon name from the Macaulay Library[14], a scientific media archive related to birds, amphibians, fishes and mammals.
3. *musicbrainz/getSongByName* searches the MusicBrainz music information encyclopedia[15] for music tunes whose title match a given name with a minimum confidence of 90%.

6.1 Test Setting

We wrote several test queries that invoke TAXREF-LD's SPARQL endpoint along with one, two or three of the SPARQL micro-services

[11]https://hub.docker.com/u/frmichel/
[12]http://taxref.mnhn.fr/sparql
[13]https://www.flickr.com/groups/806927@N20
[14]https://www.macaulaylibrary.org/
[15]https://musicbrainz.org/

```
prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix schema: <http://schema.org/>

SELECT ?species ?audioUrl WHERE {

    SERVICE <https://erebe-vm2.i3s.unice.fr:8890/sparql> {
        ?genus a owl:Class; rdfs:label "Delphinus".
        ?s rdfs:subClassOf ?genus; rdfs:label ?species.
    }

    SERVICE ?sms { [] schema:contentUrl ?audioUrl. }

    # Construction of the SPARQL micro-service endpoint URL
    BIND(IRI(concat(
     "https://erebe-vm2.i3s.unice.fr/sparql-ms/macaulaylibrary/
      getAudioByTaxon?name=", encode_for_uri(?species))) as ?sms)
}
```

Listing 3: Dynamic construction of a SPARQL micro-service endpoint URL: the first SERVICE clause retrieves the species (sub-classes) of genus *Delphinus*. For each species, the second SERVICE clause invokes a SPARQL micro-service to retrieve associated audio recordings.

```
prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix schema: <http://schema.org/>

CONSTRUCT {
  ?species                        # from TAXREF-LD
    schema:subjectOf ?photo;      # from Flickr
    foaf:depiction ?img;          # from Flickr
    schema:contentUrl ?audioUrl;  # from the Macaulay Library
    schema:subjectOf ?page.       # from MusicBrainz
} WHERE {

  SERVICE <https://erebe-vm2.i3s.unice.fr:8890/sparql>
    { ?species a owl:Class; rdfs:label "Delphinus delphis". }

  OPTIONAL {
    SERVICE SILENT <https://erebe-vm2.i3s.unice.fr/sparql-ms/
            flickr/getPhotosByGroupByTag?group_id=806927@N20
            &tags=taxonomy:binomial=Delphinus+delphis>
      { ?photo foaf:depiction ?img. }
    }

  OPTIONAL {
    SERVICE SILENT <https://erebe-vm2.i3s.unice.fr/sparql-ms/
            macaulaylibrary/getAudioByTaxon?name=Delphinus+delphis>
      { [] schema:contentUrl ?audioUrl. }
    }

  OPTIONAL {
    SERVICE SILENT <https://erebe-vm2.i3s.unice.fr/sparql-ms/
            musicbrainz/getSongByName?name=Delphinus+delphis>
      { [] schema:sameAs ?page. }
    }
}
```

Listing 4: SPARQL query *Q* enriches the Linked Data representation of taxon *Delphinus delphis* using SPARQL micro-services to retrieve data from Flickr, the Macaulay Library and MusicBrainz.

mentioned above. Each SPARQL micro-service is invoked within a dedicated SPARQL SERVICE clause. The micro-service endpoint URL is built either dynamically and bound to a SPARQL variable as exemplified in Listing 3, or statically as exemplified in Listing 4.

We used two different SPARQL engines to evaluate the test queries: Corese-KGRAM [3] and Virtuoso OS Edition[16]. The queries in Listings 3 and 4 were properly executed on Corese-KGRAM. By contrast, only queries with static service URLs as in Listing 4 could be executed on Virtuoso. Indeed, Virtuoso cannot evaluate a SERVICE clause wherein the service endpoint is given by a variable. Note that this feature is not in the normative part of the SPARQL 1.1 Federated Query recommendation [24], although it is part of the SPARQL 1.1 grammar. Hence, an implementation may choose to address this feature or not, but the semantics is not formally defined.

