Tight integration of Web APIs with Semantic Web

Barry Nouwt [1]

[1] TNO - Netherlands Organization for Applied Scientific Research, The Hague, Netherlands
Barry.Nouwt@tno.nl

Abstract. This short paper describes an initial work to facilitate a better fusion of Web APIs and Linked Data. It explores how the Semantic Web (SW) community can benefit from data available through Web APIs. We believe a tight integration of Web APIs with SW technologies like SPARQL, OWL and reasoning has several advantages over loose integration. We demonstrate how a SPARQL query can be executed against a virtual triple store consisting of virtual triples from (possibly multiple) Web APIs. We discuss the challenges and limitations faced and describe further research on this topic.

Keywords: Semantic Web, Web API, Apache Jena.

1 Introduction

Semantic Web (SW) standards such as OWL [1], RDF [2] and SPARQL [3] promise to solve the harmonization and interoperability issues among the large amount of data sources available on the Web [10]. Using these standards when publishing and querying data is a good practice to enhance accessibility, reusability and reasoning across different sources, use cases and domains. However, to fully leverage the benefits of SW technologies, data sources on the Web should be in principle accessible via a SPARQL endpoint that allows to query RDF/OWL data, but in practice this is not the case. Although there are valid examples of SPARQL endpoints[1], the majority of data sources on the Web are exposed using different technologies, such as Web APIs or websites, which require an additional effort to be integrated with SW technologies.

In this paper we investigate how data that is only available through Web APIs can be integrated with data offered via SPARQL endpoints. Such integration would greatly increase the number of data sources to which SW technologies can be applied, since the number and usage of Web APIs is increasing [10]. For example, the most famous social media platforms provide comprehensive Web APIs that give users access to their social information, and several public transport organizations expose their travel information via Web APIs. Further, various Web APIs exist that provide information on topics like weather, maps or news.

Some related work has dealt with facilitating a better fusion of Web APIs and the SW. For example, the work in [4] aims to achieve a similar integration goal with SPARQL endpoints, although for relational databases rather than Web APIs. Their

[1] A list of SPARQL endpoints can be found here: https://www.w3.org/wiki/SparqlEndpoints

approach of translating SPARQL to SQL achieves integration at the level of SPARQL queries, while we propose to integrate on the level of the triple store. The efforts in [5], [6] and [9] propose the use of a RDF/OWL model to describe meta data about Web APIs with the option to map the Web API response to other RDF/OWL models. The solution in [7] adopts an 'ontology based access' approach to heterogeneous data sources on the Web, and the work in [8] tries to integrate semantically enriched Web APIs with SW technologies like SPARQL. These works address the lack of semantics in Web APIs, while our work assumes a Web API's semantics is mapped to a RDF/OWL model and investigates how actual integration without loss of functionality could be achieved.

The rest of the paper is structured as follows: Section 2 differentiates between tight and loose integration of Web APIs with SW technologies, and elaborates on the advantages of tight integration. Section 3 provides the main contribution of this paper by describing our initial attempt to realize a tight integration scenario. Finally, Section 4 discusses the limitations of our results, presents the challenges that still remain open and proposes some topics for further research.

2 Integration

We consider a typical SW solution as a solution that consists of the following four elements:

- An RDF/OWL model that describes the domain knowledge in the form of classes and properties about which a user wants interesting answers
- a SPARQL endpoint that allows users to formulate their SPARQL query (question) and receive the SPARQL result (answer)
- a triple store that is the database where answers to user questions are looked up, and
- the actual data that is a collection of facts in terms of the classes and properties described in the RDF/OWL model.

Whenever a user formulates a SPARQL query, the answers are sought in the actual data in the triple store. Optionally, additional facts that can be derived by the reasoner and included in the search for an answer. The integration of Web APIs with SW technologies like SPARQL, OWL and reasoning can be realized in a 'tight integration' or 'loose integration' manner.

Tight integration has several advantages over loose integration. Consider the example of a RDF/OWL model that defines a city that has a label, an average temperature and a population size, which is shown in **Fig. 1**.

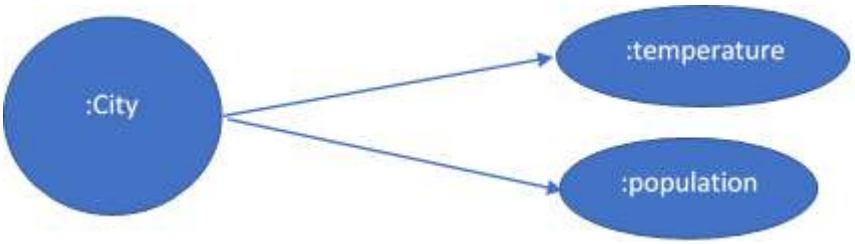

Fig. 1. Our simple example model in which cities have an average temperature and a population size.

Although the city's label and population sizes are available as triples in a triple store, the average temperature is only available through a public Web API. This Web API requires a city name and returns the average temperature for this city. **Fig. 2** shows two different scenarios for answering a user that requests both the population size and the average temperature for the city of Amsterdam.

