Linked Data Objects (LDO): A TypeScript-Enabled RDF Devtool

Jackson Morgan(✉)

O.team, Washington DC 20001, USA
jackson@o.team

Abstract. Many RDF devtools exist for JavaScript that let developers read and modify RDF data, but they often opt to use proprietary interfaces and don't fully leverage the methods of reading and writing data with which JavaScript developers are familiar. This paper introduces Linked Data Objects (LDO), a JavaScript-based devtool designed to make reading and writing RDF as similar to programming traditional TypeScript applications as possible. LDO generates TypeScript typings and a JSON-LD context from an RDF Shape like ShEx. A JavaScript Proxy uses the generated code to serve as an abstracted interface for an underlying RDF/JS dataset, allowing the developer to manipulate RDF as a TypeScript object literal. LDO is currently in use in a small number of RDF-based projects and our user studies indicate that LDO's interface-parity with TypeScript object literals is preferable to proprietary interfaces in other JavaScript RDF devtools. Finally, this paper proposes future work to make LDO even more approachable for JavaScript developers.

Keywords: RDF · Devtool · JavaScript · TypeScript

Resource Type: Software Framework
License: MIT License
Permanent URL: https://purl.archive.org/o.team/ldo
Canonical Citation: https://doi.org/10.5281/zenodo.7909200

1 Introduction

Ease of use is paramount for JavaScript developers when they are evaluating which devtool to use in their project, and members of the semantic web community are rightfully striving to make RDF as approachable for novice developers as possible [20]. There have been many approaches to making RDF accessible in JavaScript, the web's native language.

Some libraries like RDF/JS [1], clownface [2], Tripledoc [3], and rdflib [4] implement unique interfaces for accessing and modifying data. For example, RDF/JS employs its own methods like `quad`, `namedNode`, and `dataset` to manipulate data. A developer can create a new quad by writing (Fig. 1):

T. R. Payne et al. (Eds.): ISWC 2023, LNCS 14266, pp. 230–246, 2023.
https://doi.org/10.1007/978-3-031-47243-5_13

```
dataset.add(quad(namedNode("a"), namedNode("b"), namedNode("c")))
```

Fig. 1. Creating a new quad in RDF/JS

While this might be a straightforward technique for anyone versed in the semantic web, it is daunting for new developers who need to read API documentation and the basic concepts of linked data to use it properly.

We hypothesize that the closer the developer experience is to the native environment they're used to, the easier it will be for a developer to use a tool, even if advanced use-cases require those developers to leave the comfort zone of their native environment. Many RDF devtools have employed techniques that work towards that goal. In this paper, we will explore those techniques and present Linked Data Objects (LDO), a library designed to allow developers to easily use RDF in a TypeScript setting specifically.

2 Related Work

A plethora of JavaScript libraries exist to help JavaScript developers use RDF data. In this section, we discuss these libraries, the design choices they made, and their strengths and weaknesses.

Matching JSON's Interface. JSON is the primary data structure for JavaScript, and therefore is an interface that feels natural for JavaScript developers. A few libraries acknowledge this in their design, most notably JSON-LD [5]. As a serialization of RDF, JSON-LD lets developers read and write data to a document using JSON's interface. Its ease of use is probably why many of the subsequently referenced libraries – including LDO – use it as a basis.

JSON-LD is not without its flaws, however. Pure JSON is a tree, not a graph. This means that with raw JSON-LD, a developer cannot traverse a full circuit of the graph without searching the graph for a certain node. It is possible to flatten a JSON-LD document so that a developer can directly look up objects by their subject ids, however, this would require extra work on the part of the developer. A true match of JSON's interface would allow a developer to use uninterrupted chaining (Fig. 2).

```
// uninterrupted chaining
person.friend.name
// chaining interrupted by a lookup on a flat JSON-LD doc
jsonldDocument[person.friend["@id"]].name
```

Fig. 2. A comparison of uninterrupted chaining and a lookup on flattened JSON-LD

Libraries that use JSON-LD as a base interface like rdf-tools [9] and shex-methods [10] are limited by JSON-LD's raw structure as a tree.

Schemas. While RDF's formal semantics follow the open world assumption [21], TypeScript naturally follows a closed world assumption. To reconcile this, many libraries have adopted a Schema system to define how data should be structured.

Schema systems fall into three categories. Firstly, some devtools like LDflex [6], RDF Object [7], and SimpleRDF [8] ask the user to provide a JSON-LD context. While this is a simple solution, JSON-LD contexts do not strongly define the "shape" of an RDF node (as languages like ShEx and SHACL do) and lacks useful features like asserting a predicate's cardinality (as languages like OWL do). For example, JSON-LD contexts only define possible fields, not where those fields should be used. A context may define the existence of fields "name", "friends", "file extension", and "dpi", but not clarify that only "name" and "friends" should be used on a "Person" node.

