EDN-LD: A simple linked data tool

James A. Overton[1]*
[1]Knocean, Toronto, Ontario, Canada

ABSTRACT

EDN-LD is a set of conventions for representing linked data using Extensible Data Notation (EDN) and a library for conveniently working with those representations in the Clojure programming language. It provides a lightweight alternative to existing linked data tools for many common use cases, much in the spirit of JSON-LD. We present the motivation and design of EDN-LD, and demonstrate how it can clearly and concisely transform tables into triples.

1 INTRODUCTION

EDN-LD is a set of conventions for representing linked data using Extensible Data Notation (EDN),[1] and a library for conveniently working with those representations in the Clojure programming language.[2] Clojure is a modern Lisp that runs on the Java Virtual Machine (JVM) and has full access to the vast ecosystem of Java libraries. Since many linked data libraries and tools also target the JVM, Clojure is a tempting alternative to Java for working with linked data. Tawny-OWL is another example of a linked data tool written in Clojure (Lord, 2013), however it is focused on ontology development and takes quite a different approach from EDN-LD. With this project our goal is to provide a lightweight alternative to existing linked data tools for many common use cases, much in the spirit of JSON-LD.[3] In this presentation we discuss the motivation and design of EDN-LD, and demonstrate how it can clearly and concisely transform tables into triples.

EDN-LD is open source software, published under a BSD license. The source code is written in a literate style, with extensive unit tests. It is available on GitHub[4] with a tutorial that also serves as an automated integration test. Our interactive online tutorial can be used without needing to install Clojure.[5] Feedback and contributions are welcome on our GitHub site.

2 JSON-LD

EDN-LD shares many of the motivations and goals of JSON-LD. First we will discuss the benefits and shortcomings of JSON-LD, then show how EDN-LD improves on it in several respects.

JavaScript Object Notation (JSON)[6] is a subset of the JavaScript programming language that is widely used for expressing literal data within JavaScript programs. JSON's elements are: null, booleans, strings, and numbers. Elements can be combined into arrays and objects, where the latter are effectively maps from strings to other values. These simple elements are common to virtually every programming language, and JSON is now widely used for data transfer between programs. It has replaced heavier formats such as XML in many applications.

JSON has many limitations, including a lack of comments, ambiguous numbers, and the lack of any mechanism for extending its types. In practise, strings are used to represent most types of data, but since it is difficult to attach type information to aid in their interpretation, this can quickly lead to ambiguity.

The ubiquity of JSON was one motivation for the JSON-LD W3C Recommendation: "A JSON-based Serialization for Linked Data".[7] In JSON-LD strings are used to represent IRIs (and compact IRIs) for resources, plain literals can be strings, and typed literals are objects (maps) with a special `@value`, `@type`, and `@language` keys. Graphs and datasets are represented as nested objects (maps) and sets are represented by arrays, with details depending on the chosen "Document Form".

The core of JSON-LD is the `@context` map, which can be specified inside a JSON record, externally using a link, or provided by the consuming application. The context allows for strings to be interpreted as IRIs, for compact IRI strings to be expanded, and for types to be attached to literals. Since the context can be supplied externally, existing JSON data can be reinterpreted as JSON-LD by providing an appropriate context.

JSON-LD is an exciting addition to the ecosystem of linked data tools, but it is constrained by the limitations of the JSON format. The heavy use of strings, in particular, can make it difficult to distinguish between a literal string, a compact IRI, or a fully resolved IRI. The complex context processing[8] and expansion algorithms[9] are indicative of this problem, as is the need for several similar-but-different "Document Forms". EDN-LD uses the richer elements and structures available in EDN to reduce these problems.

3 EXTENSIBLE DATA NOTATION

Like JSON and JavaScript, Extensible Data Notation (EDN) is the a data format at the core of Clojure. The basic EDN elements are: nil, booleans, strings, characters, symbols, keywords, integers, and floating point numbers. These can be combined into lists, vectors, maps, and sets. Any element can serve as the key or value of a map. EDN is extensible in the sense that it allows for *tagged elements*, indicated by a special tag followed by an EDN element. EDN also allows two kinds of comments. Multiple alternatives to strings (i.e. keywords and symbols), more carefully defined numbers, sets, and more flexible maps all make it easier to express complex data efficiently and unambiguously in EDN than in JSON.

