LDApp - A JavaScript Linked Data Stack

http://bergos.github.io/ldapp-www/

Thomas Bergwinkl

bergnet.org bergi@axolotlfarm.org,
https://www.bergnet.org/people/bergi/card#me

Abstract. LDApp is designed as a very modular JavaScript Linked Data stack. A requirement for the modular design was an API to handle the graph data. RDF-Interfaces was the only JavaScript API standard, but lacks store handling and easy management of multiple parsers and serializers. The existing RDF-Interfaces API was extended to cover these use cases. To complete the framework, a JSON-LD integration is desirable to use JavaScript native language features to deal with graph data. New modules for public interfaces, authentication, authorization, persistence and JSON-LD handling have been written to use this API.

Keywords: Linked Data, RDF, JavaScript, JSON-LD, WebID, RDF Interfaces, Access Control

1 Introduction

Node.js[1] on the server-side gains more and more market share. A wide base of frameworks, running the same code on the server and client and asynchronous I/O are just a few reasons for this trend. This seems to make JavaScript also a perfect candidate for Linked Data. New technologies like JSON-LD[2] make more pragmatic approaches possible. But still fundamental modules and frameworks to deal with Linked Data are missing. The goal of the LDApp project is to lower barriers for Linked Data beginners. Existing frameworks are used were possible. Based on the established web application framework Express[3], LDApp creates the server side of the missing Linked Data framework. Also on the client side LDApp glues together modules to a uniform framework. To achieve acceptance it should be possible to install and run the basic framework in less than 5 minutes, so the entrance level for developers is low. This was realized by an out of the box solution without the need for configuration. The server and browser modules are shown in Fig. 1 and Fig. 2.

2 RDF APIs

To create a complete Linked Data stack, standard APIs are a must. Beside serialisation dependent APIs (JSON-LD, RDFa[4]), RDF-Interfaces [5] was the only generic standard. RDF-Interfaces defines interfaces for graphs, triples and

