# A JavaScript RDF store and application library for linked data client applications

Antonio Garrote Hernández
Universidad de Salamanca
antoniogarrote@usal.es

María N. Moreno García
Universidad de Salamanca
mmg@usal.es

## ABSTRACT

In this paper we present a pure JavaScript implementation of an RDF store supporting the SPARQL query language that can be executed in modern browsers as well as in server side JavaScript platforms. We also present a declarative JavaScript library, built on top of the store, that makes it possible to build rich web clients combining the power of structured linked data, lightweight RDF notations like JSON-LD and the SPARQL query language with the dynamic nature of the DOM event model to provide a simple development framework appealing to general web developers with little prior knowledge of the semantic web stack of technologies.

## Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Information Storage

; H.4 [**Information System Applications**]: Communication Applications

## General Terms

Design

## 1. INTRODUCTION

RDFStore-js [1] is a JavaScript implementation of an RDF quad store with support for the SPARQL query and update language that can be executed in the browser and in the Node.js [2] server side JavaScript platform.

RDFStore-js supports all of SPARQL 1.0, most of SPARQL 1.1/update and a significant portion of SPARQL 1.1 query. The Node.js version of the store also implements the SPARQL protocol for RDF so it can be used as a stand-alone SPARQL end-point, accessible through HTTP requests.

[1]https://github.com/antoniogarrote/rdfstore-js
[2]http://nodejs.org/

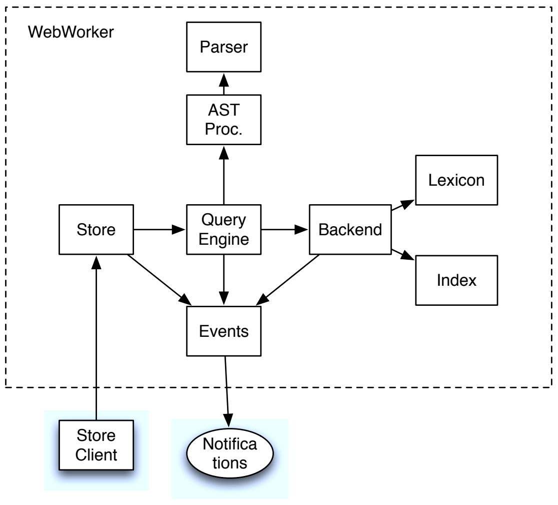

**Figure 1: RDFStore-JS architectural components**

The main goal of the library is to provide the foundation for the data layer of complex JavaScript client applications. It can also be used as a general purpose RDF library for server side applications using RDF as the exchange format in APIs despite of the final persistence layer being used.

The store supports loading of remote RDF graphs in SPARQL queries using browser's native networking features or an additional input/output interface. The store includes parsers for JSON-LD and N3/Turtle serializations of RDF. Support for additional RDF serializations can be added as extensions.

## 2. DESIGN

The store takes advantage of advanced browser features, providing persistent storage and multi-threaded execution where available in the browser. These features are also available in the node.js using server side implementations of the browser APIs. The figure 1 shows the main components of the store architecture.

### 2.1 In-Memory Storage and Indexing

RDFStore-js storage model is based on a single conceptual table where RDF quads composed of subject, predicate and object plus an additional graph URI are stored in a lexicon component. Additionally, a set of b-tree indexes are generated to improve the performance of data retrieval queries.

These indexes are the minimal number of indexes needed to cover different query patterns in SPARQL basic graph queries [2] and are used to speed range scans over the lexicon.

## 2.2 Persistent Storage

Persistence has been added to the lexicon and indexes in different ways in browser and node.js implementations of the store because of the different storage layers of both platforms.

Browser persistence is based on the use of the Web Storage API [4]. This API defines a `localStorage` object wrapping a persistent key value map capable of storing `String` JavaScript objects. Storage capacity is also limited to a few mega bytes of data, 15 MB being the current default capacity in most implementations. The lexicon and indices of the store are saved in this persistence layer using a custom serialization scheme for JSON objects encoding quad components.

Persistent storage is complemented with the use of a write through cache component. Due to the limitation in the web local storage size, the whole storage can be kept at the same time in main memory.

