RDF-REST: A Unifying Framework for Web APIs and Linked Data

Pierre-Antoine Champin

RDF-REST: A Unifying Framework for Web APIs and Linked Data

Pierre-Antoine Champin

LIRIS**, Université Claude Bernard Lyon 1,
pierre-antoine.champin@liris.cnrs.fr,

Abstract. There has been a lot of efforts to bridge the gap between web APIs and linked data. The RDF-REST framework, that we developed in the process of building a RESTful service using linked data, is part of them. This paper presents the approach adopted in the RDF-REST framework, and show its benefits in the design and use of web APIs. One of these benefits is an easy integration of existing APIs with each other and with linked data.

1 Introduction

In the recent years, the web has moved from mostly human-oriented content to machine-readable content. This machine-readable web is not homogeneous, though. On the one hand, we have web APIs that allow machines to interact with each other, but are using more or less *ad-hoc* data structures. On the other hand, we have RDF based linked data [4,6], allowing to link and merge data from different sources, but mostly read-only.

There has been a lot of efforts to bridge the gap between these two kinds of approaches. Proposals have been made to interpret as linked data the data structures used by existing web APIs, like the Atom format [15] or the various dialects built atop JSON [14]. Other efforts aim at enhancing linked data with read-write capabilities [5,7,9,13,16]. In particular, the Linked Data Platform (LDP) working group[1] of the W3C aims at specifying a standard way to interact with linked data, following the principles of the REST architectural style [3].

This papers reports on a similar effort[2] to build a RESTful service based on linked data. While initially targeting a specific application, we ended up implementing a generic framework that can be used for both integrating existing web APIs and designing a various range of RESTful services. One of our goals is also to ease the implementation of linked-data-based services complying with the future LDP specification. We focus here on the general approach deployed in our framework, which we believe can be beneficial even beyond our own implementation.

** LIRIS UMR 5205 CNRS / INSA de Lyon / Université Claude Bernard Lyon 1 / Université Lumière Lyon 2 / École Centrale de Lyon

[1] LDP Working Group: http://www.w3.org/2012/ldp/

[2] http://champin.net/rdfrest

The rest of this paper is structured as follows. In Section 2, we describe the motivations and the rationale governing the approach adopted in the RDF-REST framework. In Section 3 we describe the RDF-REST framework itself. In Section 4, we demonstrate the benefits of RDF-REST compared to other frameworks. Finally, we point out some current limitations and discuss future evolutions of the framework in Section 5.

2 Motivation and rationale

As stated in the introduction, RDF-REST was first developed as the foundation for a specific application. Our motivations for using RDF and REST are, however, totally independent of the application domain, hence our further choice to make RDF-REST into a reusable framework.

2.1 On using REST

REST (REpresentational State Transfer) is the architectural style of the web [3]. It is expressed as a set of design constraints that web-based services should satisfy in order to preserve the good properties of the web (scalability, generality of interfaces, independent evolution of components, and transparency for intermediary components). It seems therefore only natural to comply with REST when designing a web-based service; however, many such services are only partially doing so (so called REST-like services), and sometimes not at all [10].

This is due to the fact that REST design is different from classical object-oriented (OO) design that most developers were taught, which makes it harder to achieve (for those developers). First, in REST, resources[3] are never handled directly, but only through *representations*. Second, REST imposes that all resources have a *uniform* interface. Behaviour differentiation is not achieved by exposing different interfaces as in OO design, but by exchanging different kinds of representations through the uniform interface.

In RDF-REST, we decided to keep the best of both worlds by splitting resource implementations in two parts: a *core* and a *wrapper*. Cores implement the behaviour of each resource, and they all expose the same interface (later called the RDF-REST interface) closely mapped to the HTTP verbs (see Section 3.1 for more details). Wrappers implement application-specific interfaces atop the RDF-REST interface.

In a way, this wrapper-core dichotomy is very similar to the client-server dichotomy of the web: it encourages loose coupling and separation of concerns between the two parts of the implementation. While it may first seem overly complex or costly to enforce this dichotomy *inside* an application, we will show in Section 4 that the benefits outweigh this apparent problem.

[3] In REST, processing units are called *resources*. While this notion bears some similarity to that of *object*, we stick to the term resource to emphasise the difference with classical OO design. We will only use the term object when referring to the implementation.

2.2 On using RDF

According to the REST principles, the resource's uniform interface is exclusively used to exchange *representations*. As stated above, the diversity of those representations is the key to differentiating resources and their behaviours. We therefore need a data model that is flexible enough to support those multiple kinds of representations.

