# The Web as an Infrastructure for Knowledge Management: Lessons Learnt

Aurélien Bénel, L'Hédi Zaher

# The Web as an infrastructure for knowledge management: Lessons learnt

Aurélien Bénel[1] and L'Hédi Zaher[2]

[1] ICD/Tech-CICO, Troyes University of Technology (France)
aurelien.benel@utt.fr
[2] IM Développement (Tunisia)
hedi.zaher@imdev.tn

**Abstract.** Research works, whether they aim at building a 'Semantic Web' or a 'Social Semantic Web', consider as a prerequisite that the ideal architecture for managing knowledge would be the Web. Indeed, one can only admire how the CERN internal hypertext scaled out to a world wide level never seen before for this kind of applications. However, current knowledge structures and related algorithms cause new kind of architectural issues. About these issues faced by both communities, we would like to bring out three lessons learnt, three steps in setting up a scalable infrastructure. We will focus on a typical case of knowledge management but with a higher than usual volume of data. Starting with SPARQL, a commonly used Semantic Web technology, we will see the benefits of the REST architecture and the MapReduce design pattern.

## 1 Introduction

This paper deals with Web services design for knowledge management. Although HTTP scaled out remarkably for the World Wide Web, scaling out knowledge management using web services is still an open issue.

Weirdly enough, in the 'Semantic Web' program (Berners-Lee *et al.*, 2001), the World Wide Web Consortium focused more on formats and languages than on the use of its own protocol. Meanwhile, through trials and errors, we developed an experience in the design of an infrastructure for a 'Social Semantic Web' (Zacklad *et al.*, 2003, Bénel *et al.*, 2010). Even if this approach has been developed as an opposite to the 'Semantic Web', we think that the lessons learnt in setting up a scalable infrastructure could benefit both approaches.

We will focus on a typical case of knowledge management but with a higher than usual volume of data. Then, we will bring out three steps in designing an infrastructure. Starting with SPARQL, a commonly used Semantic Web technology, we will see the benefits from the REST architecture and the *MapReduce* design pattern.

## 2 Requirements

We will illustrate our experience feedback with *i-Semantec*, a project related to knowledge capitalization, management and reuse in large industrial companies.

In these firms, data and documents from various stakeholders and relating to the lifecycle of the product are already managed by integrated systems named 'PLM' (for 'Product Lifecycle Management'). But there are mainly two problems with these integrated systems. First, because the integration is based on the formalization of the main business processes, it often ignores the specificity of each profession. And then each profession tends to implement its own databases and documents, which escape capitalization. Second, in a time of market instability, the information system should be more adaptable to the continuous change of processes and partners networks needed to meet customers' requirements.

Since this project only serves as an example we will not write more about functional and technical requirements. We will only focus on the feature list we had to implement:

- browsing a technical data warehouse which models may vary depending on the project;
- enhancing these data with freely defined attributes.

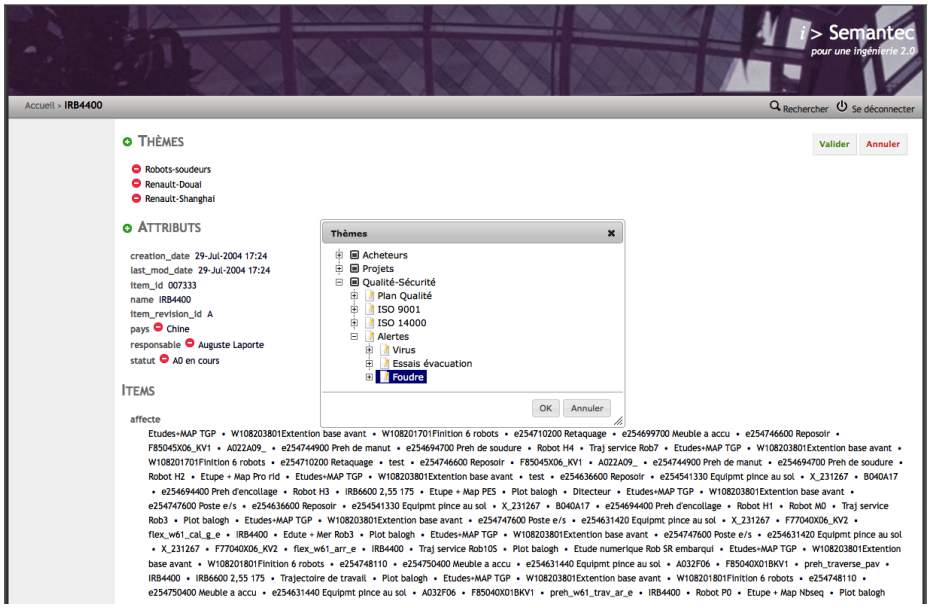


**Fig. 1.** Enriching the technical data of a robot in *i-Semantec* (*Agorae* screenshot)

Items had to be browsable:

- by class of items,
- by (used) attribute and value,
- from another item through a composition relationship, a sequence in a production line, etc.

