LDkit: Linked Data Object Graph Mapping Toolkit for Web Applications

Karel Klíma[1(✉)], Ruben Taelman[2], and Martin Nečaský[1]

[1] Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czechia
{karel.klima,martin.necasky}@matfyz.cuni.cz

[2] IDLab, Department of Electronics and Information Systems, Ghent University – IMEC, Ghent, Belgium
ruben.taelman@ugent.be

Abstract. The adoption of Semantic Web and Linked Data technologies in web application development has been hindered by the complexity of numerous standards, such as RDF and SPARQL, as well as the challenges associated with querying data from distributed sources and a variety of interfaces. Understandably, web developers often prefer traditional solutions based on relational or document databases due to the higher level of data predictability and superior developer experience. To address these issues, we present LDkit, a novel Object Graph Mapping (OGM) framework for TypeScript designed to provide a model-based abstraction for RDF. LDkit facilitates the direct utilization of Linked Data in web applications, effectively working as the data access layer. It accomplishes this by querying and retrieving data, and transforming it into TypeScript primitives according to user defined data schemas, while ensuring end-to-end data type safety. This paper describes the design and implementation of LDkit, highlighting its ability to simplify the integration of Semantic Web technologies into web applications, while adhering to both general web standards and Linked Data specific standards. Furthermore, we discuss how LDkit framework has been designed to integrate seamlessly with popular web application technologies that developers are already familiar with. This approach promotes ease of adoption, allowing developers to harness the power of Linked Data without disrupting their current workflows. Through the provision of an efficient and intuitive toolkit, LDkit aims to enhance the web ecosystem by promoting the widespread adoption of Linked Data and Semantic Web technologies.

Keywords: Linked Data · Developer experience · Data abstraction

1 Introduction

The Semantic Web and Linked Data have emerged as powerful technologies to enrich the World Wide Web with structured and interlinked information [3]. Despite their potential, the adoption of these technologies by web application

T. R. Payne et al. (Eds.): ISWC 2023, LNCS 14266, pp. 194–210, 2023.
https://doi.org/10.1007/978-3-031-47243-5_11

developers has been hindered the challenging nature of querying distributed Linked Data in web applications [4], referred to as the expressivity/complexity trade-off.

Expressivity in Linked Data typically pertains to the ability to represent rich semantics and relationships between resources, often using ontologies and vocabularies [11]. The more expressive the data model, the more accurately and precisely it can represent the intended meaning and relationships between entities.

On the other hand, complexity refers to the difficulty in querying, processing, and managing the data. As the expressiveness of the Linked Data model increases, so does the complexity of the underlying query language, such as SPARQL [9], and the processing algorithms required to handle the data. This can lead to increased development effort, computational resources, and time needed to work with the data.

In order to leverage Linked Data in web applications, developers need to overcome these challenges. In recent years, several projects have been developed that address these needs, most prominently the Comunica [19] query engine, and LDflex [23], a domain-specific language for querying and manipulating RDF. These tools abstract away some of the complexity of Linked Data by simplifying querying mechanisms, without sacrificing expressiveness. Nevertheless, the Linked Data tooling ecosystem for web application development remains limited, as the tools and libraries available today may not be as mature or feature-rich as those for more established web development technologies.

In the past decade, the landscape of web development has undergone significant changes, with web technologies maturing considerably. The emergence of powerful web application frameworks, such as React[1] and Angular[2], has led to a rapid increase in the development and deployment of front-end web applications. The rise of TypeScript[3], a strongly typed programming language that builds on JavaScript, has brought a variety of benefits to the whole web ecosystem. The addition of a static typing system allows for strict type checking at compile-time, leading to improved code quality, enhanced developer productivity, and better tooling support. These technologies have provided developers with robust tools, enabling them to create sophisticated and feature-rich web applications more efficiently than ever before.

The growing complexity of web applications has highlighted the need for new types of data abstractions. Traditional data access patterns may no longer be sufficient to address the unique challenges posed by modern web development, such as the architecture split between back-end and front-end systems, or integrating with diverse APIs.

To address the needs of modern web developers, we introduce LDkit, a novel Linked Data abstraction designed to provide a type-safe and developer-friendly way for interacting with Linked Data from within web applications.

[1] https://react.dev/.
[2] https://angular.io/.
[3] https://www.typescriptlang.org/.

LDkit enables developers to directly utilize Linked Data in their web applications by providing mapping from Linked Data to simple, well-defined data objects; it shields the developer from the challenges of querying, fetching and processing RDF data.