The tests were performed on a Scientific Linux 6.9 server running on a virtual machine equipped with 16 CPU cores and 48 GB of RAM. The server hosts a copy of TAXREF-LD's dataset (9.6 million triples) on a Virtuoso OS Edition server 7.20. Its SPARQL endpoint is accessible at URL "https://erebe-vm2.i3s.unice.fr:8890/sparql".

The server also hosts the three SPARQL micro-services. They are accessed at URL "https://erebe-vm2.i3s.unice.fr/sparql-ms/<Web API>/<service name>".

6.2 Test Results

Query *Q*, in Listing 4, consists of a CONSTRUCT query form. It retrieves from TAXREF-LD the URI of the common dolphin species (*Delphinus delphis*) that it enriches with 15 photos retrieved from Flickr, 28 audio recordings from the Macaulay Library, and 1 music tune from MusicBrainz. Figure 4 portrays a snippet of the response to this query in the Turtle syntax, along with photos, audio recordings pictures and a MusicBrainz Web page.

Table 1 reports the execution time of query *Q*, averaged over fifteen runs against the Corese-KGRAM SPARQL engine. Column "*Triples in temp. graph*" gives the number of triples generated (i) by applying the JSON-LD profile to the API response, and (ii) the optional INSERT query (to spawn the triples that JSON-LD cannot yield). Column "*Triples in resp.*" gives the number of triples that match query *Q*'s graph pattern. Strikingly, the Web API query execution is accountable for over 95.6% of the execution time. The overhead imposed by the SPARQL micro-service consistently ranges from 16ms to 30ms, accounting for 2% of the total time for Macauley library's API to 4.35% for Flickr's API.

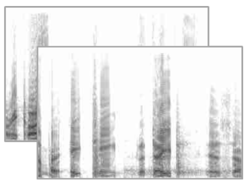
The evaluation of query *Q* takes an average 2.93s ± 0.75. Let us underline that this time is tightly bound to the SPARQL engine's evaluation strategy. The Corese-KGRAM engine evaluates each SERVICE clause sequentially, although they may be evaluated in parallel since there is no dependency between them. By contrast, the Virtuoso SPARQL engine seems to adopt a far worse strategy: not only it invokes the SERVICE clauses sequentially although they do not show any dependency, but more importantly it invokes each SERVICE clause once for each solution retrieved from previously evaluated SERVICE clauses. In our tests, it invoked the Flickr service 28 times, *i.e.* once for each solution of the Macauley library micro-service, thus entailing a very poor evaluation time. At this point, it remains unclear whether this is the result of a specific evaluation strategy or a bug.

The response to query *Q* (Figure 4) contains URIs generated on the fly using Flickr's internal photo identifiers such as "http://erebe-vm2.i3s.unice.fr/ld/flickr/photo/31173090936". When looking them up, an additional SPARQL micro-service dereferences them to an RDF document describing the photo. Hence, we have come up with a consistent set of SPARQL micro-services able to evaluate SPARQL

[16]Virtuoso OS Edition: http://vos.openlinksw.com/owiki/wiki/VOS/

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix schema: <http://schema.org/>