Secondly, some devtools translate a language with strong definitions into a JSON-LD context. For example, rdf-tools [9] generates a context from OWL, and shex-methods [10] generates a context from ShEx [15]. Unlike the libraries that depend on JSON-LD context alone, these libraries have provisions to ensure certain predicates are only used on their intended types.

Thirdly, some devtools like Semantika [11] and ts-rdf-mapper [12] ask the user to define the schema in JavaScript itself. They develop their own unique JavaScript interface to let the developer define how their actions in JavaScript translate to RDF. These libraries are similar to Java libraries in the semantic web space including So(m)mer [26] and Jenabean [27] which use Java decorators to achieve the same goal.

If the goal is to design a devtool that is as close to JavaScript as possible, the third option sounds like the obvious technique to employ, but it does have a downside. Any development work done to define a schema in a JavaScript-only environment is not transferable to other languages. Philosophically, semantic data should be equally as usable on any platform no matter what language it's using. Having a single schema that works with multiple programming languages (like OWL, ShEx, or SHACL) makes it easier for developers on many different platforms to read and write the same semantic data.

For LDO, we have decided to employ the second option, using a strongly defined language-agnostic schema (in our case ShEx). As will be discussed in the "User Studies" section, this design choice has a negative impact on approachability for JavaScript developers who are unfamiliar with RDF. However, the "Future Work" section discusses the potential for a universal schema library accessible to all developers. In that future, the relative unapproachability of RDF schemas for JavaScript developers is inconsequential.

TypeScript. Multiple studies [13, 14] have shown that strongly typed languages are more useful for developers than weakly typed languages because strongly typed languages permit useful tools like auto-suggest and type checking that inform developers how to interact with data.

While many libraries like RDF/JS [1], clownface [2], Tripledoc [3], and rdflib [4] use TypeScript, their typings apply to the interface for accessing data and not the data itself. Other libraries like rdf-tools [9], shex-methods [10], and ts-rdf-mapper [12] generate typings based on a schema. This allows developers to know, for example, that a "Person" has a "name" field that is type "string." We decided to do the same for LDO.

Similar Java Libraries. LDO is conceptually similar to Java RDF code generators like Owl2Java [28]. These take some standard (like OWL) and generatse POJOs that can be used in the project. However, providing a native-feeling environment is a bit more difficult in JavaScript than Java. In Java, developers are used to interacting with a class

and methods, so a code generator only needs to generate methods that match a given schema-like input. That could be an approach in JavaScript, but JavaScript developers are more likely to interact with data through raw JSON rather than JavaScript classes. That's why LDO considers all operations that could possibly be done on a raw JSON object literal (the " =" operator, the "delete" operator, array iterators, array methods, chaining etc.).

The aforementioned libraries and their design decisions are displayed in Table 1.

Table 1. Various RDF JavaScript Devtools rated on Design Considerations

Library	Has a JSON-like interface	Not represented as a tree	Uses a Schema	Typings are generated from Schema
RDF/JS [1]		✓		
clownface [2]		✓		
Tripledoc [3]		✓		
rdflib [4]		✓		
JSON-LD [5]	✓		JSON-LD Context	
LDflex [6]		✓	JSON-LD Context	
RDF Object [7]		✓	JSON-LD Context	
SimpleRDF [8]	✓	✓	JSON-LD Context	
rdf-tools [9]	✓		Generated (OWL)	✓
shex-methods [10]			Generated (ShEx)	✓
Semantika [11]		✓	Defined in JS	
ts-rdf-mapper [12]	✓	✓	Defined in JS	✓
So(m)mer [26]	N/A	✓	Defined in Java	✓
jenabean [27]	N/A	✓	Defined in Java	✓
Owl2Java [28]	N/A	✓	Generated (OWL)	✓
LDO	✓	✓	Generated (ShEx)	✓

3 Linked Data Objects (LDO)

Linked Data Objects (LDO) is designed to satisfy the design considerations discussed above. It contains two main libraries: ldo[1] and ldo-cli[2]. We will also mention a few dependencies that were built to support LDO: shexj2typeandcontext[3], jsonld-dataset-proxy[4] and o-dataset-pack[5].

Generally, LDO's developer experience is divided into five steps (as described in Fig. 3): (1) building from the schema, (2) parsing raw RDF, (3) creating a Linked Data Object, (4) reading/modifying data, and (5) converting data back to raw RDF.

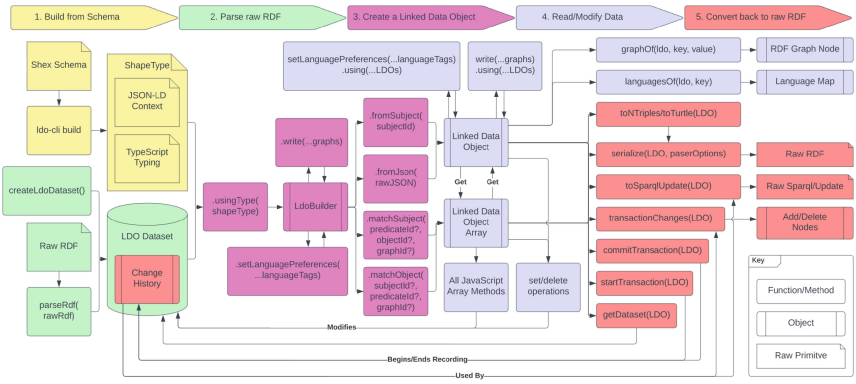

Fig. 3. A diagram showing all the developer-facing methods and their intended flow as a developer experience.