In this paper, we present the design and implementation of LDkit, highlighting its ability to simplify the integration of Semantic Web technologies into web applications and improve the overall developer experience. By providing an efficient and intuitive toolkit, LDkit aims to promote the widespread adoption of Linked Data and Semantic Web technologies in web applications.

The rest of the paper is organized as follows. Section 2 introduces the related work, and Sect. 3 discusses requirements for LDkit as a viable Linked Data abstraction. Section 4 provides overview of design, implementation and embedding of LDkit in web applications, followed by Sect. 5 that evaluates LDkit from three distinct perspectives, including a real-world usage of the framework. We conclude in Sect. 6 and identify directions for future research.

2 Related Work

2.1 Web Application Data Abstractions

There are various styles of abstractions over data sources to facilitate access to databases in web development. These abstractions often cater to different preferences and use cases.

Object-Relational Mapping (ORM) and Object-Document Mapping (ODM) abstractions map relational or document database entities to objects in the programming language, using a data schema. They provide a convenient way to interact with the database using familiar object-oriented paradigms, and generally include built-in type casting, validation and query building out of the box. Examples of ORM and ODM libraries for JavaScript/TypeScript include *Prisma*[4], *TypeORM*[5] or *Mongoose*[6]. Corresponding tools for graph databases are typically referred to as Object-Graph Mapping (OGM) or Object-Triple Mapping (OTM) [14] libraries, and include *Neo4j OGM*[7] for Java and *GQLAlchemy*[8] for Python.

Query Builders provide a fluent interface for constructing queries in the programming language, with support for various database types. They often focus on providing a more flexible and composable way to build queries compared to ORM/ODM abstractions, but lack convenient development features like automated type casting. A prominent query builder for SQL databases in web application domain is *Knex.js*[9].

[4] https://www.prisma.io/.
[5] https://typeorm.io/.
[6] https://mongoosejs.com/.
[7] https://github.com/neo4j/neo4j-ogm.
[8] https://github.com/memgraph/gqlalchemy.
[9] https://knexjs.org/.

Driver-based abstractions provide a thin layer over the database-specific drivers, offering a simplified and more convenient interface for interacting with the database. An examples of a driver-based abstraction heavily utilized in web applications is the *MongoDB Node.js Driver*[10].

Finally, *API-based Data Access* abstractions facilitate access to databases indirectly through APIs, such as RESTful or GraphQL APIs. They provide client-side libraries that make it easy to fetch and manipulate data exposed by the APIs. Examples of API-based data access libraries include *tRPC*[11] and *Apollo Client*[12].

Each style of abstraction caters to different needs and preferences, ultimately the choice of abstraction style depends on the project's specific requirements and architecture, as well as the database technology being used. There are however several shared qualities among these libraries that contribute to a good developer experience. All of these libraries have *static type support*, which is especially beneficial for large or complex projects, where maintaining consistent types can significantly improve developer efficiency. Another aspect is *good tooling support*: these libraries often provide integrations with popular development tools and environments. This support can include autocompletion, syntax highlighting, and inline error checking, which further enhances the developer experience and productivity. Furthermore, most of these libraries offer a consistent API across different database systems, which simplifies the process of switching between databases or working with multiple databases in a single application. Finally, abstracting away low-level details allows developers to focus on their application's logic rather than dealing with the intricacies of the underlying database technology.

2.2 JavaScript/TypeScript RDF Libraries

JavaScript is a versatile programming language that can be utilized for both front-end development in browsers and back-end development on servers. As Linked Data and RDF have gained traction in web development, several JavaScript libraries have emerged to work with RDF data. These libraries offer varying levels of RDF abstraction and cater to different use cases.

Most of the existing libraries conform to the RDF/JS Data model specification [2], sharing the same RDF data representation in JavaScript for great compatibility benefits. Often, RDF libraries make use of the JSON-LD (JavaScript Object Notation for Linked Data) [17], a lightweight syntax that enables JSON objects to be enhanced with RDF semantics. JSON-LD achieves this by introducing the concept of JSON-LD *context*, which serves as a dictionary that maps JSON keys to RDF property and type IRIs. This mapping allows for JSON objects to be interpreted as RDF graphs, and can also be used independently of JSON-LD documents.

[10] https://github.com/mongodb/node-mongodb-native.
[11] https://github.com/trpc/trpc.
[12] https://github.com/apollographql/apollo-client.