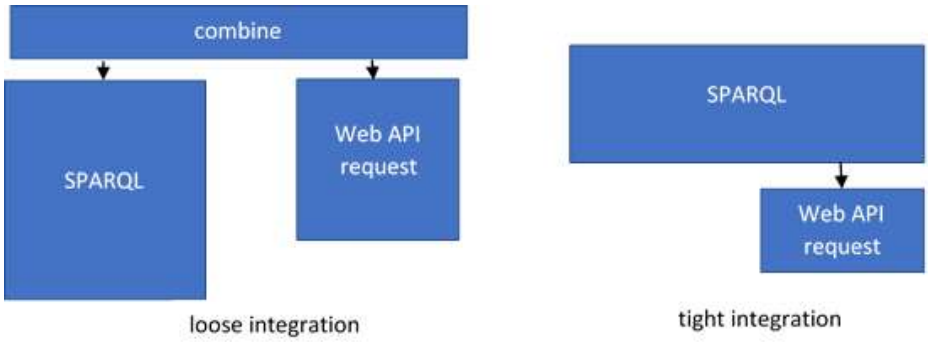

Fig. 2. The difference between what we call loose integration and tight integration of Web APIs with SW technologies such as SPARQL.

In the left part of **Fig. 2**, which we refer to as the 'loose integration' scenario, a SPARQL query retrieves the population size of Amsterdam from the triple store and this is combined, separately, with the result from the Web API request for the average temperature of the city of Amsterdam. In the right part of **Fig. 1**, which we refer to as the 'tight integration' scenario, the following SPARQL query retrieves both the population size of Amsterdam and its average temperature at once.

```
SELECT ?temp ?popsize
{
  :Amsterdam :hasAvgTemp ?temp .
  :Amsterdam :hasPopulationSize ?popsize .
}
```

In this case, the Web API request to retrieve the average temperature is triggered as part of the execution of the above SPARQL query.

The two scenarios differ in the way in which the Web API request is mapped to the model. With loose integration, the mapping between the Web API and the model is implemented in the separate component that combines the SPARQL results with the Web API results. The advantage is that combining these results on a case-by-case basis is fairly straightforward, but the disadvantage is that a user can no longer use the SPARQL interface and this limits the possible questions one can ask. Since asking random SPARQL queries is an important benefit that SW technology offers, this loose integration scenario is not desirable.

On the other hand, with tight integration the semantics of the Web API is mapped onto the domain model about cities, their population and average temperature, so that it is possible to automatically determine when and how to use the Web API. Although such integration requires a lot more effort to be implemented, the data behind the Web API is treated as if it were normal triples. These, so to say, virtual triples act as if the data was copied and transformed into triples and stored in the triple store, but instead the data stays in its original source behind the Web API, and is additionally available to the usual SW technologies such as SPARQL and reasoning.

3 Implementation

Our attempt to tightly integrate a Web API with a SPARQL endpoint for the considered example uses the concept of a virtual triple store, which has the function to expose the data behind a Web API as if they were triples. We have used Apache Jena Fuseki to implement the virtual triple store, since it is well-documented on how to extend its internal graph implementations, and for its integrated SPARQL engine and endpoint (Fuseki) that allow easy testing and demonstrating.

We have developed the VirtualRDFGraph Java library, which is a library that allows to expose multiple Web APIs as if they were triples stored in a Apache Jena graph. We have used the Service Provider Interface (SPI) loading facility included in Java to make this library automatically include any mappings that occur on the classpath. Each Web API requires a custom mapping that determines how the Web API result should be represented, as if they were triples written in terms of a certain RDF/OWL model. In the city example considered in this paper, the mapping should contain detailed information on how the result of the average temperature Web API is mapped to the city model and how this result can be converted into triples on-the-fly.

When the SPARQL query above is executed on a virtual triple store, the query gets interpreted by the Apache Jena's SPARQL engine (ARQ). The SPARQL engine creates a query execution plan by cutting the WHERE clause of the query into subject-predicate-object (SPO) patterns. In our case the query is cut as follows:

```
:Amsterdam :hasAvgTemp ?temp
```

and

```
:Amsterdam :hasPopulationSize ?popsize
```

Each of these SPO patterns are sent to our VirtualRDFGraph implementation. In turn, the VirtualRDFGraph sends the patterns to the registered Web API mapping to retrieve triples that match the pattern. The first triple `:Amsterdam :hasAvgTemp ?temp` triggers a Web API call based on the `:hasAvgTemp` property in the pattern, and uses `Amsterdam` as the parameter value. The Web API response contains the average temperature of Amsterdam, which is, for example, 15 degrees. The mapping converts the response into the following triple:

```
:Amsterdam :hasAvgTemp "15"^^xsd:integer
```

The second SPO-pattern `:Amsterdam :hasPopulationSize ?popsize` does not result in any triples for the Web API mapping, since the `:hasPopulationSize` property does not occur in its mapping. However, it gives results on the actual triples in the triple store, as follows:

```
:Amsterdam :hasPopulationSize "813562"^^xsd:integer
```