3.1 Building From the Schema

As mentioned in the "Related Work" section, we decided to orient the LDO around a language-agnostic schema. We chose ShEx [15] as our schema system for relatively arbitrary reasons: ShEx is more popular in the Solid [30] community. However, we architected the system to be able to also accommodate alternatives like SHACL [16] if a converter is built.

By itself, a ShEx schema isn't useful for LDO. It can validate raw RDF, but as LDO strives to interact with data in a JSON-like way, two pieces of data must first be derived from a ShEx schema. First, a corresponding JSON-LD context can be derived from a schema to inform LDO about the shape at runtime, and second, a TypeScript typing can also be derived from the schema to perform type checking at compile-time (or more accurately in the case of TypeScript, transpile-time) and in a developer's IDE.

[1] https://purl.archive.org/o.team/ldo.
[2] https://purl.archive.org/o.team/ldo-cli.
[3] https://purl.archive.org/o.team/shexj2typeandcontext.
[4] https://purl.archive.org/o.team/jsonld-dataset-proxy.
[5] https://purl.archive.org/o.team/o-dataset-pack.

As TypeScript typings are required before compile-time, the schema conversion script must be run before then. A command line interface (CLI) is a common design pattern to execute such scripts, and LDO has an accompanying cli called ldo-cli.

Ldo-cli makes it easy for developers to set up their projects. By running one command in their TypeScript project (`npx ldo-cli init`), ldo-cli will install all required dependencies, create a folder for developers to store their ShEx schemas, provide an example ShEx schema, and add the "build" command to their project's metadata file (package.json).

Once the developer has initialized their project, they can modify and add ShEx schemas to their "shapes" folder. Running `ldo build --input {shapesPath} --output {outputPath}` or simply `npm run build:ldo` runs the script to convert shapes into context and typings.

As an example, suppose a developer uses the ShEx Schema defined in Fig. 4.

```
PREFIX ex: <https://example.com/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ns: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ex:FoafProfile EXTRA a {
  foaf:name xsd:string
    // rdfs:comment  "A profile has 1 names" ;
  foaf:title ns:langString *
    // rdfs:comment  "A profile has 0-∞ titles, and they could have translations" ;
  foaf:knows @ex:FoafProfile *
    // rdfs:comment  "A profile has 0-∞ friends." ;
}
```

Fig. 4. An example ShEx shape that will be used in all future examples

The "build" command uses the shexj2typeandcontext library which iterates over ShExJ (ShEx's JSON-LD serialization). It builds a context by inferring a predicate name. If a predicate is not explicitly labeled with `rdfs:label`, shexj2typeandcontext will choose a field name by looking at the end of a predicate URI. For example, the predicate http://xmlns.com/foaf/0.1/name will translate to a field name of `name`. The library also includes contingencies for overlapping field names. Any fields that could have more than one object are marked with `"@container": "@set"`.

This interpretation does not exactly map to the interpretation outlined in the JSON-LD specification as the absence of a "@container" field does not officially mean a cardinality of 1 in JSON-LD. Therefore, in this version of LDO, developers should only use a JSON-LD context that was generated by the "build" command and not one that was generated externally.

The example above produces this context in Fig. 5.

```
export const foafProfileContext: ContextDefinition = {
  name: {
    "@id": "http://xmlns.com/foaf/0.1/name",
    "@type": "http://www.w3.org/2001/XMLSchema#string",
  },
  title: {
    "@id": "http://xmlns.com/foaf/0.1/title",
    "@type": "http://www.w3.org/1999/02/22-rdf-syntax-ns#langString",
    "@container": "@set",
  },
  knows: {
    "@id": "http://xmlns.com/foaf/0.1/knows",
    "@type": "@id",
    "@container": "@set",
  },
};
```

Fig. 5. JSON-LD context generated by ldo-cli given the example ShEx shape.

Once a context is produced, shex2typeandcontext iterates over the ShExJ object a second time to construct the TypeScript typings. It uses the field names defined in the context to create TypeScript interfaces. The library does not account for every feature in ShEx as mentioned in the "Future Work" section; however, it does handle enough to be usable on basic schemas. The example shape above produces the following typing (Fig. 6):

```
export interface FoafProfile {
  "@id"?: string;
  "@context"?: ContextDefinition;
  name: string;
  title?: string[];
  knows?: FoafProfile[];
}
```

Fig. 6. TypeScript typings generated by ldo-cli given the example ShEx shape.