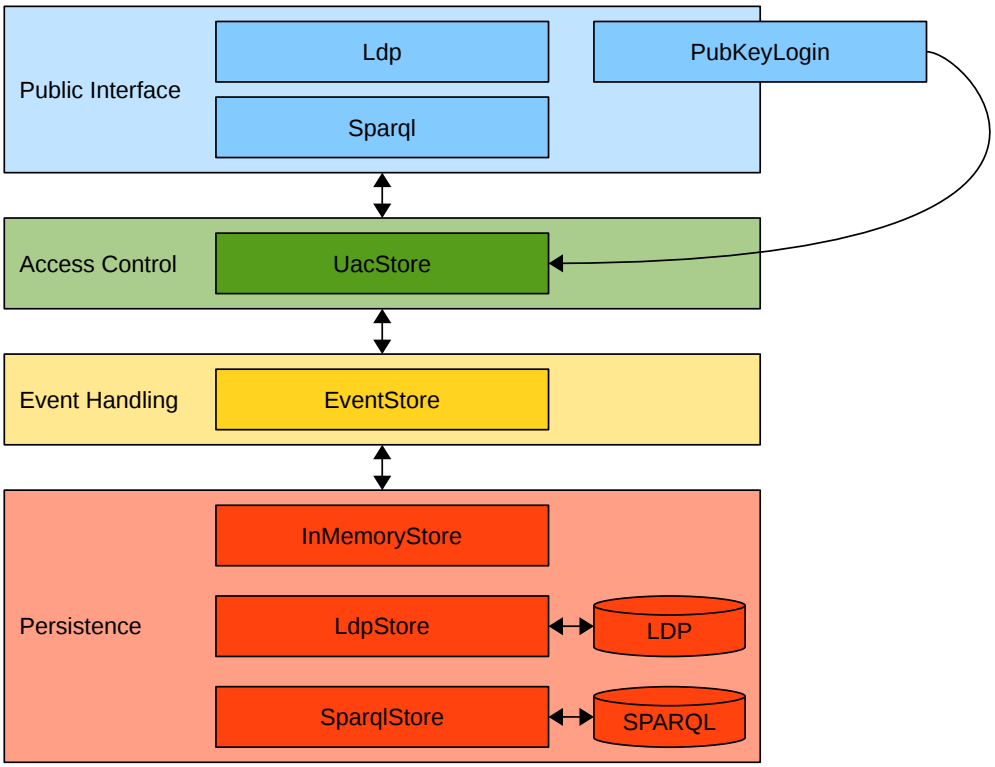

Fig. 1. LDApp modules on the server

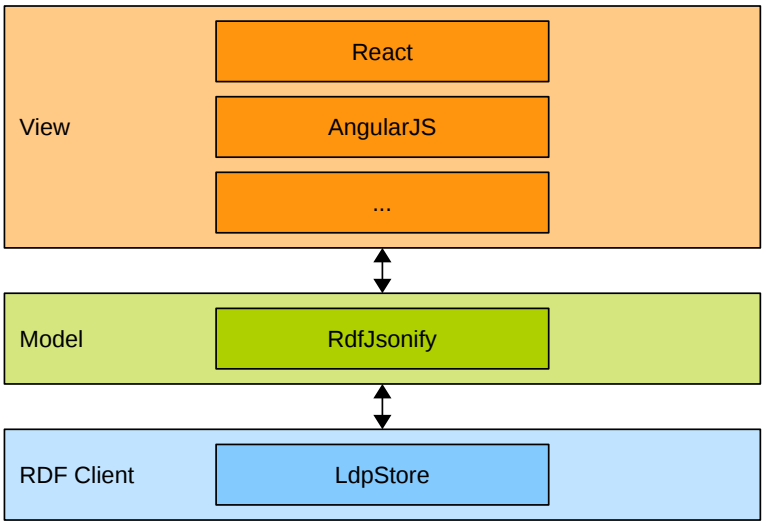

Fig. 2. LDApp modules in the browser

nodes, but the handling of multiple graphs is not specified. The concept of named graphs was not covered at all. This makes real world use cases more complicated to handle in JavaScript. Nevertheless the things covered by RDF-Interfaces look very well. As the status of the specification was close to final, the decision was made to use that standard and create a second, extending specification for the missing topics. The new specification was named RDF Interfaces Extension [6] or short RDF-Ext. A Store interfaces was defined to handle read and write access to named graphs in an asynchronous way. Also the parser and serializer interfaces where extended to handle asynchronous data. This requirement was introduced, because of the JSON-LD standard. The new ECMAScript 6 standard[7] contains promises to handle asynchronous calls in a more reasonable way. As promises can be implemented using only ECMAScript 5 features, there are already many implementations. Because of the many asynchronous function calls, the decision was made to support Promises in the RDF-Ext specification.

2.1 Implementation (RDF-Ext)

The reference implementation requires a RDF-Interfaces implementation. For development, the reference implementation[8] was used and updated to the latest specification changes. Store interfaces were built for in memory triple stores, SPARQL, Linked Data Platform (LDP) / RESTful graph access in general and RDFStore. The SPARQL and LDP implementations require parsers and serializers. A Turtle parser was built on top of N3.js[9]. The reference JSON-LD library was used to expand and flatten the objects, so they are easy to convert. A rdflib.js [10] Store compatible object was created to use the RDF/XML parser to create RDF-Interfaces graph objects. Flat JSON-LD objects can be created with the JSON-LD serializer and further processed with the JSON-LD library, if required. The N-Triples serializer is based on the `toString` method of the RDF-Interfaces Triple interface.

3 LDP

Accessing the graph data should be possible via dereferencing. This allows using a RDF-Ext interface on the client side to access the graph data in a transparent way. The Linked Data Platform (LDP) standard [11] achieves this requirement, but also adds some more features. To simplify the implementation, only a subset of LDP was defined which should be supported. There was already a Node.js LDP implementation[12], but it was built on top of the Redland C libraries. The RDF-Ext store interface should be used consistently, so the underlying store can be changed. Therefore a new module was developed.

3.1 Implementation (LDP)

To support different serializations, we store parsers and serializers in a map with the mime type as the key. By default we use the parsers and serializers of the

RDF-Ext reference implementation. So the most popular formats are supported out of the box. Depending on the request method, incoming data is parsed based on the mime type map. The HTTP method than is mapped to the according store interface method.

- GET → .graph
- PATCH → .merge
- PUT → .add
- DELETE → .delete

If there is response data, the serializer map is used to translate the graph object. It takes less than 200 lines of code to implement a subset of LDP using this simple logic.

4 Authentication

The authentication must use IRIs and must be decentralized to cover the concept of the Web. WebID-TLS[13] was chosen, because of its simple design. One step in the authentication process is fetching the profile graph. This step must use the standard APIs. The existing implementation for Node.js[14] uses RDFStore, therefore a new implementation was built.

4.1 Implementation (Pubkey-Login)

There are other public key authentication standards for decentralized identities, so the module was named Pubkey-Login to cover future extensions. To handle other standards, the authentication process is splitted into `getAssertion` and `validate`. Both methods use an assertion object based on Mozilla Persona standard[15]. Each method can be replaced separately. For the WebID process `getAssertion` fetches the public key and identity as an IRI from the client certificate. The `validate` method uses the RDF-Ext Store to fetch the WebID profile graph and compares the public key from the certificate with the one from the graph.

5 Authorization

The authorization to access graphs and triples must use standard APIs. To handle access control on the level of triples, Universal Access Control [16][17] was chosen. Universal Access Control was developed in an earlier PHP project [18], because of the lack of flexible triples access control standards. The idea behind Universal Access Control is a tree of filters, which describes the way to traverse a graph. On each node of the tree the access mode can be attached. The subgraph, described by the filter and mode, contains the triples the user has access to. Filter trees are managed as authorizations. Authorizations can be assigned directly to a user, group or role. Roles can be grouped and assigned to users and groups.

