LDflex: a Read/Write Linked Data Abstraction for Front-End Web Developers

Ruben Verborgh and Ruben Taelman

IDLab, Department of Electronics and Information Systems,
Ghent University – imec, Ghent, Belgium
ruben.{verborgh,taelman}@ugent.be

Abstract. Many Web developers nowadays are trained to build applications with a user-facing browser front-end that obtains predictable data structures from a single, well-known back-end. Linked Data invalidates such assumptions, since data can combine several ontologies and span multiple servers with different APIs. Front-end developers, who specialize in creating end-user experiences rather than back-ends, thus need an abstraction layer to the Web of Data that integrates with existing frameworks. We have developed LDflex, a domain-specific language that exposes common Linked Data access patterns as reusable JavaScript expressions. In this article, we describe the design and embedding of the language, and discuss its daily usage within two companies. LDflex eliminates a dedicated data layer for common and straightforward data access patterns, without striving to be a replacement for more complex cases. The use cases indicate that designing a Linked Data developer experience—analogous to a user experience—is crucial for adoption by the target group, who in turn create Linked Data apps for end users. Crucially, simple abstractions require research to hide the underlying complexity.

1 Introduction

Other than in the beginning days of the Semantic Web, user-facing Web applications nowadays are often built by a dedicated group of specialists called *front-end developers*. This specialization resulted from an increasing maturity of the field of Web development, causing a divergence of skill sets among back-end and front-end developers, as well as different technologies and tool stacks. The current Semantic Web technology stack, in contrast, focuses mostly on back-end or full-stack developers, requiring an intimate knowledge about how data is structured and accessed. A dormant assumption is that *others* will build abstractions for front-end developers [29], whereas designing an adequate developer experience requires a deep understanding of Semantic Web technologies.

If we want front-end developers to build applications that read and write Linked Data, we need to speak their language and equip them with abstractions that fit their workflow and tooling [22]. Crucially, we do *not* see this as a matter of "dumbing down" SPARQL or RDF; rather, we believe it revolves around appropriate primitives for the abstraction level at which front-end applications are developed, similar to how SQL and tables are not front-end primitives either. The keyword is *proportionality* rather than convenience: the effort to access a certain piece of data should be justifiable in terms of its utility to the application.

Pre-print. To appear in Proceedings of the 19th International Semantic Web Conference

The difficulty lies in finding an abstraction that hides irrelevant RDF complexities, while still exposing the unbounded flexibility that Linked Data has to offer. Abstractions with rigid objects do not suffice, as their encapsulation tends to conceal precisely those advantages of RDF. Tools should instead enable developers to leverage the power and harness the challenges of the open Web. This empowerment is especially important in decentralized environments such as the Solid ecosystem [26], where data is spread across many sources that freely choose their data models. Building for such a multitude of sources is significantly more complex than interfacing with a single, controlled back-end [22].

This article discusses the design, implementation, and embedding of *LDflex*, a domain-specific language that exposes the Web of Linked Data through JavaScript expressions that appear familiar to developers. We discuss its requirements and formal semantics, and show how it integrates with existing front-end development frameworks. Rather than striving for full coverage of all query needs, LDflex focuses on simple but common cases that are not well covered by existing Semantic Web technologies (which remain appropriate for complex scenarios). We examine the usage of LDflex within two companies, and study its usage patterns within production code in order to assess its application in practice.

2 Related Work

Querying data on the Web SPARQL queries [9] carry *universal semantics*: each query maintains a well-defined meaning across data sources by using URIs rather than local identifiers, making queries independent of their processing. In theory, this enables reuse across different data sources; in practice, ontological differences need bridging [25, 29]. Although very few Web developers have experience with RDF or SPARQL, query-based development has been gaining popularity because of the GraphQL language [11]. While integrating well with existing development practices, GraphQL queries lack universal semantics, so applications remain restricted to specific sources. Several Semantic Web initiatives focused on providing *simpler* experiences. For example, EasierRDF [6] is a broad investigation into targeting the "average developer", whereas the concrete problem of simplifying query writing and result handling is tackled by SPARQL Transformer [13]. However, we argue that the actual need is not primarily simplification of complex cases, since many front-end developers have a sufficient background to learn RDF and SPARQL. The problem is rather a mismatch of abstraction level, because even conceptually simple data access patterns currently require a technology stack that is considered foreign.

Programming abstractions for RDF Programming experiences over RDF data generally fall in one of two categories: either a library offers generic interfaces that represent the RDF model (such as `Triple` and `Literal`), or a framework provides an abstraction inside of the application domain (such as `Person` or `BlogPost`). The latter can be realized through object-oriented wrappers, for instance via *Object–Triple Mapping* (OTM) [12], analogous to *Object-Relational Mapping* (ORM) for relational databases. However, whereas a table in a traditional database represents a *closed* object with a *rigid* structure, representations of RDF resources on the Web are *open* and can take *arbitrary* shapes. As such, there exists an *impedance mismatch* between the object-oriented and resource-oriented worlds [5]. Furthermore, local objects do not provide a good abstraction for distributed resources [30], which developers necessarily encounter when dealing with Linked Data on the Web.

Domain-specific languages for querying A Domain-Specific Language (DSL) is a programming language that, in contrast to general-purpose languages, focuses on a specific purpose or application domain, thereby trading generality for expressiveness [15]. A DSL is *external* if it has a custom syntax, whereas an *internal* DSL is embedded within the syntax of a host language [8]. For example, the scripting language *Ripple* is an external DSL for Linked Data queries [19]. Inside of another language, external DSLs are typically treated as text strings; for instance, a SPARQL query inside of the Java language would typically not be validated at compile time. Internal DSLs instead blend with the host language and reuse its infrastructure [8]. A prominent example of an internal query DSL is *ActiveRecord* within the Ruby language, which exposes application-level methods (such as `User.find_by_email`) through the *Proxy* pattern [16]. The *Gremlin* DSL [18] instead uses generic graph concepts for traversal in different database implementations.