Finally, the "build" script produces a resource called a "ShapeType." ShapeTypes combine typings, context, and other metadata into one object so that it can easily be imported by the developer.

3.2 Parsing Raw RDF

A developer could receive RDF in many forms and can use LDO's `parseRdf()` function to convert it. ParseRdf uses N3.js [17] and JSON-LD Streaming Parser [18] to accept turtle, n-triples, JSON-LD, or any RDF/JS compatible dataset and converts it into an "LdoDataset." LdoDatasets implement the RDF/JS Dataset [19] interface but have an additional method that lets the developer create Linked Data Objects (Fig. 7).

```
const rawTurtle = `
@prefix example: <https://example.com/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix ns: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
example:Taggart
   foaf:name "Peter Quincy Taggart" ;
   foaf:knows example:Lazarus .
example:Lazarus
   foaf:name "Lazarus of Tev'Meck" ;
   foaf:title "Doctor"^^ns:langString ;
   foaf:title "Docteur"@fr ;
   foaf:knows example:Taggart .
`;
const ldoDataset = await parseRdf(rawTurtle, { format: "Turtle" });
```

Fig. 7. Use of the parsedRdf function preceded by raw RDF that will be used in upcoming examples.

3.3 Creating the Linked Data Object

A Linked Data Object represents a specific type and a specific subject inside a dataset. The developer can provide the expected type with the `usingType()` method by providing the "ShapeType" generated in step one. The `usingType()` method returns an "LdoBuilder," a class that will help build a Linked Data Object for the given type.

Some methods on the LdoBuilder, like the `write()` and `setLanguagePreferences()` methods, set preferences for the eventual Linked Data Object, but only return an LdoBuilder so that preferences can be set using the method chaining design pattern [22]. We will cover the specifics of `write` and `setLanguagePreferences` in the next section.

Finally, the developer must define the subject(s) they want their Linked Data Object to represent. The most common way of doing this is the `fromSubject()` method which returns a Linked Data Object representing the provided subject ID (Fig. 8).

```
import { FoafProfileShapeType } from "./ldo/foafProfile.shapeTypes";
const taggart = ldoDataset
  .usingType(FoafProfileShapeType)
  .setLanguagePreferences("en", "@none")
  .fromSubject("https://example.com/Taggart");
```

Fig. 8. Method chaining to get a Linked Data Object with the `usingType` and `fromSubject` methods.

The `fromJson()` method serves a similar purpose as the `fromSubject()` method, but instead of accepting a Subject ID, it accepts raw JSON. This JSON is then processed, added to the dataset, and turned into a Linked Data Object to return.

The `matchSubject()` and `matchObject()` methods are two more advanced ways to create a Linked Data Object. They both return arrays of Linked Data Objects corresponding to the matching predicate, object, and graph in matchSubject's case or the matching subject, predicate, and graph in matchObject's case (See Fig. 9). These

two methods require knowledge of quads and the underlying data structure of RDF. Therefore, it is expected that only users with knowledge of RDF will use these methods.

```
const listOfAllPeopleWhoKnowLazarus = ldoDataset
  .usingType(FoafProfileShapeType)
  .matchSubject(
    "http://xmlns.com/foaf/0.1/knows",
    "https://example.com/Lazarus",
    null
  );
const listOfAllPeopleKnownByLazarus = ldoDataset
  .usingType(FoafProfileShapeType)
  .matchObject(
    "https://example.com/Lazarus",
    "http://xmlns.com/foaf/0.1/knows",
    null
  );
```

Fig. 9. Advanced matching methods for constructing a set of Linked Data Objects

3.4 Reading/Modifying Data

Linked Data Objects. A Linked Data Object is built with the library "jsonld-dataset-proxy." This uses JavaScript's "Proxy" object [23] to intercept and redefine fundamental operations of an object. A Linked Data Object adopts the same interface as a traditional JavaScript object, but under the hood, it is translating any fundamental operation into an operation on a dataset (See Figs. 10 and 11).

```
console.log(taggart.name);
// translates to
console.log(
  ldoDataset.match(
    namedNode("https://example.com/Taggart"),
    namedNode("http://xmlns.com/foaf/0.1/name"),
    null
  ).toArray()[0].object.value
);
```

Fig. 10. A JSON get operation translates to a "match" operation on an RDF Dataset

Notice that in Fig. 11, setting a new name is translated to both "delete" and "add" operations. If the "name" field in the generated JSON-LD context contained the metadata `"@container": "@set"`, the Linked Data Object would have only done an add operation because it interprets a "set container" as permission to allow multiple quads with the "name" predicate.