RDF [6] is such a data model. Being specifically designed for the web, it uses URIs to name things, which is consistent with the REST and linked data principles. It is based on graphs, that allow to represent a wide range of data structures. Finally, it offers a mechanism for typing resources, where types are themselves identified by URIs.

Of course, not all web APIs use RDF; in fact, most of them do not, but rather use XML or JSON. We do not consider this as a problem, as RDF is flexible enough to represent the underlying data models of those languages. Indeed, there are general approaches for converting specific formats into RDF, applicable to any XML-based [1] or JSON-based formats [14], or to varieties of other formats [8,11]. The specific semantics of each format is preserved by using a dedicated set of URIs (vocabulary) in the translated graph. That way, RDF can unify different formats under a common data model, and still convey the differentiation between representations that is central to REST.

3 The RDF-REST framework

We can now present the RDF-REST framework, first by describing the RDF-REST interface that is the cornerstone of the framework; then by describing the framework architecture, which is structured around four factories. Finally, we briefly explain how developers are expected to use the framework.

3.1 The RDF-REST interface

The RDF-REST interface is the uniform interface of cores. It comprises five methods.

get_state : This method returns a read-only RDF graph representing the state of the resource. This graph will aim at staying up-to-date with respect to the resource. While this is straightforward for local implementations, this can be costly when the graph is retrieved from a remote source (checking before each access that the remote resource has not changed). Hence client implementation generally use caching mechanisms, and the graph *may* become stale temporarily.

force_state_refresh : This method forces the graph returned by *get_state* to update itself, bypassing any caching mechanism that may be used by the implementation. This method can be used whenever freshness is more important than performance.

get_edit_context : This method returns a modifiable RDF graph representing the current state of the resource, and having a *commit* method. The *commit* method is meant to be invoked after the graph has been modified, in order to apply those modifications to the resource.[4]

post_graph : This method expects an RDF graph as its input, and makes the resource process this graph in a way that depends on the resource itself, and on the content of the posted graph. A typical processing (but not the only one) is to create a new resource, of which the posted graph is a representation.

delete : This method deletes the resource from the service. After it has been successfully invoked, the corresponding core is supposed to be immediately discarded, as the resource it represented does not exist anymore.

Those methods clearly relate to the main verbs of the HTTP protocol [2]. Method *get_state* corresponds to GET (more precisely, each update of the returned graph corresponds to one GET). Method *force_state_refresh* corresponds to a GET with headers for cache bypassing. Method *get_edit_context* (more precisely method *commit* of the returned graph) corresponds to PUT. Methods *post_graph* and *delete* unsurprisingly correspond to POST and DELETE, respectively.

3.2 Framework architecture

The general architecture of the framework is described in Figure 1. It is composed of four factories. Each factory relies on a registry containing a number of classes, and where developers can add their own classes if needed.

The core factory will provide a core, implementing the RDF-REST interface described above, based on the URI of the corresponding resource. More precisely, this factory will recognise from the URI if the resource is managed by the service itself, or is a remote resource. In the former case, a local object implementing the behaviour of that resource will be returned. In the latter case, it will return a client object accessing the remote resource. By default, the corresponding registry only contains an HTTP client implementation.

The wrapper factory will provide a wrapper, based on the RDF type of the corresponding resource. The wrapper implements an application-specific interface atop the RDF-REST interface. Cores returned by the core factory will automatically use this factory and try to wrap themselves in an appropriate wrapper, if any is available for their RDF type. By default, the corresponding registry does not contain any class.

Whenever it is necessary to exchange representations of the resources with the outside, we need serialisers and parsers to transform RDF graphs to and

[4] Our implementation is actually slightly different, as it uses a more idiomatic pattern in Python (`with` statement). However, it is conceptually equivalent to the description made in this paper.