Persistence in node.js has been achieved using MongoDB [3] as the storage engine for the RDF graph. RDF quads are encoded as MongoDB documents using a different serialization, consisting of a normalized representation of the term value with an additional term type tag. This representation makes innecessary the use of a lexicon and makes it possible to leverage the full power of MongoDB native indices in the retrieval of quad patterns.

## 2.3 SPARQL Queries Parsing and Processing

The process of answering SPARQL queries over a collection of RDF graphs saved in the store starts with the parsing of the SPARQL query. This process is performed in two steps. First, the textual SPARQL query is parsed into a complex JSON object representing the abstract syntax tree for the query. Second, the abstract syntax tree is transformed into a different JSON object containing the representation of an equivalent SPARQL algebra expression according to W3C's SPARQL semantics.

The parsing step is performed using a parsing expression grammar (PEG)[4] [1] that is capable of parsing SPARQL queries, SPARQL 1.1 and SPARQL Update queries as well as turtle documents.

## 2.4 Threaded Execution

JavaScript applications running in the browser or in node.js are restricted, with the exception of the Web Workers API [3], to a single execution thread. To avoid blocking the application due to IO bounded operations, JavaScript platforms offer evented asynchronous interfaces to most IO blocking operations as well as a synchronous blocking one.

[3]http://www.mongodb.org/
[4]http://pegjs.majda.cz/

One exception to the single threaded environment of JavaScript applications is the Web Workers API [3]. This API makes it possible to execute JavaScript code in a different sand-boxed thread. Both threads cannot share any state and must communicate exclusively exchanging string messages.

The store takes advantage of this API where available, to execute SPARQL queries concurrently in a different thread. The upper layer of the store is the `Store` module. This module defines the final interface that can be used by client code interacting with the store as a set of asynchronous functions. If the Web Workers API is detected, the store module is loaded on a web worker and a different module named `RDFStoreClient` is loaded on the main thread. Both modules implement the same interface, but invocation of the methods of the `RDFStoreClient` module results in messages being sent to the previously created web worker. When the final results of the query are ready, they are returned to the client in an additional message and will be finally delivered to the client code in an asynchronous callback invocation. Encoding and decoding of results and request arguments between threads is accomplished using standard JSON to `String` serialization.

If Web Workers are not available in the browser, the `Store` module will be loaded in the main thread instead, without any required modification of the client code.

## 2.5 Benchmarking

Performance evaluation of the store has been accomplished using the LUBM [5] benchmark for a single university on different web browsers being executed on an average laptop computer. Test cases have been generated using the LUBM data generator and then transformed into JSON-LD before running the tests. The final amount of data loaded are 100545 triples stored in a single graph. The table 1 shows the results obtained in milliseconds. Since the store does not support inference, some queries in the benchmark has been re-written using `UNION` clauses that test for explicit patterns. An additional query that simply returns all the triples has also been added. The text of the queries [6] as well as the code to run the tests are included in the source code distribution of the library.

## 2.6 Events API

JavaScript client applications need to react to user events changing the state of the application. As a result, different components of the application need to be updated as the result of these state changes.

To make it easier to build this kind of event dispatching logic, an events API has been added to the store. As the rest of the public store interface, the events API can be used at two different levels, the SPARQL query level or the RDF node level.

At the SPARQL level, client code can subscribe to SPARQL queries using the `startObservingQuery`/`stopObservingQuery` functions. Each time a modification of the store RDF graph

[5]http://swat.cse.lehigh.edu/projects/lubm/
[6]http://antoniogarrote.github.com/rdfstore-js/queries.txt

| Query | Chrome 16 | Safari 5 | Firefox 11 | MSIE 9 |
|---|---|---|---|---|
| 0 | 0.552 | 1.176 | 0.834 | 0.771 |
| 1 | 0.005 | 0.033 | 0.043 | 0.016 |
| 2 | 0.018 | 0.149 | 0.046 | 0.111 |
| 3 | 0.005 | 0.022 | 0.026 | 0.023 |
| 4 | 0.155 | 0.502 | 0.311 | 0.603 |
| 5 | 0.043 | 0.091 | 0.109 | 0.131 |
| 6 | 0.023 | 0.039 | 0.045 | 0.057 |
| 7 | 0.324 | 0.573 | 0.73 | 1.678 |
| 8 | 0.828 | 1.581 | 1.789 | 2.548 |
| 10 | 0.008 | 0.022 | 0.024 | 0.027 |
| 11 | 0.001 | 0.003 | 0.006 | 0.003 |
| 12 | 0.003 | 0.007 | 0.011 | 0.006 |
| 13 | 0.042 | 0.103 | 0.098 | 0.119 |
| 14 | 0.009 | 0.028 | 0.024 | 0.035 |