One of the most comprehensive projects is *Comunica* [19], a highly modular and flexible query engine for Linked Data, enabling developers to execute SPARQL queries over multiple heterogeneous data sources with extensive customizability.

LDflex [23] is a domain-specific language that provides a developer-friendly API for querying and manipulating RDF data with an expressive, JavaScript-like syntax. It makes use of JSON-LD contexts to interpret JavaScript expressions as SPARQL queries. While it does not provide end-to-end type safety, LDflex is one of the most versatile Linked Data abstractions that are available. Since it does not utilize a fixed data schema, it is especially useful for use cases where the underlying Linked Data is not well defined or known.

There are also several object-oriented abstractions that provide access to RDF data through JavaScript objects. *RDF Object*[13] and *SimpleRDF*[14] enable per-property access to RDF data through JSON-LD context mapping. *LDO (Linked Data Objects)*[15] leverage ShEx [15] data shapes to generate RDF to JavaScript interface, and static typings for the JavaScript objects. *Soukai-solid*[16] provides OGM-like access to Solid Pods[17] based on a proprietary data model format.

Except for LDflex, the major drawback of all the aforementioned Linked Data abstractions is that they require pre-loading the source RDF data to memory. For large decentralized environments like Solid, this pre-loading is often impossible, and we instead require discovery of data during query execution [22]. While these libraries offer valuable tools for working with RDF, when it comes to web application development, none of them provides the same level of type safety, tooling support and overall developer experience as their counterparts that target relational or document databases.

In recent years, the GraphQL[18] interface has gained popularity as an alternative to REST interfaces, due to its flexible data retrieval, strongly typed schema, and the ability to group multiple REST requests into one. A notable element of this interface is the GraphQL query language, which is popular among developers due to its ease of use and wide tooling support. However, GraphQL uses custom interface-specific schemas, which are difficult to federate over, and have no relation to the RDF data model.

That is why, in the recent years, we have seen several initiatives [1,20,21] attempting to bridge the worlds of GraphQL and RDF, by translating GraphQL queries into SPARQL, with the goal of lowering the entry-barrier for writing queries over RDF. While these initiatives addressed the problems to some extent, there are still several drawbacks to this approach. Most notably, it requires the deployment of a GraphQL server, which is not always possible or desirable,

[13] https://github.com/rubensworks/rdf-object.js.
[14] https://github.com/simplerdf/simplerdf.
[15] https://github.com/o-development/ldo.
[16] https://github.com/NoelDeMartin/soukai-solid.
[17] https://solidproject.org/.
[18] https://graphql.org/.

depending on the use case. Furthermore, this extra architectural layer may add significant performance overhead.

3 Requirements Analysis

The goal of LDkit is to provide a type-safe and developer-friendly abstraction layer for interacting with Linked Data from within web applications. Based on our research of common web data abstractions and Linked Data libraries, we have identified a set of primary requirements for LDkit that are necessary to achieve this goal.

R1 Embraces Linked Data heterogenity
The inherent heterogeneity of Linked Data ecosystem arises due to the decentralized nature of Linked Data, where various data sources, formats, and ontologies are independently created and maintained by different parties [4]. As a result, data from multiple sources can exhibit discrepancies in naming conventions, data models, and relationships among entities, making it difficult to combine and interpret the information seamlessly [12]. LDkit should embrace this heterogeneity by supporting the querying of Linked Data from multiple data sources and various formats.

R2 Provides a simple way of Linked Data model specification
The core of any ORM, ODM or OGM framework is a specification of a data model. This model is utilized for shielding the developer from the complexities of the underlying data representation. It is a developer-friendly programming interface for data querying and validation that encapsulates the complexity of the translation between the simplified application model and the underlying data representation. In LDkit, the data model should be *easy to create and maintain*, and separable from the rest of the application so that it can be *shared* as a standalone artifact. LDkit should aim to offer a comprehensive RDF data abstraction while simplifying the data structure by default, ensuring efficient use in web applications. Simultaneously, it must allow users to override the default behavior to fine-tune the processes of RDF data querying, retrieval, and processing.