<http://taxref.mnhn.fr/lod/taxon/60878/10.0>
  schema:subjectOf <http://erebe-vm2.i3s.unice.fr/ld/flickr/photo/31173090936>;
  schema:subjectOf <http://erebe-vm2.i3s.unice.fr/ld/flickr/photo/31173091516>;
  ...
  foaf:depiction <https://farm6.staticflickr.com/5617/31173090936_ed65c5a30a_z.jpg>;
  foaf:depiction <https://farm6.staticflickr.com/5567/31173091516_f1c09fa5d5_z.jpg>;
  ...
  schema:contentUrl <https://download.ams.birds.cornell.edu/api/v1/asset/131396/audio>;
  schema:contentUrl <https://download.ams.birds.cornell.edu/api/v1/asset/122066/audio>;
  ...
  schema:subjectOf <https://musicbrainz.org/work/3ffe21ec-7d58-44cb-a65a-5f5876e12b50>.
```



MusicBrainz
Se connecter Créer un compte
Qui sommes-nous ? Produits Recherche
Delphinus delphis
Chanson
Vue d'ensemble Alias Tags Détails Modifier
Relations
compositeur : Austin Wintory
partie de : ABZÛ (video game soundtrack)
Enregistrements
Date Titre Artiste Durée
performance
Delphinus Delphis Austin Wintory 4:51

Figure 4: Snippet of the response to query Q (Listing 4) along with snapshots of the images, audio recordings and Web page whose URLs are part of the response.

Web API	Triples in temp. graph	Triples in resp.	Web API exec. time	SPARQL μ-service exec. time	Overhead	Overhead (percentage)
Flickr	261	15	0.54 ± 0.40	0.56 ± 0.40	0.020 ± 0.002	4.35% ± 1.34
Macauley Lib.	989	28	1.52 ± 0.40	1.55 ± 0.40	0.030 ± 0.001	2.02% ± 0.40
MusicBrainz	42	1	0.62 ± 0.38	0.64 ± 0.38	0.016 ± 0.001	3.16% ± 1.29

Table 1: Query execution time (in seconds) against a Web API and a SPARQL micro-service that wraps this Web API. The last column is the overhead (in %) imposed by the SPARQL micro-service compared to a direct Web API query.

queries and/or dereference URIs, built upon a Web API that has no SPARQL interface nor resource URIs in the first place.

7 RELATED WORKS

Abundant literature has been concerned with the integration of disparate data sources since the early 1990's [29]. Classically, wrappers implement the mediation from the models of multiple data sources towards a target vocabulary or ontology. A query federation engine handles a user query, determining which data sources might have relevant information for the query, deciding on a query plan and recombining the partial results obtained from the multiple sources. Our work is concerned with the wrapping of Web APIs into SPARQL endpoints, but the federation of such wrapped data sources is out of the scope of this paper. Yet, existing federated query technologies could be adapted to rewrite parts of a client's SPARQL query into SERVICE clauses, thereby querying relevant SPARQL micro-services as illustrated in Listing 4.

In the Semantic Web community, works aiming to translate heterogeneous data sources into RDF triples started very early[17]. In [7] for instance, a triple store contains semantic descriptions that are used to semantically annotate Web services, explaining how to invoke them and what kind of information can be obtained by invoking them.

Approaches specifically concerned with Web APIs are often ad hoc solutions. For instance, Flickcurl[18] is a hardwired wrapper for Flickr's Web API services. Twarql [16] wraps Twitter's Web API to enable filtering and analysis of streaming tweets. It encodes tweets content in RDF using common vocabularies and enables SPARQL querying. Yet, this approach is very specific to the Twitter and micropost content in general.

With a similar rationale, Hydra [14] is a vocabulary aimed to describe Web APIs in a machine-readable format. It puts a specific focuses on the generation of hypermedia controls so as to enable the generation of truly RESTful interfaces. Hydra, used in conjunction with JSON-LD, forms a basis to build hypermedia-driven Linked Data interfaces [13]. This basis can be harnessed to turn existing Web APIs into RESTful Linked Data interfaces whose documentation can be interpreted at run time by a generic Hydra client. Our incentive with SPARQL micro-services is to provide client applications with the whole expressiveness of SPARQL, which would be more difficult to achieve using a Hydra-described REST interface. This however remains an interesting lead. For instance, the Triple Pattern Fragment specification [28] leverages Hydra to describe hypermedia controls by means of IRI templates containing triple patterns. In the last section, we suggest the definition of a *Graph Pattern Fragment* micro-service that would comply with the Triple Pattern Fragment hypermedia controls but that would accept regular graph patterns instead of only triple patterns.