*To whom correspondence should be addressed: james@overton.ca

[1] https://github.com/edn-format/edn
[2] http://clojure.org
[3] http://json-ld.org
[4] https://github.com/ontodev/edn-ld
[5] http://try.edn-ld.com
[6] http://json.org
[7] http://www.w3.org/TR/json-ld/
[8] http://www.w3.org/TR/json-ld-api/#context-processing-algorithms
[9] http://www.w3.org/TR/json-ld-api/#expansion-algorithms

EDN does not have a type system and does not include schemas. However several schema systems have been created for validating EDN data structures. EDN-LD uses Prismatic's Schema library[10] to specify the required "shapes" for various elements.

4 EDN FOR LINKED DATA

In EDN-LD as in JSON-LD, IRIs and blank node identifiers are represented by strings. IRIs can be *contracted* to keywords using a context: a map from keywords to IRIs or other contractions. Contractions can be expanded to IRIs using the same context. Literals are always represented as maps with a special `:value` key for the lexical value, and optional `:type` and `:lang` keys. Discrete triples and quads are represented with vectors. Graphs and datasets are represented as nested maps from graph IRI to subject IRI to predicate IRI, ending with a set of objects. These two "document forms" have very different shapes, suited to different processing goals, e.g. sequences of triples for streaming and filtering, and nested maps for sorting and selecting. EDN-LD uses Apache Jena[11] to read from and write to a wide range of linked data formats.

Figure 1 shows an EDN-LD context. It includes a `:dc` prefix for Dublin Core metadata elements, and an `:ex` prefix for the example domain. The `nil` key indicates that its value `:ex` is the default prefix. The `:title` and `:author` contractions expand (recursively) to Dublin Core IRIs.

```
(def context
  {:dc     "http://purl.org/dc/elements/1.1/"
   :ex     "http://example.com/"
   nil     :ex
   :title  :dc:title
   :author :dc:author})
(expand context :title)
; "http://purl.org/dc/elements/1.1/title"
```

Fig. 1. An example of an EDN-LD context, showing an `expand` function call on the `:title` contraction, and the expanded IRI that is returned

Figure 2 shows an example of a simple data conversion pipeline using EDN-LD. First we define a map from names (strings) to contracted resource IRIs and merge our context with the default prefixes. The `->>` is a "threading macro" that inserts the first value as the last argument to the second function, and so on, letting deeply nested function calls be clearly expressed as "pipelines". Here "books.tsv" is the name of a file in tab-separated values format and `read-tsv` is a function that returns a sequence of maps for each row, each with column names as keys. The `assign-subject-iri` function is called on each of the maps to add a `:subject-iri` key with appropriate value. Then `triplify` is used to convert the maps to triples, represented as vectors: subject keyword, predicate keyword, and object keyword or literal map as determined by the `resources` map. The keywords represent contracted IRIs, and the `expand-all` function converts them to full IRI strings. Finally the `write-triples` function writes the results to the "books.ttl" Turtle file using our specified prefixes. By using keywords to distinguish contracted IRIs from full IRIs and literal data, and consistently using maps for literal data, we gain more control over the interpretation of strings than JSON-LD, without loss of concision.

```
(def resources
  {"Homer" :Homer})
(def prefixes
  (merge
    default-prefixes
    context))
(->> "books.tsv"
     read-tsv
     (map assign-subject-iri)
     (mapcat #(triplify resources %))
     (map #(expand-all context %))
     (write-triples "books.tsv"
                    prefixes))
```

Fig. 2. An example of an EDN-LD conversion pipeline

5 FUTURE WORK

EDN-LD is still in development, but available for use. We plan to implement convenient syntax for RDF collections (linked lists), and for various OWL constructs including annotation axioms and class expressions. We are also considering a ClojureScript implementation of EDN-LD. ClojureScript is a language that is closely related to Clojure, compiling to JavaScript rather than JVM bytecode. Dual Clojure and ClojureScript libraries are becoming increasingly common. But a ClojureScript version of EDN-LD could not use Jena, and would need alternative methods for reading and writing linked data files.

6 DEMONSTRATION

At ICBO we plan to demonstrate the use of EDN-LD for transforming tables to triples, and for efficiently filtering large linked data files to specified subsets.

7 CONCLUSION

EDN-LD was developed for the Immune Epitope Database (IEDB), and was preceded by several related systems for working with linked data and ontologies using Clojure. These techniques have proved valuable for rapid development of data processing workflows, merging disparate sources of biological data. EDN-LD improves on JSON-LD in several respects, and is well suited to working with linked data in Clojure.

ACKNOWLEDGEMENTS

The author was supported in this work by the Immune Epitope Database and Analysis Project, funded by the National Institutes of Health [HHSN272201200010C].

REFERENCES

Lord, P. (2013). The Semantic Web takes wing: Programming ontologies with Tawny-OWL. *OWLED 2013*.

[10] `https://github.com/Prismatic/schema`

[11] `http://jena.apache.org`