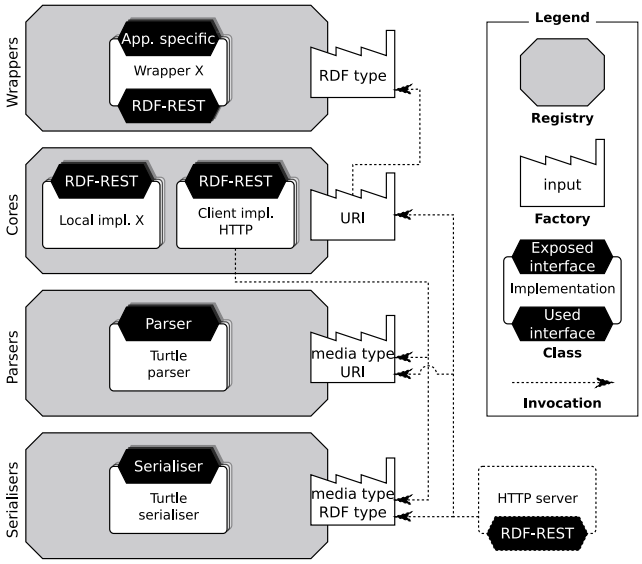

Fig. 1. Architecture of the RDF-REST framework

from various formats. This is especially important for client cores, that must communicate those representations with the remote resource.

The serialiser factory will provide a serialiser from RDF to a given format, based on the media type of that format, and the RDF type of the resource whose representation must be serialised. The parser factory will provide a parser from a given format to RDF, based on the media type of that format, and the URI from which the representation was retrieved (or to which it must be sent). By default, the serialiser and parser registries contain serialiser and parsers for the most common concrete syntaxes of RDF (Turtle, RDF/XML, N-Triple).

Finally, the RDF-REST framework contains a generic HTTP server. It uses the core factory to access the resources, and reflects HTTP queries to the corresponding method of the RDF-REST interface. It uses the serialiser and parser factories to transform between HTTP entities and RDF graphs.

3.3 Using the framework

Developing a service or application (either web-based or standard) with RDF-REST consists in providing a number of classes (wrappers, cores, serialisers or parsers) and registering them to the corresponding factory, associating them with the appropriate key(s).

For each kind of resource in the application, developers will typically provide a core implementation of the resource behaviour, and a wrapper implementation exposing an application-specific interface. They must also decide on an RDF vocabulary that the core implementation will use to describe the resource's state.

In particular, this vocabulary will include an RDF type to identify this kind of resource, that will be used to register the wrapper.

Whenever they want to support legacy formats (rather than the RDF syntaxes already supported by RDF-REST) they can also register new serialisers and parsers. Parsers are associated to a specific media type, and possibly to a URI pattern; for example, one may provide a JSON parser specialised in the JSON returned by `http://example.com/dummy/webapp`. Likewise, serialisers are associated to a specific media type, and possibly to an RDF type, as some resources may have a specialised way of serialising into a given format.

4 Benefits

In this section, we discuss the benefits of the architecture presented above, from different perspectives. This discussion is based on comparison with other frameworks, as well as on experience in using RDF-REST in developing an application[5] and a few other toy prototypes[6].

4.1 For developing RESTful services

RDF-REST proposes an approach quite different from the one usually proposed by other frameworks, such as JAX-RS,[7] Ruby on rails[8] or Django.[9] Those differences are illustrated in Figure 2.

Impedance mismatch. In most frameworks, the service is designed as any other OO application, with application objects exposing application-specific (AS) interfaces. Those specific objects are then annotated or wrapped in order to specify how they will be exposed through HTTP. As stated in Section 2, this often leads to sub-optimal REST design in the service. Furthermore, the client application (when designed in the OO paradigm) has to do the opposite work, converting HTTP-logic back into AS logic.

On the other hand, in RDF-REST, the uniform interface of cores makes them naturally exposed on HTTP, and encourages strict REST design from the start. Of course, implementing the resource's behaviour may benefit from the AS interface (which is represented in Figure 2 by the local implementations invoking themselves). But this is also possible in RDF-REST thanks to the wrappers, that can be used on the server side as they rely on the RDF-REST interface.

[5] kTBS: `http://liris.cnrs.fr/sbt-dev/ktbs`
[6] see for example `http://github.com/pchampin/semwiki`
[7] `http://jax-rs-spec.java.net/`
[8] `http://rubyonrails.org/`
[9] `http://djangoproject.com/`

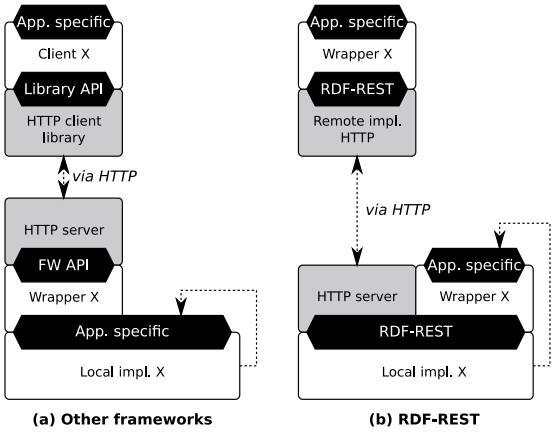

Fig. 2. Comparison of the RDF-REST approach with other frameworks
Gray classes are generic classes; white classes are application-specific classes.