SPARQL-Generate [15] extends SPARQL 1.1 to enable querying RDF graphs along with non-RDF documents. A SPARQL-Generate query relies on several extension functions to fetch and parse documents in different data formats, and defines the shape of RDF

[17]See the list hosted on W3C's Web site: https://www.w3.org/wiki/ConverterToRdf
[18]http://librdf.org/flickcurl/

triples to be produced thenceforward. As such, it could be used to query a Web API in a way similar to that of a SPARQL micro-service. Two main differences can be observed though. (i) SPARQL-Generate is an extension of SPARQL, hence, by definition, it is not supported by engines strictly complying with the SPARQL standard. By contrast, our vision is that multiple service providers could publish independent SPARQL micro-services, thereby building up an ecosystem of services all complying with standard SPARQL. (ii) SPARQL-Generate offers the advantage that querying remote data sources is performed within a single language. On the one hand, this only requires skills with Semantic Web technologies. On the other hand, this entails that a significant part of the whole process is left to the SPARQL client: querying the data source providing appropriate arguments, and translating its proprietary vocabulary into RDF triples aligned on common vocabularies. Consequently, as illustrated by authors' examples, the additional syntactic sugar required can make queries considerably cumbersome and difficult to maintain. We take a different option where this complexity is hidden from the client and handled by the SPARQL micro-service developer.

An approach very similar to SPARQL-Generate is proposed in [11]. It is based on the BIND_API clause, an extension of the SPARQL BIND clause, that binds a set of variables with values extracted from a Web API response. It suffers the same pitfalls as SPARQL-Generate with respect to our goals: the use of non standard SPARQL and the cumbersome syntactic sugar left to the SPARQL client.

ODMTP [21], On-Demand Mapping using Triple Patterns, is an attempt to query non-RDF datasets as Triple Pattern Fragments. The authors have implemented a prototype to query Twitter's Web API, that can process triple pattern queries over the whole Twitter's dataset. Conversely, SPARQL micro-services support arbitrary SPARQL queries over restricted fragments of the Web API dataset. Besides, unlike SPARQL micro-services, ODMTP cannot assign dereferenceable URIs to Web API resources. Nevertheless, ODMTP offers the TPF's paging mechanism that SPARQL micro-services should regard as a valuable extension within future works (see the discussion in section 8).

Our implementation of SPARQL micro-services maps a Web API response to RDF triples in two steps: the response is first translated to JSON-LD, then a SPARQL INSERT or CONSTRUCT query complements the process for cases where JSON-LD is not expressive enough. Alternatively, we could rely on a mapping description language such as RML [4] and xR2RML [17], but they require the developer to learn the mapping language. By contrast, in our proposition we strove to rely only on existing standards.

Let us finally mention the Apache Marmotta project[19], a comprehensive Linked Data application framework that implements the Linked Data Platform W3C recommendation [26]. Among others, it provides client modules that wrap the Web APIs of several Web portals such as Vimeo, Youtube and Facebook. Hence, it should be relatively easy to implement SPARQL micro-services on top of Marmotta. However, the examples show that the Web API wrapping and the mapping towards RDF triples are mostly hard-coded within the client libraries. Our point is to make the deployment of new SPARQL micro-services as simple as writing a SPARQL query and a configuration file.

[19]http://marmotta.apache.org/

8 CONCLUSION AND PERSPECTIVES

SPARQL Micro-Services are a lightweight type of Linked Data Fragment interface that enable combining Linked Data with data from non-RDF Web APIs. SPARQL querying and URI dereferencing are supported against a virtual graph delineated by the Web API service being wrapped, the arguments of this service and the types of RDF triples that this SPARQL micro-service can spawn.