**Table 1: LUBM-1 results for different browsers**

changes the results of the queries tracked by the events API, the new set of results will be returned to the subscribing callback functions.

At the RDF node level, the functions `startObservingNode/stopObservingNode` can be invoked to receive notifications whenever the state of an RDF graph node changes. The new value of the RDF node will be returned to the callback functions as graph objects according to the RDF Interfaces specification.

### 2.7 JavaScript - RDF Application Library

SemanticKO [7] is a complementary application development library implementing mechanisms to establish declarative bidirectional bindings between RDF graphs stored using RDFStore-JS and the DOM tree of a HTML document. The library is a modified and extended version of KnockoutJS [8], a JavaScript library implementing the Model-View-ViewModel architectural pattern [5].

The following snippet shows how RDF data for a status update stored in the RDF store together with its author and described using the SIOC vocabulary can be bound to elements in a HTML document:

```
<div class='MicroblogPost'
     about='<http://test.com/posts/twitter/342>'>
  <div class='header'>
     <span class='creator'
          rel='[sioc:has_creator]'>
       <img class='avatar'
            data-bind='attr: {src: [sioc:avatar]}'>
       </img>
       <span data-bind='text: [sioc:name]'></span>
     </span>
  </div>
  <div class='body'>
       data-bind='text: [sioc:content]'>
  </div>
</div>
```

[7]https://github.com/antoniogarrote/semantic-ko
[8]http://knockoutjs.com/

Bound JavaScript node objects subscribe to changes in the RDF node kept in the store using RDFStore-JS events API. Every time the values in the RDF node graph change, the `Node` object is notified and it changes the values of the observable properties accordingly.The change in these observable properties triggers the re-evaluation of dependent DOM `Node` objects bound to related RDF nodes or triggers changes in the properties declared in `data-bind` attributes. As a result an updated DOM tree is computed reflecting the changes in the RDF graph. In the opposite direction, interaction between the user and HTML DOM nodes will automatically change the value of bound RDF nodes.

## 3. CONCLUSION

RDFStore-JS implements an RDF store that can be used as the foundation for the data layer in a client application, allowing the storage, query and manipulation of data from different services that can be merged easily thanks to the use of the RDF data model and the SPARQL query language.

RDFStore-JS shows how RDF stores and the SPARQL query language can play an important role in web development, not only as the persistence layer for server applications, but also as a middleware layer in the client.

The store also offers APIs with different granularity, one based in the use of raw SPARQL queries and RDF graphs serializations and an alternative API where RDF nodes are manipulated as JSON objects much in the same way as other JavaScript libraries. This API is also an example of the use of recent semantic standards for the integration of RDF data in JavaScript applications like JSON-LD or the RDF Interfaces API.

Additionally, RDFStore-JS also provides an evented API that fits in the JavaScript asynchronous execution model. SemanticKO builds on this evented API to provide an application development library, where declarative bindings between the DOM tree of a HTML document and the RDF graph maintained in the store are automatically updated to respond to the user interaction with the application.

## 4. REFERENCES

[1] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, pages 111–122, 2004.

[2] A. Harth and S. Decker. Yet another rdf store: Perfect index structures for storing semantic web data with contexts. Deri research paper, DERI, 2005.

[3] I. Hickson. Web workers. Last call WD, W3C, Sept. 2010. http://www.w3.org/TR/2009/WD-workers-20091222/.

[4] I. Hickson. Web storage. Last call WD, W3C, Oct. 2011. http://www.w3.org/TR/2011/WD-webstorage-20111025/.

[5] J. Smith. Wpf apps with the model-view-viewmodel design patter. *MSDN Magazine*, Feb. 2009.