Code reuse, modularity and consistency. As stated above, client code using the service may rely on a AS client library[10] exposing the service resources as classical objects, *i.e.* with the same interface as the one used on the server side. Those two different implementations of the same interface will usually contain very similar code, but can not easily be factorised, as they rely on very different basis (HTTP client library *vs.* local implementation).

In RDF-REST, however, the *same* wrapper (see Figure 2 (b)) can be used both on the client-side and the server-side, by virtue of the RDF-REST API, which abstracts away the local/remote nature of the underlying core. This reduces the amount of code to write and code redundancy.

One may be concerned by the fact that server-side implementation of the core should not treat alike input that comes from the outside and input that comes from itself through the wrapper. The former can not in general be trusted and must be checked before processing, while for the later, this checking is not necessary and might harm performances. To solve this problem, we added an optional *trust* parameter to methods *get_edit_context*, *post_graph* and *delete* of the RDF-REST interface. When this parameter is set to true, the implementation may bypass some checks. The wrapper can therefore set this parameter when it invokes those RDF-REST methods with input that it deems trustable. The generic HTTP server, on the other hand, will always set it to false, so that content from the outside is never trusted.

[10] "Application specific" should not be read here as "service specific", since RESTful services must not require specialized clients. "Application" here refers to a class of compatible services, embodied by a common media type that those services and their clients would use.

4.2 For integrating web APIs

RDF-REST is not limited to developing new services, but can also be useful as a unifying framework for using existing ones through web APIs. In that case, developers will not have to provide core implementations, as they will rely on the HTTP client core. However, they will still have to provide wrappers corresponding to the accessed resources, as well as serialisers and parsers to support the formats used by these APIs.

Consider an application integrating several photo sharing services, each one using a different format for representing its resources. Developers would have to provide a dedicated serialiser and parser for each service. However, as the resource types exposed by those services are essentially the same (*e.g.* Photo, User, PhotoCollection...), all of those serialisers and parses could use the same RDF vocabulary. It follows that only one wrapper per resource type would have to be provided, making those various services homogeneous from the point of view of the application.

Here we see how the unifying expressiveness of RDF, combined with the use of a common vocabulary, eases the integration of heterogeneous web APIs. Furthermore, a wealth of reusable RDF vocabularies for various application domains is already available on the web of linked data.

4.3 From a linked data perspective

Read-only linked data on the web can be seen as very simple RESTful services. As such, linked data can be integrated using RDF-REST as easily as any other service (actually more easily, as the serialisers and parsers are already provided by the framework). Obviously, this allows seamless integration of classical web APIs with linked data.

Conversely, as RDF-REST uses RDF as the underlying data model for all representations, and as it automatically supports common RDF syntaxes, any service developed with RDF-REST exposes its data as linked data (although strictly speaking, it only qualifies as linked data if it links to URIs outside the service).

Finally, as stated in the introduction, we are planning to make RDF-REST evolve consistently with the future Linked Data Platform (LDP) W3C recommendation [12]. Out goal is to make LPD compliance the default behaviour of RDF-REST based services (although full compliance will probably also depend on how developers implement their cores).

5 Limitations and future work

Below, we discuss the current limitations of the RDF-REST approach and implementation, as well as planned evolutions.

A first limitation of our approach is that it relies on the assumption that all resources can be represented in RDF. While this is in theory true, it is impractical for some resources, such as sounds, images or videos. Note that RDF

representation can still be used to reflect the meta-data of those resources, but the actual data is better represented in dedicated media type. We are considering extending the RDF-REST interface to account for this use case.

Another limitation of the approach is that it assumes that web-based services are developed from scratch. Indeed, if an existing application already exists, with existing objects exposing application-specific interfaces, and the RDF-REST interface needs to be implemented on top of those objects, we have an architecture similar to the one represented on Figure 2 (a), and we lose the benefits described in Section 4.1. However, as this is the architecture used by popular frameworks, the situation is not so bad, especially considering that RDF-REST still offers interoperability with linked data.

There is also a limit to our implementation that we wish to address: it is implemented in Python, which makes it less prone to adoption than if it was in a more widespread language such as Java. We do believe that the approach is valuable in itself, and so that other implementations in different languages are not only possible, but also desirable.