## 3 First step: Semantic Web technologies

In the *i-Semantec* project, our partners were specialists in product data management. They extracted data from two industrial projects as triples and stored them in a RDF data warehouse system (Sriti *et al.*, 2007). This system implemented the HTTP biding for SPARQL (Clark *et al.*, 2008). Each project had been provided with its data model as a schema in RDFS. We then tried to build the features described earlier on this typical Semantic Web infrastructure. As we will see, we experienced serious and blocking issues.

### 3.1 Browsing items

The first issue encountered in implementing the Web service for items browsing was about performance. As shown in Tab. 1, response times which were quite acceptable for 1k triples, scaled very badly with 1 million triples. Moreover, the only mechanism for getting better response times on successive identical queries was the in-memory cache of the database, which is very dependent on free memory.

| | approx. 1k items | approx. 1G items |
|---|---|---|
| Item details | 0.5 | 1 |
| Listing items types | 0.5 | 0.5 |
| Listing items for a type | 0.2 | 40 |
| Listing used attributes | 0.4 | 30 |
| Listing values for an attribute | 0.1 | 3 |
| Listing items for an attribute value | | 5 |

**Table 1.** Comparison of response time to SPARQL queries (seconds) on databases containing respectively 834 and 931,338 RDF triples. Tests are done with RAP 0.9.4 by Ch. Bizer and MySQL on Linux with a Xen virtual machine equivalent to an AMD Athlon 64 X2 with 2 Gb memory.

### 3.2 Enriching items

Triples model, aka Entity-attribute-value model (EAV), is a well known model which can combine multiple schemas and accept user-defined attributes.

However, in its RDF/XML implementation, attributes are XML elements. As XML elements, they should be defined at design time in a schema. This makes it more difficult to let users define attributes on the fly, contrary to JSON for instance, where an attribute (key) is just a string (Crockford, 2006).

Moreover, SPARQL Update, the extension to the SPARQL query language that provides the ability to add, update, and delete RDF triples was not implemented at the time of the project. The five-year lag between the normalization

of queries (2008) and updates (Schenk *et al.*, 2010) (2013) is probably indicative of the difference in priorities between the Semantic Web and the Social Semantic Web.

Therefore, we had to use our own knowledge management Web service for user-defined enrichments, integrated with a read-only connector used to query the RDF data warehouse through our own protocol.

Weirdly enough, in an opportunistic and serendipitous manner, the fact that it was a read-only access and a non-updatable data warehouse solves the first issue: payloads computed from the RDF data could be cached once and for all on disk.

In other words, to meet the requirements we had to 'hack' the Semantic Web technologies, which appeared more a burden than a help for this project.

## 4 Second step: REST architecture

From an architectural perspective, one notable difference between our protocol (Zhou *et al.*, 2006) and SPARQL was that our protocol was 'RESTful'.

REST is an architectural style for 'distributed hypermedia systems', introduced by one of the author of HTTP in his thesis (Fielding, 2000). It aims at generalizing the scalable design of the original Web to the complex Web applications and Web services of nowadays. As its complete name suggests ('representational state transfer'), the main idea of REST is that in a protocol like HTTP, a payload should always be a state of a resource representation. This implies that neither resource representation nor resource identifiers should refer to actions. Instead, one should use the corresponding methods defined in the protocol (e.g. GET, POST, PUT, DELETE, HEAD, etc. in HTTP). To qualify these methods, the HTTP specification introduces two important notions: 'safe' and 'idempotent'.

> "In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered 'safe'." (Fielding *et al.*, 1999).

Being *safe*, requests using GET (accordingly to the specifications) will be cacheable, preemptively loadable, or even usable in a repeatable history. However to be cached by a server, a client, a proxy or a reverse proxy, changing POST with GET (like SPARQL does) will not be enough. The server has to implement a cache invalidation mechanism based on an update timestamp or on a content hash. Moreover, a resource representation should be cached only if there are chances that it will be retrieved again. Therefore a URL should map to an identifiable object rather than to a common query. This also makes integration easier, since using a different data management system would not require to emulate a complete query language but just one query.