JavaScript and its frameworks Since JavaScript can be used for both front-end and back-end Web application development, it caters to a large diversity of developers in terms of skills and tool stacks. A number of different frameworks exist for front-end development. The classic jQuery library [4] enables browser-agnostic JavaScript code for reading and modifying HTML's Document Object Model (DOM) via developer-friendly abstractions. Recently, frameworks such as *React* [17] have been gaining popularity for building browser-based applications. JavaScript is single-threaded; hence, costly I/O operations such as HTTP requests would block program execution if they were executed on the main thread. JavaScript realizes parallelism through *asynchronous* operations, in which the main thread delegates a task to a separate process and immediately resumes execution. When the process has finished the task, it notifies the JavaScript thread though a *callback function*. To simplify asynchronous code, the `Promise` class was recently introduced into the language, with the keywords `async`–`await` as syntactical sugar [14].

JavaScript and RDF Because of its ubiquitous embedding in browsers and several servers, the JavaScript programming language lends itself to reusing the same RDF code in server-side and browser-based Web apps. For compatibility, the majority of RDF libraries for JavaScript conform to API specifications [2] created by the W3C Community Group RDF/JS. The modular *Comunica* query engine [23] is one of them, providing SPARQL query processing over a federation of heterogeneous sources. JavaScript also gave birth to the *JavaScript Object Notation* (JSON), a widely used data format even in non-JavaScript environments. The JSON-LD format [20] allows adding universal semantics to JSON documents by mapping them to RDF. JSON-LD allows JSON terms to be interpreted as URIs using a given JSON-LD *context* that describes term-to-URI mappings. Since JSON-LD contexts can exist independently of JSON-LD documents, they can be reused for other purposes. For example, GraphQL-LD [24] leverages them to add universal semantics to GraphQL. The object-oriented abstractions *SimpleRDF*[1] and *RDF Object*[2] provide access to RDF data by applying JSON-LD contexts to regular objects. Instead of per-property access, *Soukai Solid*[3] considers entire RDF data shapes. All of the aforementioned abstractions require preconfiguring the context or shape and preloading an RDF graph in memory before data can be accessed in an object-oriented manner, limiting them to finite graphs.

[1] *https://github.com/simplerdf/simplerdf*
[2] *https://github.com/rubensworks/rdf-object.js*
[3] *https://github.com/NoelDeMartin/soukai-solid*

3 Requirements Analysis

This section lists the main requirements of LDflex for achieving the goal of a read/write Linked Data abstraction for front-end developers.

R1: Separates data and presentation Because of specialization and separation of concerns, front-end developers who work on the *presentation layer* typically should not come in contact with the data storage layer or its underlying database. Instead, they usually retrieve data through a *data access layer*, which is a higher-level abstraction over the storage layer that hides complexities of data storage that are irrelevant to front-end developers. For example, front-end developers could use a framework that exposes the *active record* architectural pattern instead of manually writing SQL queries for accessing relational databases. While SPARQL queries can abstract data access over a large variety of RDF interfaces, repeated SPARQL patterns in the presentation layer can become cumbersome to write. Specifically, we need a solution to capture repeated access to simple data patterns that occur frequently in typical front-end applications.

R2: Integrates into existing tooling The high tempo at which front-end Web development happens is only possible through specific workflows and tools used by front-end developers. Any solution needs to fit into these workflows, and provide compatibility with these tools. For instance, popular front-end frameworks such as React are based on composable, stateful components. In order for an abstraction layer to be useful, it must be able to integrate directly with such frameworks, without requiring further manual work.

R3: Incorporates the open world Since relational databases contain a finite number of data elements with a fixed schema, data access layers for relational databases can be *static* and based on a fixed set of properties and methods. In contrast, there is always more RDF data to be found for a given resource, and RDF data shapes can exist in various ontologies. Therefore, a data access layer for RDF must be *dynamic* so that it can handle arbitrarily shaped RDF data and various ontologies at runtime.

R4: Supports multiple remote sources In addition to interacting with local RDF data, it is crucial that an abstraction can also seamlessly access Linked Data from remote sources, preserving the semantics of data. Furthermore, RDF data for one resource can be spread over *multiple* sources across the Web, especially in decentralized scenarios, so a solution must consider this distribution and its consequences for application development.

R5: Uses Web standards It is required to interoperate with different modes of data access and data interfaces. This means that solutions have to be compatible with existing Web standards, such as RDF, SPARQL, and HTTP protocols and conventions regarding caching and authentication. Furthermore, since the solution will need to be deployed inside of browsers, it needs to be written in (or be able to be compiled to) languages and environments that are supported by modern browsers, such as JavaScript or WebAssembly, and corresponding browser APIs.

R6: Is configurable and extensible Since different applications have different data and behavioral demands, it must be possible to configure and extend the interpretation of the abstraction. On the one hand, developers must be able to configure the mapping from the data access layer to the RDF storage layer. On the other hand, developers must be able to customize existing features and to add new functionality, while controlling the correspondence with the storage layer.

4 Syntax and Semantics

Based on the above requirements, we have designed the LDflex DSL for JavaScript. We discuss related Web languages, and explain its syntactical design and formal interpretation.