R3 Has a flexible architecture
A Linked Data abstraction for web applications needs to encompass several inter-related processes, such as generating SPARQL queries based on the data model, executing queries across one or more data sources, and transforming RDF data to JavaScript primitives and vice-versa. In LDkit, each of these processes should be implemented as a standalone component for maximum *flexibility*. A flexible architecture allows LDkit to *adapt* to varying use cases and requirements, making it suitable for a wide range of web applications that leverage Linked Data. Developers can *customize* the framework to their specific needs, modify individual components, or extend the functionality to accommodate unique requirements. Finally, as Linked Data and web technologies evolve, a flexible architecture ensures that LDkit remains relevant

and can accommodate new standards, formats, or methodologies that may emerge in the *future*.

R4 Provides solid developer experience
LDkit can achieve a good developer experience by focusing on several key aspects. First, LDkit should provide a clear and intuitive API to make the learning curve more manageable for developers new to the framework. Second, the toolkit should leverage TypeScript's type safety features, enabling better tooling support and error prevention. This provides developers with instantaneous feedback in the form of autocomplete or error highlighting within their development environment. Third, LDkit must ensure compatibility with popular web application libraries and frameworks, allowing developers to incorporate LDkit into their existing workflows more easily. By focusing on these aspects, LDkit can create a positive developer experience that fosters rapid adoption and encourages the effective use of the framework for reading and writing Linked Data in web applications.

R5 Adheres to existing Web standards and best practices
LDkit should adhere to both general web standards and web development best practices, and Linked Data specific standards for several reasons. First, compliance with established standards ensures interoperability and seamless integration with existing web technologies, tools, and services, thereby enabling developers to build on the current web ecosystem's strengths. Second, adhering to Linked Data specific standards fosters best practices and encourages broader adoption of Linked Data technologies, contributing to a more robust and interconnected Semantic Web. Finally, compliance with existing web standards allows for the long-term sustainability and evolution of the LDkit framework, as it can adapt and grow with the ever-changing landscape of web technologies and standards.

4 LDkit

We have designed LDkit OGM library according to the aforementioned requirements. In this section, we provide a high level perspective of LDkit capabilities and discuss some of its most important components.

Let us illustrate how to display simple Linked Data in a web application, using the following objective:
Query DBpedia for Persons. A person should have a name property and a birth date property of type date. Find me a person by a specific IRI.

The example in Listing 1.1 demonstrates how to query, retrieve and display Linked Data in TypeScript using LDkit in only 20 lines of code.

On lines 4–11, the user creates a data *Schema*, which describes the shape of data to be retrieved, including mapping to RDF properties and optionally their data type. On line 13, they create a *Lens* object, which acts as an intermediary between Linked Data and TypeScript paradigms. Finally, on line 18, the user requests a data artifact using its resource IRI and receives a plain JavaScript object that can then be printed in a type-safe way.

```
import { createLens } from "ldkit";
import { dbo, xsd } from "ldkit/namespaces";

const PersonSchema = {
  "@type": dbo.Person,
  "name": dbo.birthName,
  "birthDate": {
    "@id": dbo.birthDate,
    "@type": xsd.date,
  },
} as const;

const Persons = createLens(PersonSchema, {
  sources: ["https://dbpedia.org/sparql"]
});

const adaIri = "http://dbpedia.org/resource/Ada_Lovelace";
const ada = await Persons.findByIri(adaIri);

console.log(ada.name); // "The Hon. Augusta Ada Byron"
console.log(ada.birthDate); // Date object of 1815-12-10

```

Listing 1.1. LDkit usage example

Under the hood, LDkit performs the following:

- Generates a SPARQL query based on the data schema.
- Queries remote data sources and fetches RDF data.
- Converts RDF data to JavaScript plain objects and primitives.
- Infers TypeScript types for the provided data.

4.1 Data Schema

On the conceptual level, a data schema is a definition of a data shape through which one can query RDF data, similar to a data model for standard ORM libraries. Specifically, the schema describes a class of entities defined by RDF type(s) and properties.

We have designed the LDkit *schema* based on the following criteria:

- LDkit can generate SPARQL queries based on the schema.
- LDkit can use the schema as a mapping definition between RDF and JavaScript primitives (both ways).
- LDkit can infer a correct TypeScript type from the schema.
- Developer can adjust the resulting data shape; specifically, they can require some properties to be optional or arrays.
- Developer can nest schemas within other schemas.

– Developer can reuse and share schemas independently of LDkit.
– Schemas must be easy to create and should be self-explanatory.