> "Methods can also have the property of 'idempotence' in that (aside from error or expiration issues) the side-effects of $N > 0$ identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property." (Fielding *et al.*, 1999).

Being *idempotent*, requests using PUT or DELETE (accordingly to the specifications) will be resilient to client retries on timeouts. Contrary to the updates done with POST by SPARUL (Schenk *et al.*, 2010), an update with PUT will never cause multiple creations due, for example, to an excessive load of the server.

At this stage, we have what most bloggers call a REST service (Tilkov, 2007):

- one updatable resource by object (with computed links to other objects),
- one 'super-resource' by class (in order to list instances).

However, we will see in the next section that this design is not ideal for cache management and for complex actions performance, and we will introduce a more efficient RESTful design.

## 5 Third step: *MapReduce* design pattern

In the previous section, we described an approach in which every object is mapped to an HTTP resource. The main issue of this naive way to follow the REST architecture is the difficulty to implement a server-side cache. Indeed, if the resource representation includes data from other resources (typically for reverse links), it becomes quite complex to determine from the history if a resource representation has changed without computing it again.

A second issue in this approach is that the more objects you need to load, the more requests you have to send. For example, loading the data of a prolific user of our software (to run a data visualization algorithm (Zhou & Bénel, 2008)) required 25 000 requests with this architecture. Because of each request latency, the overall time needed to load these data is 31 minutes! To reduce latency to a constant time, the number of requests needed for a complex operation should be constant. Therefore, on large datasets, a bulk of objects should be mapped to a single resource. Then a new problem would arise: the gain of caching a resource representation will be very low since it would change much more frequently. Instead, one should cache partial results needed for generating this representation. This can be fairly complex, but luckily, the *MapReduce* design pattern addresses this problem.

*MapReduce* (Dean & Ghenawat, 2004) is a design pattern for processing large data sets. In a *MapReduce* framework, developers only have to implement (see Fig. 2):

- a *map* function that "generates [for each chunk of data] a set of intermediate key/value pairs",
- and a *reduce* function that "merges all intermediate values associated with the same intermediate key".

Then the framework handles the cache of partial results and the scaling of the algorithm over different processes and computers.

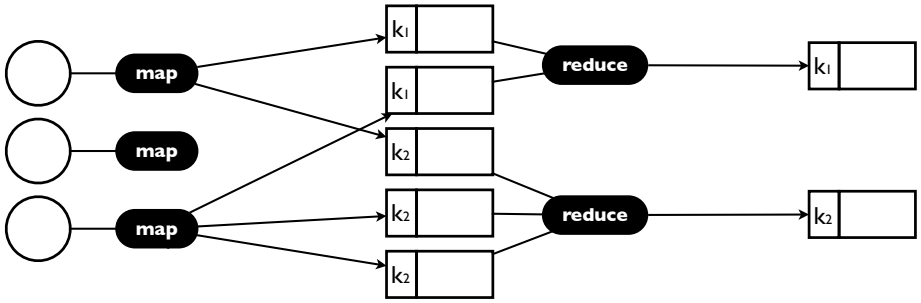


**Fig. 2.** *MapReduce*: data flow

One should note that processing data with *MapReduce* is drastically different from doing it with a relational database. In a relational approach, the index depends on the data model, and is designed to optimize any queries that could be defined on it. In a *MapReduce* approach, the index depends on the map functions. Algorithms must be adapted to use intermediate values sorting wisely (similarly to what was done with IBM/BULL card sorters). Compared with the relational approach, *MapReduce* is like computing the results of a query for every possible parameter found in the data. What could be seen as a burden is usually an optimization, since only partial results affected by an update are computed again.

As observed in a database system like *CouchDB* (Anderson *et al.*, 2010), using *MapReduce* has three important impacts on a REST service interface.

First, different kinds of resources are needed: the objects that are updatable and the *views* that are selections on the results of applying *map* and (optionally) *reduce* functions on these objects. In other words, contrary to what we saw in the previous section, computed values are not displayed in updatable resources anymore but in different resources that are read-only.

Second, owing to the implementation of the *MapReduce* framework (partial results cache and index), getting a broader *view* is incredibly faster than getting the same data with a bunch of narrower ones. Therefore, the granularity of resources depends on their type: *views* tend to be far more coarse-grained than objects (whose granularity corresponds to what is usually updated at the same time).