4.1 Relation to Existing Languages

The LDflex language draws inspiration from existing path-based languages for the Web. The jQuery library [4] introduced a DSL for the traversal of HTML's Document Object Model, following the *Fluent Interface* pattern [7] with method chaining. For example, the expression `$('ol').children().find('a').text()` obtains the anchor text of the first hyperlink in an ordered list. This DSL is *internal*, as it is embedded within its host language JavaScript. Like Gremlin, it is implemented on the *meta*-level, with built-in methods such as `children` and `attr` referring to abstract HTML constructs (child nodes and attributes) rather than concrete cases (such as `<li>`). We call the evaluation of jQuery paths *safe* because supported methods are always defined—even if intermediary results are missing. For example, for elements without child nodes, calling `element.children().children()` will not produce a runtime error but rather yield an empty set.

The JSON-LD format is a subset of JSON, which itself is a subset of JavaScript. When a JSON-LD document is parsed into main memory during the execution of a JavaScript program, the resulting object can be traversed by chaining property accessors into a path. For instance, given a parsed JSON-LD document stored in a `person` variable, the expression `person.supervisor.department.label` could—depending on the object's JSON-LD context [20] and frame [21]—indicate the department label of a person's supervisor. Like jQuery paths, JSON-LD paths are valid JavaScript expressions and thus form an *embedded* DSL. In contrast to jQuery, JSON-LD paths use *data*-level constructs that refer to a concrete case (such as `supervisor` or `department`), rather than common metamodel concepts shared by all cases (such as `subject` or `predicate`). Evaluation is *unsafe*: syntactically valid paths might lead to runtime errors if an intermediate field is missing. For example, missing or incomplete data could lead the JSON-LD path segments `supervisor` or `department` to be `undefined` and therefore cause the evaluation to error.

4.2 Syntactical Design

Generic syntactical structure To achieve the requirements derived in Section 3, we combine the data-level approach of JSON-LD with the safe evaluation from jQuery, leveraging the Fluent Interface pattern [7]. LDflex adopts the syntax of JSON-LD paths consisting of consecutive property accesses, whose set of names is defined by a JSON-LD context [20] that also lists the prefixes. It follows the jQuery behavior that ensures each syntactically valid path within a given context results in an errorless evaluation. It provides extension points in the form of custom properties and methods to which arguments can be passed.

The grammar in Listing 1 expresses the syntax of an LDflex path in Backus–Naur form with start symbol ⟨path⟩. The terminal *root* is a JavaScript object provided by an LDflex implementation. *short-name* corresponds to JSON-LD *term*, *prefix* to JSON-LD *prefix*, *local-name* to JSON-LD *suffix* [20], and *arguments* is any valid JavaScript method arguments expression. Listing 2 displays examples of valid grammar productions.

⟨path⟩ ⊨ *root* ⟨segments⟩
⟨segments⟩ ⊨ ϵ | ⟨segment⟩ | ⟨segment⟩ ⟨segments⟩
⟨segment⟩ ⊨ ⟨property-access⟩ | ⟨method-call⟩
⟨property-access⟩ ⊨ **["** *full-URI* **"]** | **["** ⟨shorthand⟩ **"]** | **.** ⟨shorthand⟩
⟨shorthand⟩ ⊨ *prefix* **_** *local-name* | *short-name*
⟨method-call⟩ ⊨ *short-name* **(** *arguments* **)**

Listing 1. LDflex expressions follow a path-based syntax, detailed here in its Backus–Naur form.

```
1  const blog = data["https://alice.example/blog/"];
2  const comments = blog.blogPost.comment;
3  const blogAuthor = blog.foaf_maker.givenName;
4  displayItems(blog, comments, blogAuthor);
5
6  async function displayItems(topic, items, creator) {
7    console.log(`Items of ${await topic.name} at URL ${await topic}`);
8    console.log(`created by  ${await creator}:`);
9    for await (const item of items)
10     console.log(`- ${item}: ${await item.name}`);
11 }
```

Listing 2. In this code, LDflex paths are used to collect all comments on posts from a given blog. (This interpretation assumes that the Schema.org JSON-LD context and `foaf` prefix are set.)

In practice, several variations on this core grammar exist, leveraging the syntactical possibilities of JavaScript. For instance, an LDflex can be assigned to a variable, after which further segments can be added to that variable. In the remainder of this section, we focus on the core fragment of LDflex consisting of path expressions.

Usage as paths Multiple LDflex path syntax variations are displayed in lines 1 to 3 of Listing 2. Line 1 assumes the availability of a root object called `data`, on which we access a property whose name is a full URL. Since LDflex has safe evaluation, it guarantees that `blog` is not `undefined` for *any* arbitrary URL (the mechanism for which is explained in Section 5). Line 2 contains a continuation of the path from the previous line, using the `blogPost` and `comment` shorthands from the Schema.org JSON-LD context (assumed preset), which represent *http://schema.org/blogPost* and *http://schema.org/comment*, respectively. The prefix syntax is shown on Line 3, where `foaf_maker` represents *http://xmlns.com/foaf/0.1/maker* (assuming the `foaf` prefix has been preset). LDflex is JSON-LD-compatible, and thus also supports *compact IRIs* [20] in the familiar `foaf:maker` syntax. However, since property names containing colons need to be surrounded with brackets and quotes in JavaScript (`["foaf:maker"]`), we offer an alternative syntax that replaces the colon with either an underscore or dollar sign to bypass such escaping needs. Finally, those three paths are passed as arguments to a function call on line 4. Note in particular how these lines are syntactically indistinguishable from regular JavaScript code, even though they interact with Linked Data on the Web instead of local objects.