Complying with the micro-service architecture principles, a SPARQL micro-service should typically be loosely coupled, fine-grained (it provides access to a small graph centered on a specific resource such as a photograph, a tweet or the temperature of a room) and deployed independently of other services using *e.g.* light container technologies. Eventually, an ecosystem of SPARQL micro-services could emerge from independent service providers, allowing Linked Data-based applications to glean pieces of data from a wealth of distributed, scalable and reliable services.

For such an ecosystem to arise however, two crucial issues shall be tackled. Firstly, to enable services discovery, SPARQL micro-services should provide self-describing metadata such as the expected query string parameters, or the types of triples generated. Secondly, although we envision SPARQL micro-services as a way to access small fragments, it should be possible to retrieve such fragments by smaller pieces using a paging mechanism.

To tackle those issues, Verborgh et al. advocated that Linked Data Fragments should provide self-describing, uniform interfaces consisting of triples, metadata and hypermedia controls. Hypermedia controls contain the information needed to interact further on with a resource. In particular, they allow a client to navigate from one fragment (or a page thereof) to another one [28]. Triple Pattern Fragments implement this strategy through the generation of metadata and control information as RDF triples returned alongside the triples matching the triple pattern. This solution is effective but it imposes a restriction: the TPF interface can only return triples (or quads more generally). By contrast, SPARQL micro-services fully comply with SPARQL, in which the CONSTRUCT and DESCRIBE query forms return triples whereas the ASK and SELECT query forms return solutions formatted as variable bindings. The dereferenceable URIs dynamically assigned to Web API resources can be thought of as hypermedia controls. But how to provide additional metadata or control information within responses to SELECT or ASK query forms? We can think of two solutions to solve this issue. Firstly, extend the format of SPARQL query results with additional specific bindings. A second alternative could be to restrict SPARQL micro-services to support only CONSTRUCT and DESCRIBE query forms. This would spawn some sort of *Graph Pattern Fragment* interface, *i.e.* a generalized TPF interface that accepts regular graph patterns instead of only triple patterns, but still complies with the TPF metadata and hypermedia controls specification.

Let us finally mention that, in this article, we have focused specifically on consuming Web APIs data with SPARQL. In a broader picture, the micro-service principles could be applied to other types of APIs, so as to enable Semantic Web applications to reach out to other data sources. Furthermore, many APIs provide read-write access, empowering users to interact with contents, other users, etc. Hence, an interesting perspective would be to think of SPARQL micro-services as a way to support distributed SPARQL Update over Web APIs, thus eventually contributing to build an actual read-write Web of Data.

REFERENCES

[1] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. 2014. RDF 1.1 Turtle: Terse RDF Triple Language. *W3C Recommendation* (2014). https://www.w3.org/TR/2014/REC-turtle-20140225/

[2] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL Web-Querying Infrastructure: Ready for Action?. In *Proceedings of the 12th International Semantic Web Conference*. Springer, 277–293.

[3] Olivier Corby and Catherine Faron Faron-Zucker. 2010. The KGRAM Abstract Machine for Knowledge Graph Querying. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. IEEE, 338–341.

[4] Anastasia Dimou, Miel Vander Sande, Jason Slepicka, Pedro Szekely, Erik Mannens, Craig Knoblock, and Rik Van de Walle. 2014. Mapping Hierarchical Sources into RDF Using the RML Mapping Language. In *Proceedings of the International Conference on Semantic Computing (ICSC)*. IEEE, 151–158.

[5] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.

[6] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. 2013. SPARQL 1.1 Protocol. *W3C Recommendation* (2013). http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

[7] Fabien L. Gandon and Norman M. Sadeh. 2004. Semantic Web Technologies to Reconcile Privacy and Context Awareness. *Journal of Web Semantics* 1, 3 (2004), 241–260.