Third, because *MapReduce* aims at distributing computing, creating an object identifier should be done in a distributable way. As many peer-to-peer software, CouchDB uses 'universal unique identifiers' (UUID). But because a UUID (Leach *et al.*, 2005) corresponds to a 'uniform resource name' (URN) rather than to a 'uniform resource location' (URL), we need to reinterpret the browsability principle of REST services (Fielding, 2008) in the light of the original chapter

thesis about REST (Fielding, 2000) which stated that the use of URNs instead of URLs could "improve the longevity of resources references". Untying a resource reference from a location has also interesting effects on services integration, since clients can aggregate the descriptions of a resource that are scattered over different services that do not necessarily 'know' each other. This can be particularly handy to meet the functional requirements given earlier, since data coming from the existing database system can be provided through a read-only *adapter* service, while every community (or person) can model and store its (his) 'viewpoint' on its (his) own service.

## 6 Conclusion

The three steps we experienced in designing a scalable Web service infrastructure for knowledge management could be summed up by focusing on what an URL maps to:

1. a query,
2. an object with computed attributes,
3. an object or a view.

As we saw in this paper, this progression has drastic effects on caching, distribution and integration.

If our proposition is still novel, it is probably because very few in the 'Semantic Web' community have had interest in enterprise technologies like REST (Ogbuji, 2010) or *MapReduce* (Oren *et al.*, 2008). As we did with use models, we hope that performance issues will be considered by the community in order to meet real user needs.

## References


Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide. O'Reilly (2010)

Bénel, A., Zhou, C., Cahier, J.-P.: Beyond Web 2.0... And Beyond the Semantic Web. D. Randall & P. Salembier. From CSCW to Web 2.0: European Developments in Collaborative Design. Computer Supported Cooperative Work. Springer (2010) 155–171

Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American, May 17 (2001)

Bizer, Ch., Heath, T., Berners-Lee, T.: Linked data-the story so far. International Journal on Semantic Web and Information Systems (IJSWIS) **5** 3 (2009) 1–22

Bizer, Ch., Schultz A.: The Berlin SPARQL Benchmark. International Journal On Semantic Web and Information Systems (IJSWIS) **5** 2 (2009) 1–24

Clark, K.G., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF. Recommendation. W3C (2008)

Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627. IETF (2006)

Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Proceedings of the Sixth Symposium on Operating System Design and Implementation (2004)

Dieng-Kuntz, R., Matta, N.: Knowledge Management and Organizational Memories. Proceedings of European Conference on Artificial Intelligence (2004)

Fielding, R.T.: REST APIs must be hypertext-driven. Blog post, 20 oct. (2008)

Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California. (2000)

Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. IETF (1999)

Leach, P., Mealling, M., Salz, R.: A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122. IETF (2005)

Ogbuji, C.: SPARQL 1.1 Uniform HTTP Protocol for Managing RDF Graphs. Working draft. W3C (2010)

Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice. com: A document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies **3** 1. Inderscience (2008) 37–52

Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. Recommendation. W3C (2008)

Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update. Recommendation. W3C (2013)

Sriti, M.-F., Eynard, B., Boutinaud, Ph., Matta, N., Zacklad, M.: Towards a semantic-based platform to improve knowledge management in collaborative product development. Proceedings of the thirteenth International Product Development Management Conference (2007)

Tilkov, S.: A brief introduction to REST. InfoQueue, 10 dec. (2007)

Zacklad, M., Cahier J.-P., Pétard, X.: Du web cognitivement sémantique au web socio sémantique : Exigences représentationnelles de la coopération. Web sémantique et Sciences humaines et sociales (2003)

Zaher, L'H., Cahier, J.P., Zacklad, M.: The Agoræ/Hypertopic approach. Workshop on Indexing and Knowledge in Human Sciences (IKHS). M. Harzallah, J. Charlet, N. Aussenac-Gilles. Actes de la semaine de la connaissance **3** (2006) 66–70

Zhou, C., Bénel, A.: From the crowd to communities: New interfaces for social tagging. Proceedings of the Eighth International Conference on the Design of Cooperative Systems (2008) 242–250

Zhou, C., Bénel, A., Lejeune, C.: Towards a standard protocol for community-driven organizations of knowledge. Proceedings of the thirteenth international conference on Concurrent Engineering. Frontiers in Artificial Intelligence and Appl. **143**. IOS Press (2006) 438–449