Resolution to a value On lines 1 to 4, LDflex expressions are treated purely as paths, which are created, extended, and passed around. An unresolved LDflex path *points* to values rather than representing those values itself. Obtaining the actual values involves one or more network requests, but since JavaScript is single-threaded, we cannot afford the mere creation of paths to consume such time. Instead, LDflex leverages a syntactical feature of JavaScript to explicitly trigger asynchronous resolution when needed: by placing the `await` keyword in front of an LDflex path, it resolves to the first value pointed to by the expression. Lines 7, 8 and 10 show this mechanism in action, where for instance `creator` (corresponding to `blog.foaf_maker.givenName`) is resolved to its value. Note how `topic` resolves to a URL because it points to a named node within the RDF model. We can retrieve a human-readable label for it by resolving the `list.name` path (line 7), which will resolve to its *http://schema.org/name* property. Importantly, in addition to representing values, resolved LDflex paths obtained using `await` still behave as regular LDflex paths to which additional segments can be added. This is exemplified on line 10, where the resolved path `item` is extended to `item.name` and in turn resolved via `await`.

Resolution to a series In several cases, resolving to a single value is preferred. For instance, even if multiple given names are specified for a person, displaying just one might be sufficient in a given context. In other contexts, all of its values might be needed. LDflex offers developers the choice between resolving to a singular value using `await`, or iterating over multiple values using the `for...await` syntactical construct. On line 9, the `items` path (corresponding to `blog.blogPost.comment`) is resolved asynchronously into a series of values. Every resulting `item` is a resolved LDflex path, that can be used as a raw value (`item` on line 10), or a regular LDflex path that is subsequently resolved (`await item.name` on line 10). In this example, the iteration variable `item` points to the URL of a comment, whereas `item.name` resolves to a human-readable label.

4.3 Formal Semantics

Similar to a JSON-LD document, the specific meaning of an LDflex expression is determined by the context in which it occurs. The resulting interpretation carries universal semantics. For LDflex expressions, the interpretation depends on the active LDflex *configuration* set by a domain expert, which includes settings such as:

- the definition and interpretation of the root path;
- the JSON-LD context used for resolving names into URIs;
- the definition of method calls and special properties.

In general, an LDflex expression consists of consecutive property accesses, representing a path from a known subject to an unknown object. For example, the path `data["https://alice.example/blog/"].blogPost.comment.name` formed by lines 1, 2 and 10 of Listing 2 corresponds to the SPARQL query displayed in Listing 3. This interpretation assumes a configuration that sets Schema.org as the JSON-LD context, and which interprets properties on the root node `data` as the URL of a subject resource. The configuration also expresses how data sources are selected. For instance, it might consider the document `https://alice.example/blog/` as an RDF graph or as a seed for link traversal [10], or might look up a query interface at `https://alice.example/`.

```
SELECT ?name WHERE {
  <https://alice.example/blog/> <http://schema.org/blogPost> ?post.
  ?post <http://schema.org/comment> ?comment.
  ?comment <http://schema.org/name> ?name.
}
```

Listing 3. The expression `data["https://alice.example/blog/"].blogPost.comment.name` of Listing 2 is interpreted as a SPARQL query expressing "titles of comments on posts of a given blog".

The query processing itself is handled entirely by an existing SPARQL query engine, and partial results can be cached and reused across LDflex paths for performance reasons.

We will now introduce a formal semantics for the set of LDflex expressions *LDf*. It is determined by an interpretation function $I_C\colon \mathit{LDf} \to Q \times S$, where Q is the set of SPARQL queries and S the set of source expressions over which SPARQL queries can be processed. Such an interpretation function can be instantiated by an LDflex configuration $C = \langle \mathit{ctx}, \mathit{root}, \mathit{other} \rangle$, where $\mathit{ctx}\colon \$ \to \mathcal{U}$ represents a JSON-LD context that maps JavaScript strings to RDF predicate URIs, and $\mathit{root}\colon \$ \to \mathcal{U} \times S$ a function that maps strings to a start subject URI and a source expression. The *other* set is reserved for other interpretation aspects, such as built-in properties or method names (not covered here).

We consider an LDflex expression $e \in \mathit{LDf}$ as a list consisting of a root property $r \in \$$ and $n > 0$ property accessors $k_i \in \$$, such that $e = (r, k_1, \ldots, k_n) \in \$^n$. The result of its interpretation $I_C(e) = \langle q, s \rangle$ is defined as follows. The root property r is resolved to a URI $u_r \in \mathcal{U}$ using $\mathit{root}(r) = \langle u_r, s_r \rangle$. Every property string k_i is resolved to a URI u_i using ctx, such that $\forall i \in [1,n]\colon u_i = \mathit{ctx}(k_i)$. Then, we generate a set of n triple patterns $\mathit{TP}_e \subset (\mathcal{U} \cup \mathcal{V}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{V})$, with $\mathit{tp}_i = \langle s_i, p_i, o_i \rangle \in \mathit{TP}_e$ conforming to the constraints:

- the subject of the first pattern is the root property's URI: $s_1 = u_r$
- the predicates correspond to the mapped JSON-LD properties: $\forall i \in [1,n]\colon p_i = u_i$
- the objects are unique variables: $\forall i, j \in [1,n]^2\colon o_i \in \mathcal{V} \wedge i \neq j \Rightarrow o_i \neq o_j$
- the objects and subjects form a chain of variables: $\forall i \in [2,n]\colon s_i = o_{i-1}$

These triple patterns form the SPARQL query q returned by I_C, which is a `SELECT` query that projects bindings of the basic graph pattern TP_e to the variable o_n. The second element s_r from the result of *root* determines the returned data source $s = s_r$ for query evaluation. The example in Listing 3 is obtained from its LDflex expression with $v \mapsto \langle v, v \rangle$ as *root*.

Importantly, this semantics ensures that the creation of any sequence of path segments always succeeds—even if no actual RDF triples exist for some intermediate predicate (which would cause an error with JSON-LD). This is because the LDflex expression represents a query, not a value. When the expression is prefixed with the `await` keyword, it resolves to an arbitrary RDF term t_i resulting from the evaluation of its underlying SPARQL query over the specified data sources: $\langle t_i \rangle \in [\![q]\!]_s$ (or `undefined` if there is none). With the `for...await` construct, all RDF terms $\langle \langle t_1 \rangle, \ldots, \langle t_m \rangle \rangle = [\![q]\!]_s$ are returned one by one in an iterative way.