While most JSON operations perfectly map to RDF operations, some require creative interpretation. For example, in LDO, `delete taggart.knows[0]` and `tagart.knows[0]= undefined` are different operations. The first will remove all quads associated with an object (in this case, it would remove all quads associated

```
taggart.name = "Jason Nesmith";
// translates to
ldoDataset.deleteMatches(
  namedNode("https://example.com/Taggart"),
  namedNode("http://xmlns.com/foaf/0.1/name")
);
ldoDataset.add(quad(
  namedNode("https://example.com/Taggart"),
  namedNode("http://xmlns.com/foaf/0.1/name"),
  literal("Jason Nesmith")
));
```

Fig. 11. A JSON set operation translates to a delete and add operation on an RDF Dataset

with Dr. Lazarus), and the second will only delete the adjoining quad (in this case Dr. Lazarus still exists, but Taggart doesn't know him). While this is a departure from the JSON-purist approach we've taken thus-far, we believe it is necessary here because both operations are useful for a developer.

Linked Data Object Arrays. LDO will create an array for any key with a cardinality of over 1. Linked Data Object Arrays are more than just an array of Linked Data Objects, they are JavaScript proxies themselves that intercept fundamental operations. As such, reading from or writing to an index (for example `taggart.knows[0]`) and every JavaScript array method translates to operations on the underlying RDF dataset.

One point of difference is that JavaScript arrays are ordered, and RDF sets are not. This difference is especially relevant when implementing JavaScript array methods that depend on the ordered nature of arrays like `splice()` or `sort()`. In the spirit of keeping LDO as similar to JSON as possible, Linked Data Object Arrays maintain an internal state that keeps track of the order of entities. However, developers are warned not to depend on order as edge cases like modifying the dataset without a Linked Data Object can cause the ordering to change unexpectedly.

Non-JSON Concepts. RDF contains features like graph support and language tag support that don't perfectly map onto JSON paradigms. While the JSON-LD specification has solutions for these features, we found that they break the simplicity of the TypeScript typings generated in step 1. Instead, we opted to handle these features by using functions outside of the Linked Data Object.

Graph Support. Every quad has a graph, and to enable graph support, LDO needs to answer two questions for the developer: "How do I discover which graph certain specific information is on?" and "If I add new data, how do I control which graphs it is written to?".

The first question can be answered with the `graphOf()` function. This function essentially lets the developer describe a triple using a Linked Data Object (the subject), a key on that object (the predicate), and an array index in the case that the key has a cardinality of greater than 1 (the object). It returns an array of all graphs on which that triple exists. For example, `graphOf(taggart, "knows", 0)` might return "[defaultGraph]" because the triple`ex:Taggart foaf:knows ex:Lazarus` exists on the default graph.

The second question is answered with the `write().using()` function. The developer can provide a list of RDF-JS compatible graph nodes to the `write` function and a list of Linked Data Objects to the `using` method, and any succeeding triple additions to the dataset will be made in the defined graphs. The write graphs can also be defined when creating the linked data object using the `write()` method on the LdoBuilder (Fig. 12).

```
console.log(graphOf(taggart, "name")); // Logs [defaultGraph]
write(namedNode("otherGraph")).using(taggart);
taggart.name = "Jason Nesmith";
console.log(graphOf(taggart, "name")); // Logs [otherGraph]
```

Fig. 12. A demonstration of graph support in LDO

Language Tag Support. Having access to all languages of a langString is nice to have, but in most cases, developers have a preference for a specific language, and can use the `setLangaugePreferences().using()` function to communicate that with LDO (Fig. 13).

```
const titleLanguageMap = languagesOf(taggart, "title");
titleLanguageMap.en?.add("Commander").add("Mr.");
titleLanguageMap.es?.add("Comandante").add("Sr.");
titleLanguageMap["@none"].add("Captain").add("Mr.");
setLanguagePreferences("es", "@none").using(taggart);
console.log(taggart.title[0]); // Logs Comandante
```

Fig. 13. A demonstration of Language Tag support in LDO

3.5 Converting Data Back to Raw RDF

Once modifications have been made to the data, developers will want to convert their data back into a form that's applicable outside of LDO. This form could be an RDF JS Dataset – in which case the `getDataset()` function can be used – or an RDF serialization like turtle, n-triples, or JSON-LD – in which case the `serialize()`, `toTurtle()`, `toJsonLd()`, and `toNTriples()` functions can be used.

Tracking Changes. Some systems, like Solid [30], allow SPARQL update queries to modify data, developers interfacing with such systems may prefer update queries over raw RDF documents. To build a SPARQL update query, we first must keep track of changes made by the developer. LdoDataset extends TransactionalDataset from the "o-dataset-pack" library. A transactional dataset keeps an internal record of all changes made during a transaction. To start a transaction, the developer can use the `startTransaction()` function and to end a transaction, they can use the `commitTransaction()` function (Fig. 14).

```
startTransaction(taggart);
taggart.name = "Jason Nesmith";
// Logs:
// DELETE DATA {
//   <https://example.com/Taggart> <http://xmlns.com/foaf/0.1/name>
//   "Peter Quincy Taggart" .
// }; INSERT DATA {
//   <https://example.com/Taggart> <http://xmlns.com/foaf/0.1/name>
//   "Jason Nesmith" .
// }
console.log(await toSparqlUpdate(taggart));
commitTransaction(taggart);
startTransaction(taggart);
// Logs "" because no changes are in this transaction
console.log(await toSparqlUpdate(taggart));
```