A *schema* is a plain TypeScript object that follows the formal specification defined in Listing 1.2. LDkit schemas are based on JSON-LD context format, and simple JSON-LD contexts can be easily transformed to LDkit schemas. Having the schema defined in TypeScript allows for end-to-end data type safety. In addition, developers may benefit from autocomplete features and syntax checks within their development environment, to aid schema reuse and composition.

```
type Schema = {
  "@type": Iri | Iri[];
  [key: string]: Iri | Property;
}
type Property = {
  "@id": Iri;
  "@type"?: DatatypeIri;
  "@context"?: Schema;
  "@optional"?: true;
  "@array"?: true;
  "@multilang"?: true;
}
type Iri = string
type DatatypeIri = /* supported data type, e.g. xsd:date */
```

Listing 1.2. Formal specification of LDkit schema in TypeScript

4.2 Reading and Writing Data

In LDkit, reading and writing data is realized through *Lens*.

A data *Lens* turns a particular data *Schema* to an interactive entity repository. Conceptually, a Lens represents a collection of data *entities* that conform to the specified Schema. The interface of Lens follows the data mapper architectural pattern, facilitating bidirectional transfer of data between an RDF data store and in-memory data representation of *entities*, which are plain JavaScript objects. In background, Lens handle building and executing SPARQL queries, data retrieval and transformation according to the data Schema.

A Lens instance provides the following data access interface:

– **find([where], [limit]): entity**[] retrieves all entities that correspond to the data schema, optionally applies additional conditions and limit
– **findByIri(iri): entity** retrieves an entity with a particular resource IRI
– **count([where]): number** counts all entities that correspond to the data schema, optionally applies additional conditions
– **insert(...entities)** creates new entities in the data source

- **update(...entities)** updates entity properties in the data source
- **delete(...iris)** removes entities from data source based on their IRI.

When any of these methods are invoked, LDkit creates an appropriate SPARQL or SPARQL UPDATE [7] query and execute it against the underlying data source. Consequently, in order to modify data, the data source must permit update operations. The algorithm that generates the queries is complex, as it takes into account payload of the interface methods, as well as data schema, therefore its description is out of scope of this article.

The presented interface is similar to other data mapper-based TypeScript frameworks and covers all basic data reading and manipulation. Listing 1.3 demonstrates how this interface may be used to retrieve and update a data entity.

```
const Persons = createLens(PersonSchema);

const alanIri = "http://dbpedia.org/resource/Alan_Turing";
const alan = await Persons.findByIri(alanIri);

alan.name = "Not Alan Turing"; // fictitious name
alan.birthDate = new Date("1900-01-01"); // fictitious birth date
await Persons.update(alan);
```

Listing 1.3. Example of reading and writing data in LDkit

While the Lens interface is expressive enough to cover common cases of working with data that are structured in a relational fashion, it may be insufficient or inconvenient for some advanced use cases, such as working with large arrays possibly containing thousands of items. While LDkit supports reading such arrays, modifying them with operations like *insert a value to array* or *remove a value from array* may be cumbersome to perform through the standard interface. For these cases, the Lens exposes advanced methods for the developer to interact directly with RDF, either in the form of SPARQL query or RDF quads:

- **query(sparqlQuery)** retrieves entities based on a custom SPARQL query
- **updateQuery(sparqlQuery)** performs a SPARQL UPDATE query on the data source
- **insertData(quads[])** inserts RDF quads array to data source
- **deleteData(quads[])** removes RDF quads array from data source.

4.3 Data Sources and Query Engine

In LDkit, a *Query engine* is a component that handles execution of SPARQL queries over data sources. The query engine must follow the RDF/JS Query specification [18] and implement the `StringSparqlQueryable` interface.

LDkit ships with a simple default query engine that lets developers execute queries over a single SPARQL endpoint. It is lightweight and optimized for browser environment, and it can be used as a standalone component, independently of the rest of LDkit. The engine supports all SPARQL endpoints that conform to the SPARQL 1.1 specification [9].

LDkit is fully compatible with Comunica-based query engines. Comunica [19] provides access to RDF data from multiple sources and various source types, including Solid pods, RDF files, Triple/Quad Pattern Fragments, and HDT files.

4.4 Current Limitations

While the presented data model of LDkit reduces complexity of SPARQL and RDF, it introduces some trade-offs, as it is not as expressive. First, reverse relations are not yet supported. That could be useful for scenarios when one needs to display incoming links. In order to achieve this, the developer needs to provide a custom SPARQL query that would produce a graph corresponding to the specified schema. Second, there is the issue of multiplicity of RDF properties: contrary to the world of relational databases, where each cell in a table usually corresponds to a single value, the world of Linked Data does not have this constraint. As a result, there may be an unknown number of triples with the same RDF property linked to a particular resource. This may either be by design, if the data is supposed to represent a set of values, or the data may be of poor quality and there may be some duplicates. Ultimately, the developer needs to choose, whether they prefer to read one, albeit random, value, or an array of values that may be redundant.