Some LDflex configurations can have additional functionality, which is not covered in the general semantics above. For instance, a JSON-LD context can have *reverse properties*, for which the subject and objects of the corresponding triple pattern switch places.

4.4 Writing and Appending Data

The formal semantics above cover the case where an LDflex path is used for *reading* data. However, the same query generation mechanism can be invoked to execute SPARQL `UPDATE` queries. Since updates require filling out data in a query, they are modeled as *methods* such that arguments can be passed. The following methods are chainable on each LDflex path:

- `.add(...)` appends the specified objects to the triples matching the path.
- `.set(...)` removes any existing objects, and appends the specified objects.
- `.replace(old, ...new)` replaces an existing object with one or more new objects.
- `.delete(...)` removes the specified objects, or all objects (if none specified).

5 Implementation and Embedding

In this section, we discuss the implementation of LDflex within JavaScript to achieve the intended syntax and semantics as described in previous section. We first explain the main architecture, followed by an overview of the developed LDflex libraries.

5.1 Loosely-Coupled Architecture

Proxy Since the LDflex grammar allows for an *infinite* number of root paths and property URIs, we cannot implement them as a finite set of regular object methods. Instead, we make use of the more flexible Proxy pattern [16]. In JavaScript, `Proxy` allows customizing the behavior of the language by intercepting basic built-in constructs such as property lookup and function invocation. Intercepting every property access at one point is sufficient to define the behavior of the infinite number of possible properties.

Handlers and resolvers To achieve flexibility in terms of the functionality and logic that happens during LDflex expression evaluation, we make use of a loosely-coupled architecture of standalone *handlers* and *resolvers*. During the creation of the proxy-based LDflex path expression object, different handlers and resolvers can be configured, which allow the functionality of fields and methods on this path expression to be defined. Handlers are attached to a specific field or method name, which are used for implementing specifically-named functionality such as `.subject`, `.add()`, and `.sort()`. Resolvers are more generic, and are invoked on every field or method invocation if no handler was applicable, after which they can optionally override the functionality. For example, a specific resolver will translate field names into URIs using a configured JSON-LD context.

Implementing multiple interfaces Using the handlers and resolvers, the LDflex path expression behaves as an object with chainable properties. To allow path expressions to resolve to a *single value* using the `await` keyword, LDflex paths implement the JavaScript `Promise` interface through a handler. To additionally allow resolution to *multiple values*, LDflex paths also implement implement the `AsyncIterable` contract through another handler. Thanks to the `Proxy` functionality, an expression can thus simultaneously behave as an LDflex path, a `Promise`, and an `AsyncIterable`. The returned values implement the RDF/JS `Term` interface and thus behave as URIs, literals, or blank nodes. Furthermore, again by using `Proxy`, every value also behaves as a full LDflex path such

that continuations are possible. This complex behavior is exemplified in Listing 2, where the LDflex path `items` on line 9 is treated as an iterable with `for...await`, resulting in multiple `item` values. On line 10, those are first treated an RDF/JS `Term` (`item`), then as an LDflex path (`item.name`), and finally as a `Promise` (with `await item.name`).

Query execution To obtain result values, path expressions are first converted into SPARQL queries, after which they are executed by a SPARQL engine. By default, SPARQL queries for data retrieval are generated, as described in Section 4.3. When update handlers are used, SPARQL `UPDATE` queries are generated. This SPARQL query engine can be configured within the constructor of the path expression, which allows a loose coupling with SPARQL engines. Since LDflex passes SPARQL queries to existing query engines, it is not tied to any specific processing strategy. For instance, the engine could execute queries over SPARQL endpoints, in-memory RDF graphs, federations of multiple sources, or different query paradigms such as link-traversal-based query processing [10]. The performance of LDflex in terms of time and bandwidth is thus entirely determined by the query engine.

5.2 LDflex Libraries

Core libraries The JavaScript implementation of LDflex is available under the MIT license on GitHub at *https://github.com/LDflex/LDflex*, via the DOI `10.5281/zenodo.3820072`, and the persistent URL *https://doi.org/10.5281/zenodo.3820071*, and has an associated canonical citation [28]. The LDflex core is independent of a specific query engine, so we offer plugins to reuse the existing query engines *Comunica* (*https://github.com/LDflex/LDflex-Comunica*) and *rdflib.js* (*https://github.com/LDflex/LDflex-rdflib*) which enable full client-side query processing. Following best practices, LDflex and all of its related modules are available as packages compatible with Node and browsers on *npm* and are described following the FAIR principles as machine-readable Linked Data in RDF at *https://linkedsoftwaredependencies.org/bundles/npm/ldflex*. To make usage easy for newcomers, various documentation pages, examples, and tutorials created by ourselves and others are linked from the GitHub page. A live testing environment is at *https://solid.github.io/ldflex-playground/*. The sustainability plan includes a minimum of 3 years of maintenance by our team, funded by running projects related to Web querying.

Solid libraries An important application domain for LDflex is the Solid decentralized ecosystem [26]. In Solid, rather than storing their data collectively in a small number of centralized hubs, every person has their own *personal data vault*. Concretely, personal data such as profile details, pictures, and comments are stored separately for every person. Solid uses Linked Data in RDF, such that people can refer to each other's data, and to enable universal semantics across all data vaults without resorting to rigid data structures.

We created LDflex for Solid (available at *https://github.com/solid/query-ldflex/*) as an LDflex configuration that reuses a Solid-specific JSON-LD context containing shorthands for many predicates relevant to Solid. It is configured with custom handlers such as *like* and *dislike* actions. As certain data within Solid data pods requires authentication, this configuration includes a Comunica-based query engine that can perform authenticated HTTP requests against Solid data pods. It allows users to authenticate themselves to the query engine, after which the query engine will use their authentication token for any subsequent queries. Because of authentication, LDflex can be context-sensitive: within an expression such as `user.firstName`, `user` refers to the currently logged-in user.