Fig. 14. A demonstration of transactions and change tracking

4 User Studies

To validate the design consideration assumptions, we conducted interviews with nine software engineers of varying proficiency (five with no proficiency in RDF but experience in JavaScript and four with RDF proficiency). Interview subjects were recruited by a social media post on our personal Twitter and LinkedIn pages as well as invitations to friends. Each one-hour study consisted of an opening interview discussing the participant's knowledge of JavaScript and RDF followed by a hands-on study. Participants were asked to clone a starter TypeScript repo and were given a string of turtle-format RDF representing a user profile. They were then asked to use LDO to change the name listed on the profile. Afterwards, participants were questioned about their experience. Finally, they were asked to read the documentation for another new-developer focused JavaScript library, LDflex, and provide an opinion on its interface choices versus LDO's. LDflex was selected a tool for comparison as both tools focus on developer friendliness for RDF. The major takeaways from the user interviews are listed below.

Building Schemas. The process of writing and building schemas was generally understood by RDF proficient participants. Each of them navigated the build process and understood the purpose of the generated files when asked. Though one did express concern for RDF novices, noting "I think it would be hard to teach them how to write ShEx and Turtle."

Novices had a more difficult time with schemas, they were unable to construct their own schemas and relied on an auto-generated version. One participant went as far as saying that they preferred LDflex to LDO because it is "easier to use and to get something off the ground more quickly. There are fewer steps." LDflex relies on a simple JSON-LD context as its schema replacement. This participant believed that it would be easier for them to build a JSON-LD context on their own than a ShEx schema.

Converting From Raw RDF. Converting from raw RDF to a Linked Data Object was similarly difficult. Even one of the experienced RDF developers stumbled on the process saying, "I got confused around the [RDF/JS] dataset term because at that point, all I worked with is rdflib," thus showing that familiarity of RDF/JS interfaces are not

universal and will require additional education even with seasoned RDF JavaScript developers.

While the RDF novices were unphased by the existence of a Dataset, the `parseRdf()` function presented a challenge. "BaseIRI was a bit difficult, but I might have gotten it given a bit of time. I'm just doing pattern matching." Indeed, every novice struggled with RDF parsing and particularly didn't understand the concept of a BaseIRI. A BaseIRI is often necessary when parsing raw RDF, but understanding it does require knowledge of RDF's quirks that novice developers don't possess.

One novice, however, did have positive feedback for the conversion stage saying, "Given that there's an example that I can copy paste, I know that I need to call these two functions to turn it into an object."

Manipulating Data. Once users had overcome the setup process, feedback was generally positive about manipulating data. The experienced RDF subjects expressed praise for the simplicity of LDO versus other libraries they've used. "In [other libraries I've used] I need to create a service from scratch… I'm writing the construct query directly. There's no update. You have to delete and insert triples. So, [LDO] simplified this," one said. "I do think having the types is helpful… it was really, really easy to read/write data and change it," said another.

Further positive feedback came from the novice participants. "There seems like there's a lot of value there if you're doing a lot of complicated operations. I do think having the types is helpful." Additionally, they affirmed the approachability of JSON-congruence, noting a preference for LDO's editing interface over LDflex's. "[LD-Flex's] syntax is really weird. Await on a foreach loop? You need to learn a new syntax which is extra work."

An unexpected recipient of praise was LDO's transaction system. One experienced RDF developer said, "That's cool that you can start the transaction and it's almost like a database. That's new. I don't remember any of these other libraries letting you do that. I really like that." Though, one of the non-RDF developers was confused that you didn't need to commit a transaction before calling `toSparqlUpdate()`.

5 Case Studies

LDO has also been used in a few projects. This section details two such projects and the experience of the developers working on them. Beyond the projects listed here, a small community of developers are using LDO, yielding 17 stars on GitHub and 1,719 total downloads from NPM as of May 8th, 2023.

Capgemini and Försäkringskassan. Försäkringskassan (the Swedish Social Insurance Agency) is a Swedish government agency responsible for financial welfare including, but not limited to, pensions, housing benefits, child allowances, and immigrant support. They approached Capgemini, a global consulting company, with the problem of digitizing welfare legislation. By representing legislation as linked data, they hope to provide automated tools to deliver social insurance as dictated by law.

Capgemini developer, Arne Hassel, decided to use LDO for the Försäkringskassan project proof of concept. "I'm very happy with LDO… I've used rdflib which is very powerful, but very verbose. Inrupt's client libraries are also powerful but don't connect with the vision of the data in my mind. LDO makes it feel more natural to work with the data," he said. "For me, the most vital part of LDO is that it has a very easy to understand representation of the graph. It works from subjects and you have properties that represent the predicates and you can follow that. For me, it's a very neat way of working with data, especially with types… That's one of the core things I like: it's as close to JSON as possible."