Another language-related plan for the future is to use JavaScript translators (such as GWT[11] for Java of Pyjs[12] for Python) in order to allow the client part of Figure 2 (b) to be dynamically embedded in a web browser (what Fielding calls "code on demand" [3]). This would comply even more closely to the REST principles, making existing clients (web browsers) dynamically able to cope with new media types exposed by RDF-REST-based services. Note that the Drumkit framework[13] has a similar approach, but mandates the use of JavaScript on the server side.

6 Conclusion

In this paper, we have presented RDF-REST, a framework based on REST and linked data principles. It eases the development of RESTful services that consume and produce linked data, but also makes it possible to integrate existing heterogeneous web APIs with each other, and with linked data sources.

We have implemented this framework and used it to develop RESTful services, but we purposefully described it at a rather abstract level. Indeed, our experience with the framework showed that the *approach* it proposes, regardless of a particular implementation or programming language, can lead to better RESTful design of web APIs, and to a convergence of the web of services with the web of linked data.

Acknowledgements This work is supported by the French National Research Agency (ANR) through the KolFlow project (code: ANR-10-CONTINT-025), part of the CONTINT research program.

[11] `https://developers.google.com/web-toolkit/`
[12] `http://pyjs.org/`
[13] `http://drumkitjs.com/`

References

1. D. Connolly. Gleaning resource descriptions from dialects of languages (GRDDL). W3C recommendation, W3C, Sept. 2007. <http://www.w3.org/TR/grddl/>.
2. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, IETF, June 1999.
3. R. T. Fielding. Architectural styles and the design of network-based software architectures. 2000.
4. T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space (1st edition)*, volume 1 of *Synthesis Lectures on the Semantic Web: Theory and Technology*. Morgan & Claypool, 2011.
5. K. Kjernsmo. The necessity of hypermedia RDF and an approach to achieve it. In *Proceedings of the First Linked APIs workshop at the Ninth Extended Semantic Web Conference*, 2012.
6. G. Klyne and J. J. Caroll. Resource description framework (RDF): concepts and abstract syntax. W3C recommendation, W3C, Feb. 2004. <http://www.w3.org/TR/rdf-concepts/>.
7. R. Krummenacher, B. Norton, and A. Marte. Towards linked open services and processes. In *Future Internet-FIS 2010*, page 68–77. Springer, 2010.
8. H. Mühleisen and C. Bizer. Web data commons – extracting structured data from two large web corpora. In *5th Linked Data on the Web workshop (LDOW)*, Lyon, France, 2012.
9. C. Ogbuji. SPARQL 1.1 graph store HTTP protocol. W3C recommendation, W3C, Apr. 2013. <http://www.w3.org/TR/sparql11-http-rdf-update/>.
10. D. Renzel, P. Schlebusch, and R. Klamma. Today's top "RESTful" services and why they are not RESTful. In *Web Information Systems Engineering-WISE 2012*, page 354–367. Springer, 2012.
11. M. V. Sande, L. De Vocht, D. Van Deursen, E. Mannens, and R. Van de Walle. Lightweight transformation of tabular open data to RDF. In *I-SEMANTICS 2012 Posters & Demonstrations Track*, Graz, Austria, Sept. 2012.
12. S. Speicher and J. Arwe. Linked data platform 1.0. W3C working draft, W3C, Mar. 2013. <http://www.w3.org/TR/ldp/> Accessed on 2013-04-15.
13. S. Speiser and A. Harth. Taking the LIDS off data silos. In *Proceedings of the 6th International Conference on Semantic Systems*, I-SEMANTICS '10, page 44:1–44:4, New York, NY, USA, 2010. ACM.
14. M. Sporny, G. Kellogg, and M. Lanthaler. JSON-LD 1.0 – a JSON-based serialization for linked data. W3C last call working draft, W3C, Apr. 2013. <http://www.w3.org/TR/json-ld-syntax/> Accessed on 2013-04-15.
15. H. Story and D. Ayers. AtomOwl vocabulary specification. http://bblfish.net/work/atom-owl/2006-06-06/AtomOwl.html, June 2006. Accessed on 2013-03-11.
16. R. Verborgh, T. Steiner, R. Van de Walle, and J. Gabarró Vallés. The missing links–How the description format restdesc applies the linked data vision to connect hypermedia apis. In *Proceedings of the First Linked APIs workshop at the Ninth Extended Semantic Web Conference*, 2012.