The LDflex for Solid library is the basis for the Solid React Components (available at *https://github.com/solid/react-components/*), which are reusable software components for building React front-ends on top of Linked Data sources. These components can be used in React's DSL based on JavaScript and HTML to easily retrieve single and multiple values, as can be seen in Listing 4. Since these LDflex *micro-expressions* are regular strings, there is no specific coupling to the React framework. As such, LDflex can be reused analogously in other front-end frameworks.

```
<h2>Ruben's name</h2>
<Value src='["https://ruben.verborgh.org/profile/#me"].firstName'/>
<h2>Ruben's friends</h2>
<List src='["https://ruben.verborgh.org/profile/#me"].friends.firstName'/>
```

Listing 4. This example shows how React components, in this case `Value` and `List`, can use LDflex micro-expressions (highlighted) to retrieve Linked Data from the Web.

6 Usage and Validation

This section summarizes interviews we conducted[4] on the usage of LDflex within two companies. We evaluate the usage of LDflex within Janeiro Digital and Startin'blox by validating the requirements set out in Section 3.

6.1 Janeiro Digital

Janeiro Digital[5] is a business consultancy company in Boston, MA, USA that counts around 100 employees. They have worked in close collaboration with Inrupt[6], which was founded as a commercial driver behind the Solid initiative. They have developed the Solid React Software Development Kit (SDK), a toolkit for developing high-quality Solid apps without requiring significant knowledge on decentralization or Linked Data.

The employees within Janeiro Digital have a mixed technology background; several of them are dedicated front-end developers. Janeiro Digital makes use of LDflex as the primary data retrieval and manipulation library within the Solid React SDK. LDflex was chosen as it was a less verbose alternative to existing RDF libraries such as rdflib.js. Since most developers had never used RDF or SPARQL before, rdflib.js was very difficult to work with due to the direct contact with RDF triples. Furthermore, front-end developers would have to write SPARQL queries, while they were used to abstraction layers for such purposes. Since LDflex offers an abstraction layer over RDF triples and SPARQL queries, and makes data look like JavaScript objects, it proved to be easier to learn and work with.

The Solid React SDK provides several React components and code generators, which heavily make use of LDflex to meet simple data retrieval and manipulation needs. Because of LDflex, Janeiro Digital has been able to eliminate their previous dependency on the RDF library rdflib.js for building interactive applications over distributed Linked Data. Below, we briefly discuss three representative usages of LDflex within the SDK.

[4] The unabridged interview text is at *https://ruben.verborgh.org/iswc2020/ldflex/interviews/*.

[5] *https://www.janeirodigital.com/*

[6] *https://inrupt.com/*

```
const folder = data['http://example.org/myfolder'];
const paths = [];
for await (const path of folder['ldp:contains']) {
  paths.push(path.value);
}
```

Listing 5. Solid React SDK logic for collecting resources within a container.

```
await user.vcard_hasPhoto.set(namedNode(uri));
```

Listing 6. Solid React SDK logic for adding or changing a profile image.

Collecting Files in a Folder Listing 5 shows how LDflex `for...await` loops are being used to iterate over all resources within a container in a Linked Data Platform interface.

Saving Profile Photos Listing 6 shows the code that allows users to change their profile picture using the `.set()` method.

Manipulating Access Control for Files Listing 7 shows how Solid's Web Access Control authorizations for resource access can be manipulated using LDflex. In this case, a new `acl:Authorization` is created for a certain document.

The LDflex usage within the Solid React SDK shows a successful implementation of our requirements. Since Janeiro Digital deliberately chose LDflex due to its abstraction layer over RDF and SPARQL, it *separates data and presentation (R1)*. As LDflex can be used within React applications, even in combination with other RDF libraries such as rdflib.js, it achieves the requirement that it *integrates into existing tooling (R2)*. Next, LDflex *incorporates the open world (R3)* because it allows the SDK to make use of any ontology they need. Since LDflex *uses Web standards (R5)*, the SDK can run in client-side Web applications. Furthermore, the SDK can directly interact with any Solid data pod, and even combine multiple of them, which verifies the requirement that it *supports multiple remote sources (R4)*. The Solid configuration of LDflex discussed in Section 5 is used within the SDK, which shows that LDflex *is configurable and extensible (R6)*.

```
const { acl, foaf } = ACL_PREFIXES;
const subject = `${this.aclUri}#${modes.join('')}`;
await data[subject].type.add(namedNode(`${acl}Authorization`));
const path = namedNode(this.documentUri);
await data[subject]['acl:accessTo'].add(path);
await data[subject]['acl:default'].add(path);
```

Listing 7. Solid React SDK logic for authorizing access to a certain document.

6.2 Startin' Blox

Startin'blox[7] (SiB) is a company in Paris, France with a team of 25 freelancers. They develop the developer-friendly SiB framework with Web components that can fetch

[7] *https://startinblox.com/*

```
<sib-display
    data-src="data/list/users.jsonld"
    fields="username, first_name, last_name, email, profile.city"
></sib-display>
```

Listing 8. An SiB component for displaying the given fields of a list of users.

```
data["data/list/users.jsonld"].username
data["data/list/users.jsonld"].first_name
data["data/list/users.jsonld"].last_name
data["data/list/users.jsonld"].email
data["data/list/users.jsonld"].profile.city
```

Listing 9. All LDflex expressions that are produced in the SiB component from Listing 8.

data from Solid data vaults. Usage of SiB happens within the Happy Dev network[8] (a decentralized cooperative for self-employed developers), the European Trade Union Confederation, the International Cooperative Alliance, Smart Coop, and Signons.fr.