As a result, both triples above are returned to the SPARQL engine, which then combines them in the SPARQL response below and sends it back to the user. The response contains a single binding for the two selected variables `?temp` and `?popsize`.

```
{
  "head": {
    "vars": [ "temp" , "popsize" ]
  } ,
  "results": {
    "bindings": [
      {
       "temp": {"type": "literal" , "value": "15"},
       "popsize": {"type": "literal", "value": "813562"},
      }
    ]
  }
}
```

4 Discussion

We acknowledge that the example presented in this paper is rather simple, but it offers a good basis to reflect on the limitations that we have encountered and elaborate on the challenges that still remain open. This section concludes our short paper by discussing these limitations and challenges, and proposing some directions for future

research on the topic. An important limitation we have faced is that the VirtualRDFGraph limits the SPARQL queries that could be answered. An important feature of SPARQL queries is that they can be reversed. For our example, this means not asking the average temperature for the city of Amsterdam, but rather asking the cities with an average temperature of 15 degrees, which in SPARQL looks as follows:

```
SELECT ?city
{
  ?city :hasAvgTemp "15"^^xsd:integer .
}
```

In the `?city :hasAvgTemp "15"^^xsd:integer .` triple pattern, the variable has been moved from the object location of the SPO-pattern to its subject location, causing problems when calling the average temperature Web API.

Since the Web API from our example expects a particular city and returns the average temperature (and not the reverse[2]), we can no longer simply use the subject of the triple pattern and send it to the Web API. In this case, the subject is not a concrete value, but a variable, and thus we have nothing to send as a parameter to the Web API. Our current VirtualRDFGraph implementation returns a message saying that the above SPARQL query is not supported, but this neglects one of the major benefits that SW technology has to offer, i.e., asking random questions about a particular domain.

The only solution to this, without extending the Web API, is feeding the Web API one city at a time from a list of all cities, and collect and return those cities that have an average temperature of 15 degrees. The triple pattern itself no longer contains enough information and more context information is necessary, i.e. a list of all cities. The following query retrieves the necessary context information:

```
SELECT ?city
{
  ?city a :City
}
```

Unfortunately, our current implementation at the graph level does not allow access to this required context information. An alternative could be to implement the integration at the reasoner level, making the access to contextual information hopefully easier to realize. In this alternative, the average temperature Web API can be thought as an external (black-box) reasoner that can infer the average temperature based on a given city. Further research is required to test the viability and compare these two approaches.

Another challenge is the tight integration of Web APIs that do not return a single value (such as our average temperature Web API does) but returns a more complex

[2] The Web API could be extended to not only return the average temperature given a particular city, but also return a list of cities given an average temperature, but extending will not always be an option.

structure with multiple values. Instead of mapping a Web API to a single (data) property, the mapping might become considerably more complex. A solution could be to map the same Web API multiple times, each time focusing on a single value in the response and ignoring the rest.

Currently, the mapping between the Web API and the RDF/OWL model is captured in Java source code and this limits the usability of the VirtualRDFGraph. Initiatives like [5], [6] and [9] might provide ways to describe these mappings in a more user friendly way.

References

1. Hitzler P., Krötzsch M., Parsia B., Patel-Schneider P.F., Rudolph S. (2009). OWL 2 Web Ontology Language Primer, http://www.w3.org/TR/2009/REC-owl2-primer-20091027/
2. Schreiber G., Raimond Y., Manola F., Miller E., McBride B. (2014). RDF 1.1 Primer, http://www.w3.org/TR/2004/REC-rdf-primer-20040210/
3. W3C SPARQL Working Group (2013). SPARQL 1.1 Overview, https://www.w3.org/TR/rdf-sparql-query/
4. Bagosi T. et al. (2014) The Ontop Framework for Ontology Based Data Access. In: Zhao D., Du J., Wang H., Wang P., Ji D., Pan J. (eds) The Semantic Web and Web Science. CSWS 2014. Communications in Computer and Information Science, vol 480. Springer, Berlin, Heidelberg
5. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., ... & Sirin, E. (2004). OWL-S: Semantic markup for web services. W3C member submission, 22, 2007-04.
6. De Bruijn, J., Fensel, D., Kerrigan, M., Keller, U., Lausen, H., & Scicluna, J. (2008). The Web Service Modeling Ontology. Modeling Semantic Web Services: The Web Service Modeling Language, 23-28.
7. Decker, S., Erdmann, M., Fensel, D., & Studer, R. (1999). Ontobroker: Ontology based access to distributed and semi-structured information. In Database Semantics (pp. 351-369). Springer US.
8. Mouhoub, M. L., Grigori, D., & Manouvrier, M. (2015, May). LIDSEARCH: A SPARQL-Driven Framework for Searching Linked Data and Semantic Web Services. In European Semantic Web Conference (pp. 112-117). Springer International Publishing.
9. Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., & Domingue, J. (2010). iServe: a linked services publishing platform. In CEUR workshop proceedings (Vol. 596).
10. Tan, W., Fan, Y., Ghoneim, A., Hossain, M. A., & Dustdar, S. (2016). From the Service-Oriented Architecture to the Web API Economy. IEEE Internet Computing, 20(4), 64-68.