4.5 LDkit Components

Thanks to its modular architecture, components comprising the LDkit OGM framework can be further extended or used separately, accommodating advanced use cases of leveraging Linked Data in web applications. Besides *Schema*, *Lens* and *Query engine* already presented, there are other components and utilities that can be used by developers to facilitate working with Linked Data. The `Decoder` and `Encoder` components transform data from RDF to JavaScript plain objects and vice-versa based on the provided data schema. The `QueryBuilder` generates SPARQL CRUD queries based on a data schema. Furthermore, there is a universal SPARQL query builder available, allowing for type-safe SPARQL query composition, and a set of utilities for working with RDF quads. Finally, LDkit also includes *Namespaces* definitions for popular Linked Data vocabularies, such as Dublin Core [5], FOAF [6] or Schema.org [16].

This level of flexibility means that LDkit could also support other query languages, such as GraphQL.

4.6 LDkit Distribution and Sustainability

The TypeScript implementation of LDkit is available under the `MIT` license on GitHub at https://github.com/karelklima/ldkit, via the *DOI 10.5281/zen-*

odo.7905468, and the persistent URL https://doi.org/10.5281/zenodo.7905468, and has an associated canonical citation [13].

Following the standard practices, LDkit is published as an NPM package[19] and as a Deno module[20]. To make adoption easy for new developers, documentation and examples are available at https://ldkit.io or linked from the GitHub repository.

In order to demonstrate our commitment to the long-term maintenance and improvement of LDkit, we have developed a comprehensive sustainability plan. Our team guarantees a minimum of five years of ongoing maintenance, during which we will be dedicated to addressing any issues, optimizing performance, and ensuring compatibility with the evolving Linked Data landscape. LDkit has already been adopted by several academic and non-academic projects, with more projects set to incorporate it in the future. This growing user base helps to guarantee ongoing interest and support for the framework. As LDkit continues to be used in new research projects, our team will work closely with the academic community to gather feedback and identify areas for further improvement. Finally, we have identified several features that are not yet included in LDkit but will enhance its capabilities and usefulness in the future. Our team will actively work on incorporating these features into LDkit, ensuring its continued relevance and applicability to a wide range of use cases. By implementing this sustainability plan, we aim to ensure that LDkit remains a valuable and dependable resource for web developers and researchers working with Linked Data, both now and in the years to come.

5 Evaluation

In this section, we present the evaluation of LDkit from three distinct perspectives to provide a comprehensive assessment of the framework's capabilities. First, we discuss the primary requirements that LDkit aims to address and how it satisfies these needs. Second, we demonstrate a real-world use case of LDkit to showcase its practical applicability in web applications. Finally, we examine the framework's performance.

5.1 Requirements Reflection

Earlier in this paper, we presented a list of five primary requirements that LDkit must meet in order to provide a developer-friendly abstraction layer for interacting with Linked Data from within web applications.

LDkit provides a simple way of Linked Data model specification (**R2**) through *schema*, which is a flexible mechanism for developers to define their own custom data models and RDF mappings that are best suited for their application's requirements. The schema syntax is based on JSON-LD context, and as such it

[19] https://www.npmjs.com/package/ldkit.
[20] https://deno.land/x/ldkit.

assumes its qualities: it is self-explanatory and easy to create, and can be reused, nested, and shared independently of LDkit.

Thanks to the flexible data model definition and interoperability with the Comunica query engine, LDkit effectively embraces Linked Data heterogenity (**R1**) by providing means to query Linked Data from various data sources, formats, and vocabularies. Furthermore, its modular architecture (**R3**) allows for a high level of customization. LDkit components can be adapted and extended to accommodate unique requirements, or used standalone for advanced use cases.

LDkit offers good developer experience (**R4**) in several ways. Its API for reading and writing Linked Data is simple and intuitive, and should feel familiar even to the developers new to RDF, as it is inspired by interfaces of analogous model-based abstractions of relational databases. By incorporating end-to-end data type safety, which is the biggest differentiator from LDflex, LDkit provides unmatched tooling support, giving developers instantaneous feedback in the form of autocomplete or error highlighting within their development environment. The official website of LDkit[21] contains comprehensive documentation, along with a step-by-step "getting started" guide for new developers, and includes examples of how to use LDkit with popular web application frameworks, such as React. These aspects contribute to a positive developer experience and encourage effective use of LDkit in web applications.