5.1 Implementation (UAC)

The authorization rules are read and mapped once from a RDF-Interfaces graph object into a JSON object for performance reasons. The parsed authorizations and roles are used by the AccessControl class to filter graph objects according to the defined access rules. There is also a store interface wrapped around the AccessControl class, which filters the read and write access to another store. Access control can be realized just by putting the UACStore in front a store.

6 JSON-LD integration

The JavaScript Object Notation (JSON) allows easy handling of structured data in JavaScript. JSON-LD was designed to interpret JSON data as Linked Data with minimal changes. This makes JSON-LD a perfect candidate to handle Linked Data in JavaScript. The JSON-LD parser and serializer of RDF-Ext can be used to translate graph data. This must be done always in the same way. To avoid code duplication a module just for this process must be implemented. Also the interface of the module should look like a REST interface for a JSON object store. This lowers the entrance level to Linked Data. Existing applications can be migrated very easy.

6.1 Implementation (RDF-JSONify)

The JSONify class implements the methods `delete`, `get`, `patch` and `put`, which are known from the HTTP standard. They also behave in the same way. Additionally a JSON-LD context must be provided to translate the graph data to JSON objects. This can be done per method call or based on the URL using a base path or regular expression. The asynchronous calls are handled using Promises. JSONify also has a class to deal with objects that have a high chance to be read multiple times. CachedJSONify returns the object synchronous, if it's cached. Though callbacks can be reduced, which produces less time consuming updates in frameworks like AngularJS or React.

7 Conclusion

LDApp is a big step forwards to lowers the entrance level to Linked Data for JavaScript developers. It takes less than 5 minutes to get a running Linked Data stack without the need for configuration. The modular approach allows to switch very fast the persistence layer, which is usefully for developers to start simple with an in memory store and change later to a more powerful store. But also the access control can be changed or completely removed if not required. A resource based access control like Web Access Control[19] could be implemented with minimal effort. Beside the LDP interface a SPARQL interface could be implemented and used instead or in addition. The gap of standards, which was recognized during the project realization, was filled by RDF-Ext. With LDApp

Linked Data becomes more attractive for JavaScript developers. Many recurring patterns are now covered by modules of LDApp which was published under the MIT license.

Acknowledgement. The author thanks Adrian Gschwend for technical assistance and Pascal Mainini for code review.

References

1. Joyent, Inc.: Node.js, http://nodejs.org/
2. Manu Sporny, Gregg Kellogg, Markus Lanthaler, Dave Longley, Niklas Lindstrm: JSON-LD, http://www.w3.org/TR/json-ld/
3. TJ Holowaychuk, Douglas Christopher Wilson: Express, http://expressjs.com/
4. Nathan Rixham, Mark Birbeck, Ivan Herman: RDFa API, http://www.w3.org/TR/rdfa-api/
5. Nathan Rixham, Manu Sporny, Benjamin Adrian, Mark Birbeck, Ivan Herman: RDF Interfaces Specification, http://www.w3.org/TR/rdf-interfaces/
6. Thomas Bergwinkl: RDF Interfaces Extension Specification, http://bergos.github.io/rdf-ext-spec/
7. Ecma International: Draft Specification for ES.next (Ecma-262 Edition 6), http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts
8. Nathan Rixham, Bergwinkl Thomas: RDF-Interfaces implementation, https://github.com/bergos/rdf-interfaces
9. Ruben Verborgh: Lightning fast, asynchronous, streaming Turtle for JavaScript, https://github.com/RubenVerborgh/N3.js/
10. Massachusetts Institute of Technology: rdflib.js https://github.com/linkeddata/rdflib.js/
11. Steve Speicher, John Arwe, Ashok Malhotra: Linked Data Platform, http://www.w3.org/TR/ldp/
12. Sebastian Tramp: Linked Data Platform for Node, https://github.com/AKSW/node_ldp
13. Henry Story, Stphane Corlosquet, Andrei Sambra, Toby Inkster, Bruno Harbulot: WebID-TLS Specification, http://www.w3.org/2005/Incubator/webid/spec/tls/
14. Baptiste Lafontaine: node-webid, https://github.com/magnetik/node-webid
15. Mozilla Foundation: Mozilla Persona, https://developer.mozilla.org/en-US/Persona
16. Thomas Bergwinkl: Universal Access Control, http://ns.bergnet.org/uac/0.1/index.html
17. Thomas Bergwinkl: Generic access control for triplestores, https://www.bergnet.org/people/bergi/files/documents/2014-02-14/index.html#/2
18. Thomas Bergwinkl: ResourceMe, https://resourceme.bergnet.org/
19. World Wide Web Consortium: Web Access Control, http://www.w3.org/wiki/WebAccessControl