The proof of concept is built using React, and Arne has built his own React hooks like `useResource()` and `useSubject()` that correspond to parts of the LDO interface. For now, the project has read-only requirements, and modifying and writing data is not required, though Arne says that this will be a requirement in the future.

Arne was undaunted by LDO's setup process. "Ldo is easy once you have it set up. It's simple for me because I understand the concept of ShEx." Though he mitigated his praise saying, "There are reasons that I wouldn't use LDO. It's a one-man project, so I'm very comfortable with using it for proof of concepts, but when it comes to applying it to big production environments, I need some kind of assurance that someone will be able to fix bugs in a decent amount of time." He contrasted this with other devtools like rdflib which has existed for a long time or "inrupt/solid-client" which is supported by a well-funded company (Inrupt). "LDO feels the most natural to me, so if I could be sure there was support, that would be my choice."

Internet of Production Alliance. The Internet of Production Alliance is an organization focused on building open infrastructure in manufacturing. One of their initiatives is the Open Know-How (OKH) initiative which seeks to make designs and documentation for manufactured goods open, accessible, and discoverable. Alliance member, Max Wardeh wanted to build an application that would store designs and documentation on a Solid server and needed a tool to work with the RDF metadata for each of the manufactured goods. He chose LDO.

"I don't think we could have gotten [the OKH Solid project] done in the tight deadline without LDO," he said, "especially because we were changing things about the ontology during the project itself." He noted that the ability to update ShEx shapes as the ontology for their project changed helped decrease development time. Because LDO generated TypeScript typings, he was easily able to track where code needed to be updated with every change.

In Max's case, the use of ShEx wasn't a deterrent. In fact, he noted that LDO made it easier to work with Solid due to existing ShEx shapes. "We were able to use other established shapes like the 'Solid Profile,' and 'Solid File Structure' shapes. This was really key, especially as someone who's never done something in Solid before."

However, Max acknowledged that LDO is not useful in every use case. "In my mind, LDO is a front-end thing. If I were building out a data pipeline of some sort and the data is stored in triples, I'd probably go for RDF.ex for that kind of use case." This assessment is commensurate with the target audience for LDO.

6 Future Work

While LDO has made progress towards more usable RDF devtools, there is room for improvement as seen in the user interviews and beyond. Fortunately, the NL-Net foundation has agreed to fund part of the future work for LDO.

Novice Developer Ease-of-Use. As seen in the "User Studies" section, one of the points of contention with LDO's design was the requirement to create a ShEx schema. Developers did not want to learn a new language to define a schema. One simple solution is building support for a schema language that's more comfortable for JavaScript developers like JSON Schema [24]. JSON Schema is a well-adopted schema language structured using JSON. JSON-LD Schema [29] expands JSON Schema for use in RDF. The only feature that a JavaScript developer may find daunting is the fact that JSON-LD Schema encourages developers to define URLs in order to make the jump to its status as an RDF Schema.

Ultimately, when it comes to schemas, defining predicates with URLs is unavoidable and is therefore inherently unapproachable to novice JavaScript developers. It might be prudent to say that schemas should only be defined by experienced developers and then distributed to novice developers via known mediums like NPM. Downloading schemas is not only easier for novice developers, but it also encourages data-interoperability as projects will have the same definitions for objects. This potential future could make RDF development even easier than traditional software development as the user will no longer need to define or research their own data standards when they start a project. They can depend on data standards created by the community that are easily downloadable.

Another point of user contention was the process of fetching raw RDF and converting it into a Linked Data Object. Optimizing this process was out of scope for this paper, but user studies indicate that this should be a focus for future work. Providing a fetch library that takes as input a resource URL or query and returns a Linked Data Object would prevent a developer from needing to understand concepts like BaseIRI or various RDF content-types. Further work could even integrate with popular JavaScript libraries, like React, to make the transition to using RDF even more seamless.

Spec Compliance. As mentioned in the "Building from the Schema" section, LDO interprets the feature of a JSON-LD context in an uncompliant manner. This is because the needs for LDO do not directly map to the needs for JSON-LD. In the future, LDO should generate its own proprietary context to be used at runtime and use a JSON-LD context only for conversions between JSON-LD and other RDF serializations.

LDO's build script also only supports a subset of ShEx's features, and future work must be done to support all features in ShEx as well as all features of OWL and SHACL.

Shape Evaluation. At the moment, LDO takes for granted that a certain subject follows a given shape. This can lead to mistakes and uncompliant data. But, LDO has the potential to also be a tool to evaluate the compliance of data by running data through a shape validator.

Maintenance and Bug Fixes. Currently the NLNet foundation provides funding for continued maintenance and new features for LDO. At the time of writing, NLNet funding continues through November of 2023 at which point a proposal to renew funding will

be submitted. A new funding source and maintenance plan should be found if NLNet does not renew. Given funding is not secured to maintain LDO, Jackson Morgan will maintain LDO on a volunteer basis.