Finally, LDkit adheres to and employs existing Web and Linked Data standards (**R5**), such as JSON-LD or SPARQL, to ensure interoperability and seamless integration with existing Web technologies. LDkit follows RDF/JS data model [2] and query [18] standards, making it compatible with other existing Linked Data tools, such as Comunica, and contributing to a more robust Linked Data ecosystem for web developers.

5.2 Real World Usage

LDkit is used in a project for the Czech government[22] that aims to build a set of web applications for distributed modeling and maintenance of government ontologies[23]. The ensemble is called *Assembly Line (AL)*. It allows business glossary experts and conceptual modeling engineers from different public bodies to model their domains in the form of domain vocabularies consisting of a business glossary further extended to a formal UFO-based ontology [8]. The individual domain vocabularies are managed in a distributed fashion by the different parties through AL. AL also enables interlinking related domain vocabularies and also linking them to the common upper public government ontology defined centrally. Domain vocabularies are natively represented and published[24] in SKOS (business glossary) and OWL (ontology). The AL tools have to process this native

[21] https://ldkit.io/.
[22] https://slovnik.gov.cz.
[23] https://github.com/datagov-cz/sgov-assembly-line is the umbrella repository that refers to the repositories of individual tools (in Czech).
[24] https://github.com/datagov-cz/ssp (in Czech).

representation of the domain vocabularies in their front-end parts. Dealing with native representation would be, however, unnecessarily complex for the front-end developers of these tools. Therefore, they use LDkit to simplify their codebase. This allows them to focus on the UX of their domain-modeling front-end features while keeping the complexity of SKOS and OWL behind the LDkit schemas and lenses. On the other hand, the native SKOS and OWL representations of the domain models make their publishing, sharing, and reuse much easier. LDkit removes the necessity to transform this native representation with additional transformation steps in the back-end components of the AL tools.

5.3 Performance

To evaluate the performance of LDkit, we considered a typical use case of working with data in web applications: displaying a list of data resources. Specifically, we envisioned a scenario of building a Web book catalog. For this catalog, our objective was to obtain a list of books from the DBpedia[25] SPARQL endpoint so that we can display it to end users. We have designed three experiments, and in each case, we query DBpedia for a list of 1000 books, using the LDkit built-in query engine. The experiments are identical except for the data schema; its complexity increases with each test case. For reproducibility purposes, we have shared the experiments on GitHub[26].

Our initial assumption was that LDkit should not add significant performance overhead, and that the majority of the execution time would be spent on querying the SPARQL endpoint, since this is often the primary bottleneck when dealing with remote data sources.

To assess LDkit performance, we measured total execution time and subtracted the time it took to query DBpedia.[27] Table 1 displays the resulting aver-

Table 1. LDkit performance evaluation results

Scenario schema	Book – title	Book – title – author(Person) – name	Book – title – author(Person) – name – country – language – genre
Query time	255 ms	359 ms	2751 ms
LDkit time	23 ms	38 ms	45 ms
Total time	278 ms	397 ms	2796 ms
Number of quads	217	3955	7123

[25] https://dbpedia.org/sparql.

[26] https://github.com/karelklima/ldkit/tree/main/tests/performance.

[27] Each scenario was run 10 times, and was executed using Deno JavaScript runtime. The experiment was performed on a PC with 2.40 GHz Intel i7 CPU and 8 GB RAM.

age times for each scenario and, for illustration purposes, includes a number of RDF quads that were returned by the data source.

Our findings indicate that, even with the increasing complexity of the scenarios, LDkit maintained its performance without any substantial degradation. Since LDkit uses data schema to generate SPARQL queries that are eventually passed to a pre-configured query engine, the overall performance will be therefore determined mostly by the query engine itself, which may employ advanced strategies for query processing, such as query optimization or caching [10], leading to improved query execution times.

6 Conclusion

Web application development is a rather specific software engineering discipline. When designing a website, or any other user-facing application for that matter, the developers need to think about the product side and user interface first. In short, they need to figure out what to display to users and how. Building a great user interface and experience is the primary objective. Hence, the ever-evolving web application tooling provide sophisticated abstractions and enable developers to focus on what matters the most – the end user. Popular web application frameworks, such as React, employ declarative programming paradigm, and the use of visual components as the application building blocks. Modern data access solutions are seamlessly integrated to application frameworks, to allow for easy access to data, simplified to the application domain model, in a declarative way. In that regard, from the point of view of a front-end developer, the web application architecture, and even more so data architecture, are almost an afterthought.