[8] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. *W3C Recommendation* (2013). http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[9] Tom Heath and Christian Bizer. 2011. *Linked Data: Evolving the Web into a Global Data Space* (1st ed.). Morgan & Claypool.

[10] Bernadette Hyland, Ghislain Atemezing, and Boris Villazón-Terrazas. 2014. Best Practices for Publishing Linked Data. *W3C Working Group Note* (2014). https://www.w3.org/TR/2014/NOTE-ld-bp-20140109/

[11] Matías Jünemann, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. 2016. Incorporating API Data into SPARQL Query Answers. In *Proceedings of the 15th International Semantic Web Conference (Posters and Demos)*.

[12] Johannes Koch, Carlos A Velasco, and Philip Ackermann. 2017. HTTP Vocabulary in RDF 1.0. *W3C Recommendation* (2017). https://www.w3.org/TR/2017/NOTE-HTTP-in-RDF10-20170202/

[13] Markus Lanthaler. 2013. Creating 3rd Generation Web APIs with Hydra. In *WWW'13 Companion Proceedings of the 22nd International Conference on World Wide Web*. ACM, 35–38. https://doi.org/10.1145/2487788.2487799

[14] Markus Lanthaler and Christian Gütl. 2013. Hydra: A Vocabulary for Hypermedia-Driven Web APIs. In *Proceedings of the 6th Workshop on Linked Data on the Web (LDOW2013)*. CEUR-WS.

[15] Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. 2017. A SPARQL extension for generating RDF from heterogeneous formats. In *Proceedings of the 14th Extended Semantic Web Conference (ESWC)*. Springer, 35–50.

[16] Pablo N. Mendes, Alexandre Passant, and Pavan Kapanipathi. 2010. Twarql: Tapping into the Wisdom of the Crowd. In *Proceedings of the 6th International Conference on Semantic Systems*. ACM.

[17] Franck Michel, Loïc Djimenou, Catherine Faron-Zucker, and Johan Montagnat. 2015. Translation of Relational and Non-Relational Databases into RDF with xR2RML. In *Proceeding of the 11th international conference on Web Information Systems and Technologies (WebIST)*. 443–454.

[18] Franck Michel, Catherine Faron-Zucker, and Johan Montagnat. 2016. A Generic Mapping-Based Query Translation from SPARQL to Various Target Database Query Languages. In *Proceeding of the 12th International Conference on Web Information Systems and Technologies (WebIST)*, Vol. 2. 147–158.

[19] Franck Michel, Olivier Gargominy, Sandrine Tercerie, and Catherine Faron-Zucker. 2017. A Model to Represent Nomenclatural and Taxonomic Information as Linked Data. Application to the French Taxonomic Register, TAXREF. In *Proceedings of the 2nd International Workshop on Semantics for Biodiversity (S4BioDiv) co-located with ISWC 2017*, Vol. 1933. CEUR.

[20] Franck Michel, Johan Montagnat, and Catherine Faron-Zucker. 2014. A survey of RDB to RDF translation approaches and tools. *Research report* ISRN I3S/RR 2013-04-FR (2014). http://hal.archives-ouvertes.fr/hal-00903568

[21] Benjamin Moreau, Patricia Serrano-Alvarado, Emmanuel Desmontils, and David Thoumas. 2017. Querying non-RDF Datasets using Triple Patterns. In *Proceedings of the 16th International Semantic Web Conference (Posters and Demos)*.

[22] Marie-Laure Mugnier, Marie-Christine Rousset, and Federico Ulliana. 2016. Ontology-Mediated Queries for NOSQL Databases. In *Proceedings of the 30th Conference on Artificial Intelligence (AAAI)*.

[23] Sam Newman. 2015. *Building Microservices*. O'Reilly Media.

[24] Eric Prud'hommeaux and Carlos Buil-Aranda. 2013. SPARQL 1.1 Federated Query. *W3C Recommendation* (2013). https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/

[25] Dimitrios-Emmanuel Spanos, Periklis Stavrou, and Nikolas Mitrou. 2012. Bringing Relational Databases into the Semantic Web: A Survey. *Semantic Web Journal* 3, 2 (2012), 169–209.

[26] Steve Speicher, John Arwe, and Ashok Malhotra. 2015. Linked Data Platform 1.0. *W3C Recommendation* (2015). https://www.w3.org/TR/2015/REC-ldp-20150226/

[27] Manu Sporny, Dave Longly, Gregg Kellog, Markus Lanthaler, and Niklas Lindström. 2014. JSON-LD 1.0. A JSON-based Serialization for Linked Data. *W3C Recommendation* (2014). https://www.w3.org/TR/2014/REC-json-ld-20140116/

[28] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. *Web Semantics: Science, Services and Agents on the World Wide Web* 37–38 (2016), 184–206.

[29] Gio Wiederhold. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer* 25, 3 (March 1992), 38–49.

[30] Olaf Zimmermann. 2016. Microservices Tenets: Agile Approach to Service Development and Deployment. *Computer Science-Research and Development* 32, 3 (2016), 301–310.