The Startin'blox team has a background in Web development, and assembled to support the creation of Solid applications. LDflex was chosen as an internal library for accessing Solid data pods, as opposed to directly writing SPARQL queries for data access, since they consider SPARQL too complex to learn for new developers, and they do not have a need for the full expressiveness that SPARQL has to offer. Most developers had no direct experience with RDF directly, but they knew JSON, which lowered the entry-barrier.

The SiB framework offers Web components in which developers can define source URIs and the fields that need to be retrieved from them, as shown in Listing 8. An LDflex expression will then be produced for each field, as shown in Listing 9. This example is representative for LDflex usage within SiB, where the majority of expressions select just a single property, and some expressions containing a chain of two properties. The LDflex engine can optimize internally such that, for instance, the document is only fetched once.

The usage of LDflex within SiB shows that LDflex meets all of our introduced requirements. As SiB component users only need to define a data source and a set of fields, the data storage layer is fully abstracted for them, which means it *separates data and presentation (R1)*. Furthermore, the integration of LDflex within the SiB components exemplifies how it *integrates into existing tooling (R2)*. Next, any kind of field can be defined within SiB components without this field having to preconfigured, which shows that LDflex *incorporates the open world (R3)*. SiB is a client-side framework, and it works over Linked Data-based Solid data pods over HTTP, which shows how LDflex *uses Web standards (R5)*. Some SiB users—such as the Happy Dev network—access data that is spread over multiple remote documents, and LDflex *supports multiple remote sources (R4)*. Finally, SiB is able to configure its own JSON-LD context. Some of their specific needs, such as the ability to handle pagination, and support for language-based data retrieval, can be implemented and configured as custom hooks into LDflex, which validates that LDflex *is configurable and extensible (R6)* for their purposes.

[8] *https://happy-dev.fr/*

7 Conclusion

Most Web developers do not care about Semantic Web technologies—and understandably so: the typical problems they tackle are of a less complex nature than what RDF and SPARQL were designed for. When a front-end is built to match a single, well-known back-end, nothing beats the simplicity of JSON and perhaps GraphQL, despite their lack of universal semantics. This, however, changes when accessing *multiple* back-ends—perhaps simultaneously—without imposing central agreement on all data models. Reusing the RDF technology stack might make more sense than reinventing the wheel, which unfortunately has already started happening within, for instance, the GraphQL community [1].

The Semantic Web definitely has a *user experience* problem, and if the rest of the Web can serve as a reliable predictor, neither researchers nor engineers will be the ones solving it. Front-end Web developers possess a unique skill set for translating raw data quickly into attractive applications. They can build engaging end-user interfaces to the Semantic Web, if we can provide them with the right *developer experience* by packaging RDF technology into a relevant abstraction layer. This requires an understanding of what the actual gaps are, and those look different than what is often assumed. During the design of LDflex, we have interacted with several front-end developers. All of them had a sufficiently technical profile to master RDF and SPARQL—and some of them even did. So there is no inherent need to simplify RDF or SPARQL. The point is rather that, in several common cases, those technologies are simply not the right tools for the job at hand.

LDflex is designed to support the adoption of Semantic Web technologies by front-end developers, who can in turn improve the experience and hence adoption for end-users. Rather than aiming to simplify everything, we want to ensure that straightforward tasks require proportionally sized code. For example, greeting the user by their first name is perfectly possible by fetching an RDF document, executing a SPARQL query, and interpreting the results. That code could even be abstracted into a function. However, the fact this code needs to be written in the first place, makes building user-friendly applications more involved. LDflex reduces such tasks to a single expression, removing a burden for building more engaging apps. The LDflex abstraction layer essentially acts as a runtime-generated data layer, such that a lot of glue code can be omitted. In fact, we witnessed at Janeiro Digital how several helper functions were eliminated by LDflex.

Importantly, LDflex purposely does not strive to provide an all-encompassing tool. The path queries that LDflex focuses on do not cover—by far—the entire spectrum of relevant application queries. While the evaluation shows that path queries are applicable to many common scenarios, more expressive languages such as GraphQL-LD or SPARQL remain appropriate for the remaining cases. LDflex rather aims to fulfill the *Rule of Least Power* [3], so developers can choose the expressivity that fits their problem space. Because of its high degree of extensibility, it can be adapted to different use cases via new, existing, or partly reused configurations.

Thereby, in addition to verifying whether our design requirements were met, the evaluation also brings insights into the technological needs of applications. Crucially, many SPARQL benchmarks focus on complex queries with challenging basic graph patterns, whereas some front-end patterns might actually generate rather simple queries—but a tremendously high volume of them. Furthermore, these queries are processed on the public Web, which is sensitive to latency. Typical scientific experiments are not tuned to

such contexts and constraints, so the currently delivered performance might lag behind. This makes it clear that delivering simple abstractions is not necessarily a simple task. On the contrary, exposing complex data through a simple interface involves automating the underlying complexity currently residing in handwritten code [29]. Doing so efficiently requires further research into handling the variety and distribution of data on the Web.

Since Solid presents prominent use cases for LDflex, future work will also need to examine how expressions can be distributed across different sources. For example, an expression such as `user.friends.email` could retrieve the list of friends from the user's data vault, whereas the e-mail addresses themselves could originate from the data vault of each friend (to ensure the recency of the data). Technically, nothing stops us from already doing this today: we could process the corresponding SPARQL query with a link-traversal-based query algorithm [10], which would yield those results. However, the actual problem is rather related to *trust*: when obtaining data for display to a user, which parts should come from which sources? A possible solution is *constrained traversal* [27], in which users can explain what sources they trust for what kinds of data.