In this paper, we presented LDkit, a developer-friendly tool that enables using Linked Data within web applications in an easy and intuitive way.

LDkit is the result of a decade-long effort and experience of building front-end web applications that leverage Linked Data, and as such it is a successor to many different RDF abstractions that we have built along the way. It is designed to cater to the mindset of a web developer and help them focus on data itself and how to best present them to the users, abstracting away the complexity of querying, processing and adapting Linked Data.

In our future work, we aim to further extend the capabilities of LDkit, and we plan to build more sophisticated solutions for assisted generating of LDkit schemas and entire front-end applications from RDF vocabularies or data sources, allowing for rapid prototyping of Linked Data-based applications.

In conclusion, we believe that LDkit is a valuable contribution to the Linked Data community, providing a powerful and accessible tool to seamlessly integrate Linked Data into web applications. Throughout this paper, we have presented evidence to support this claim, demonstrating how LDkit addresses specific web development needs and how it can be utilized in real-world scenarios. We are confident that LDkit will contribute to further adoption of Linked Data in web applications.

References

1. Angele, K., Meitinger, M., Bußjäger, M., Föhl, S., Fensel, A.: GraphSPARQL: a GraphiQL interface for linked data. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pp. 778–785 (2022)
2. Bergwinkl, T., Luggen, M., elf Pavlik, Regalia, B., Savastano, P., Verborgh, R.: RDF/JS: data model specification, May 2022. https://rdf.js.org/data-model-spec/
3. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Sci. Am. **284**(5), 34–43 (2001)
4. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semant. Web Inf. Syst. **5**(3), 1–22 (2009)
5. DCMI Usage Board: DCMI metadata terms (2020). https://www.dublincore.org/specifications/dublin-core/dcmi-terms/
6. Brickley, D., Miller, L.: FOAF vocabulary specification (2014). http://xmlns.com/foaf/spec/
7. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 update, March 2013. https://www.w3.org/TR/sparql11-update/
8. Guizzardi, G., Botti Benevides, A., Fonseca, C.M., Porello, D., Almeida, J.P.A., Prince Sales, T.: UFO: unified foundational ontology. Appl. Ontol. **17**(1), 167–210 (2022)
9. Harris, S., Seaborne, A.: SPARQL 1.1 query language, March 2013. https://www.w3.org/TR/sparql11-query/
10. Hartig, O.: An overview on execution strategies for linked data queries. Datenbank-Spektrum **13**, 89–99 (2013)
11. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology, vol. 1. Morgan & Claypool Publishers (2011)
12. Hogan, A., Umbrich, J., Harth, A., Cyganiak, R., Polleres, A., Decker, S.: An empirical survey of linked data conformance. J. Web Semant. **14**, 14–44 (2012)
13. Klíma, K., Beeke, D.: karelklima/ldkit: 1.0.0, May 2023. https://doi.org/10.5281/zenodo.7905469
14. Ledvinka, M., Křemen, P.: A comparison of object-triple mapping libraries. Semant. Web **11**(3), 483–524 (2020)
15. Prud'hommeaux, E., Boneva, I., Labra Gayo, J.E., Kellog, G.: Shape Expressions Language (ShEx) 2.1, October 2019. https://shex.io/shex-semantics/
16. Schema.org: Vocabulary (2011). https://schema.org/
17. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: JSON-LD 1.1. W3C Recommendation, July 2020
18. Taelman, R., Scazzosi, J.: RDF/JS: query specification (2023). https://rdf.js.org/query-spec/
19. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular SPARQL query engine for the web. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11137, pp. 239–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_15
20. Taelman, R., Vander Sande, M., Verborgh, R.: GraphQL-LD: linked data querying with GraphQL. In: The 17th International Semantic Web Conference, ISW C2018, pp. 1–4 (2018)
21. Taelman, R., Vander Sande, M., Verborgh, R.: Bridges between GraphQL and RDF. In: W3C Workshop on Web Standardization for Graph Data. W3C (2019)

22. Taelman, R., Verborgh, R.: Evaluation of link traversal query execution over decentralized environments with structural assumptions. arXiv preprint arXiv:2302.06933 (2023)
23. 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