7 Conclusion

Linked Data Objects (LDO) is designed to make manipulating RDF as similar to manipulating TypeScript as possible. In doing so, we've designed an experience that is more approachable for JavaScript developers. User feedback shows that LDO was successful in building an approachable developer experience for manipulating linked data, but future work can be done to make the initial setup of the devtool more approachable. Given the ongoing use of LDO in real-world projects, it has promise to be a useful tool for RDF novices and experts alike.

Acknowledgements. This project was funded through the NGI0 Entrust Fund, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program, under the aegis of DG Communications Networks, Content and Technology under grant agreement No 101069594.

Resource Availability Statement: Source code for LDO and its dependencies: ldo (https://purl. archive.org/o.team/ldo), ldo-cli (https://purl.archive.org/o.team/ldo-cli), shexj2 typeandcontext (https://purl.archive.org/o.team/shexj2typeandcontext), jsonld-dataset-proxy (https://purl.archive.org/o.team/jsonld-dataset-proxy), o-dataset-pack (https://purl.archive.org/o.team/o-dataset-pack).

References

1. RDFJS. https://rdf.js.org/. Accessed 12 July 2022
2. ClownFace Documentation. https://zazuko.github.io/clownface/#/. Accessed 12 July 2022
3. Tripledoc Documentation. https://vincenttunru.gitlab.io/tripledoc/. Accessed 12 July 2022
4. rdflib source code. https://github.com/linkeddata/rdflib.js/. Accessed 12 July 2022
5. JSON-LD. https://json-ld.org/. Accessed 12 July 2022
6. Verborgh, R., Taelman, R.: LDflex: a read/write linked data abstraction for front-end web developers. In: Pan, J.Z., et al. (eds.) ISWC 2020. LNCS, vol. 12507, pp. 193–211. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62466-8_13
7. RDF Object source code. https://github.com/rubensworks/rdf-object.js#readme. Accessed 12 July 2022
8. SimpleRDF source code. https://github.com/simplerdf/simplerdf. Accessed 12 July 2022
9. RDF Tools source ode. https://github.com/knowledge-express/rdf-tools#readme. Accessed 12 July 2022
10. Shex Methods documentation. https://ludwigschubi.github.io/shex-methods/. Accessed 12 July 2022
11. Semantika source code. https://github.com/dharmax/semantika#readme. Accessed 1 July 2022
12. ts-rdf-mapper source code. https://github.com/artonio/ts-rdf-mapper. Accessed 12 July 2022
13. Fischer, L., Hanenberg, S.: An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS visual studio. In: SIGPLAN 2015, vol. 51, pp. 154–167. ACM Digital Library (2016)

14. Endrikat, S., Hanenberg, S., Robbes, R., Stefik, A.: How do API documentation and static typing affect API usability? In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pp. 632–642. Association for Computing Machinery (2014)
15. ShEx – Shape Expressions. http://shex.io/. Accessed 02 May 2023
16. Shape Constraint Language (SHACL). https://www.w3.org/TR/shacl/. Accessed 02 May 2023
17. Rdfjs/N3.js. https://github.com/rdfjs/N3.js/. Accessed 02 May 2023
18. JSON-LD Streaming Parser. https://github.com/rubensworks/jsonld-streaming-parser.js. Accessed 02 May 2023/
19. RDF/JS: Dataset specification 1.0. https://rdf.js.org/dataset-spec/. Accessed 02 May 2023
20. Who says using RDF is hard? https://www.rubensworks.net/blog/2019/10/06/using-rdf-in-javascript/. Accessed 02 May 2023
21. Keet, C.M.: Open world assumption. In: Dubitzky, W., Wolkenhauer, O., Cho, K.H., Yokota, H. (eds.) Encyclopedia of Systems Biology, p. 1567. Springer, New York (2013). https://doi.org/10.1007/978-1-4419-9863-7_734
22. Graversen, K.B.: Method Chaining. https://web.archive.org/web/20110222112016/http://firstclassthoughts.co.uk/java/method_chaining.html. Accessed 02 May 2023
23. EMCAScript 2024 Language Specification. https://tc39.es/ecma262/multipage/reflection.html#sec-proxy-objects. Accessed 02 May 2023
24. JSON Schema. https://json-schema.org/. Accessed 02 May 2023
25. Morgan, J.: LDO source code. Zenodo (2023). https://doi.org/10.5281/zenodo.7909200
26. So(m)mer. https://github.com/bblfish/sommer. Accessed 15 July 2023
27. jenabean. https://code.google.com/archive/p/jenabean/. Accessed 15 July 2023
28. Owl2Java. https://github.com/piscisaureus/owl2java. Accessed 15 July 2023
29. JSON-LD Schema. https://github.com/mulesoft-labs/json-ld-schema. Accessed 15 July 2023
30. Solid. https://solidproject.org/. Accessed 15 July 2023