One of the enlightening experiences of the past couple of months was that, during browser application development, we found ourselves also using LDflex—despite being well-versed in RDF and SPARQL. This is what opened our eyes to write this article: the reason we sometimes preferred LDflex is because it expressed a given application need in a straightforward way. We surely could have tackled every single need with SPARQL, but were more productive if we did not. This led to perhaps the most crucial insight: enabling developers means enabling ourselves.

Acknowledgements

The authors wish to thank Tim Berners-Lee for his suggestion to build a "jQuery for RDF." We thank James Martin and Justin Bingham from Janeiro Digital and Sylvain Le Bon and Matthieu Fesselier from Startin'blox for their participation in the LDflex interviews. This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" program.

References

1. Baxley, III, J.: Apollo Federation – a revolutionary architecture for building a distributed graph (May 2019), *https://blog.apollographql.com/apollo-federation-f260cf525d21*
2. Bergwinkl, T., Luggen, M., elf Pavlik, Regalia, B., Savastano, P., Verborgh, R.: RDF/JS: Data model specification. Draft community group report, W3C (Sep 2019), *https://rdf.js.org/data-model-spec/*
3. Berners-Lee, T., Mendelsohn, N.: The rule of least power. TAG finding, W3C Technical Architecture Group (Feb 2016), *https://www.w3.org/2001/tag/doc/leastPower.html*
4. Bibeault, B., Kats, Y.: jQuery in Action. Manning (2008)
5. Champin, P.A.: RDF-REST: a unifying framework for Web APIs and Linked Data. In: Proceedings of the First Workshop on Services and Applications over Linked APIs and Data (2013)
6. EasierRDF, *https://github.com/w3c/EasierRDF*
7. Fowler, M.: FluentInterface (2005), *https://www.martinfowler.com/bliki/FluentInterface.html*

8. Günther, S.: Development of internal domain-specific languages: Design principles and design patterns. In: Proceedings of the 18th Conference on Pattern Languages of Programs. pp. 1:1–1:25. ACM (2011)
9. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 query language. Recommendation, W3C (Mar 2013), *https://www.w3.org/TR/2013/REC-sparql11-query-20130321/*
10. Hartig, O.: An overview on execution strategies for Linked Data queries. Datenbank-Spektrum **13**(2), 89–99 (2013)
11. Hartig, O., Pérez, J.: Semantics and complexity of GraphQL. In: Proceedings of the 27th World Wide Web Conference. pp. 1155–1164 (2018)
12. Ledvinka, M., Křemen, P.: A comparison of object–triple mapping libraries. Semantic Web Journal (2019)
13. Lisena, P., Meroño-Peñuela, A., Kuhn, T., Troncy, R.: Easy Web API development with SPARQL transformer. In: Proceedings of the 18th International Semantic Web Conference (2019)
14. Loring, M.C., Marron, M., Leijen, D.: Semantics of asynchronous JavaScript. In: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages (2017)
15. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys **37**(4), 316–344 (Dec 2005)
16. Peck, M.M., Bouraqadi, N., Fabresse, L., Denker, M., Teruel, C.: Ghost: A uniform and general-purpose proxy implementation. Science of Computer Programming **98** (2015)
17. React: Facebook's functional turn on writing JavaScript. Communications of the ACM **59**(12), 56–62 (Dec 2016)
18. Rodriguez, M.A.: The Gremlin graph traversal machine and language. In: Proceedings of the 15th Symposium on Database Programming Languages. pp. 1–10. ACM (2015)
19. Shinavier, J.: Ripple: Functional programs as Linked Data. In: Proceedings of the Workshop on Scripting for the Semantic Web (2007), *http://ceur-ws.org/Vol-248/*
20. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: JSON-LD 1.0. Recommendation, W3C (Jan 2014), *http://www.w3.org/TR/json-ld/*
21. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: JSON-LD 1.1 framing. Working draft, W3C (Nov 2019), *https://www.w3.org/TR/json-ld11-framing/*
22. Staab, S., Scheglmann, S., Leinberger, M., Gottron, T.: Programming the Semantic Web. In: Proceedings of the European Semantic Web Conference. pp. 1–5 (2014)
23. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular SPARQL query engine for the Web. In: Proceedings of the 17th International Semantic Web Conference (Oct 2018), *https://comunica.github.io/Article-ISWC2018-Resource/*
24. Taelman, R., Vander Sande, M., Verborgh, R.: GraphQL-LD: Linked Data querying with GraphQL. In: Proceedings of the 17th International Semantic Web Conference: Posters and Demos (Oct 2018), *https://comunica.github.io/Article-ISWC2018-Demo-GraphQlLD/*
25. Verborgh, R.: Piecing the puzzle – self-publishing queryable research data on the Web. In: Proceedings of the 10th Workshop on Linked Data on the Web. vol. 1809 (Apr 2017)
26. Verborgh, R.: Re-decentralizing the Web, for good this time. In: Seneviratne, O., Hendler, J. (eds.) Linking the World's Information: Tim Berners-Lee's Invention of the World Wide Web. ACM (2020), *https://ruben.verborgh.org/articles/redecentralizing-the-web/*
27. Verborgh, R., Taelman, R.: Guided link-traversal-based query processing (2020), *https://arxiv.org/abs/2005.02239*
28. Verborgh, R., Taelman, R., Van Herwegen, J.: LDflex – A JavaScript DSL for querying Linked Data on the Web. Zenodo (May 2020). https://doi.org/10.5281/zenodo.3820071
29. Verborgh, R., Vander Sande, M.: The Semantic Web identity crisis: in search of the trivialities that never were. Semantic Web Journal **11**(1), 19–27 (Jan 2020)
30. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. Tech. Rep. TR-94-29, Sun Microsystems Laboratories, Inc. (Nov 1994)