

# Improving Linked Data development experience with LDkit

Karel Klíma<sup>a,\*</sup>, Ruben Taelman<sup>b</sup> and Martin Nečaský<sup>a</sup>

<sup>a</sup> *Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Czechia*  
*E-mails: karel.klima@matfyz.cuni.cz, martin.necasky@matfyz.cuni.cz*

<sup>b</sup> *IDLab, Department of Electronics and Information Systems, Ghent University – imec, Belgium*  
*E-mail: ruben.taelman@ugent.be*

## Abstract.

The adoption of Semantic Web and Linked Data technologies in web application development has been impacted by less-than-optimal development experience. Front-end web application development is inherently challenging, as developers must master a multitude of technologies and frameworks even to build simple applications. Adding Linked Data technologies to the mix further complicates this challenge by increasing the number of technologies that need to be learned. The Semantic Web community has historically struggled to provide front-end developers with quality tools and libraries for working with Linked Data; consequently, developers often prefer traditional solutions based on relational or document databases that offer far superior developer experience. To address this issue, we developed LDkit, an innovative Object Graph Mapping (OGM) framework for TypeScript. The framework works as the data access layer, providing model-based abstraction for querying and retrieving RDF data. LDkit transforms the data between RDF representation and TypeScript primitives according to user-defined data schemas, simplifying the use of the data and ensuring end-to-end data type safety. This paper introduces LDkit, describes its design and implementation fundamentals with focus on its developer interface and integration with other related technologies. Building on community feedback and experience from using LDkit, we introduce major enhancements that simplify data querying and update for common and uncommon web application scenarios, further improving developer experience. Finally, we demonstrate impact of LDkit by examining usage of the framework in real-world projects. LDkit aims to enhance the web ecosystem by making Linked Data more accessible and integrated into mainstream web technologies.

Keywords: Linked Data, Developer Experience, Data Abstraction

**Repository:** <https://github.com/kareklklima/ldkit>

**Documentation:** <https://ldkit.io>

**License:** MIT

## 1. Introduction

The Semantic Web and Linked Data have emerged as powerful technologies to enrich the World Wide Web with structured and interconnected data [4]. These technologies enable a more meaningful representation of web content, facilitating advanced data integration and interoperability. However, despite their advantages, their adoption has been relatively slow. A primary obstacle is the difficulty of querying distributed Linked Data within web applications [5], a challenge often framed as the expressivity/complexity trade-off.

---

\*Corresponding author. E-mail: karel.klima@matfyz.cuni.cz.

Expressivity in Linked Data refers to the ability to represent rich semantics and relationships between resources, often using ontologies and domain-specific vocabularies [15]. A more expressive data model allows for precise and semantically rich descriptions, improving data integration and reasoning capabilities. However, this increased expressivity comes at a cost. Greater expressivity typically results in increased complexity in data retrieval and processing. More sophisticated data models require advanced query languages, such as SPARQL [13], and computationally intensive processing methods. As a result, working with Linked Data often demands significant development effort and extensive computational resources, creating barriers to its practical implementation in web applications.

To address these challenges, several tools have been developed to simplify Linked Data querying while maintaining its expressivity. Notable examples include Comunica, a modular query engine designed for querying heterogeneous Linked Data sources [37], and LDflex, a domain-specific language that provides a more intuitive way to query RDF data [38]. These solutions help bridge the gap between expressivity and usability by abstracting the complexities of Linked Data querying. Nevertheless, the ecosystem of Linked Data tools for web development remains limited, with existing solutions lacking the maturity and comprehensive feature set found in more established web technologies.

Over the past decade, web development has undergone a profound transformation, driven by the increasing demand to migrate traditionally desktop-native applications to the web. As more complex software systems transitioned to web-based environments, the need for more powerful and scalable front-end technologies became evident. This shift led to the emergence and widespread adoption of advanced frameworks such as React [61], Vue [70] and Angular [39], which enabled the development of highly interactive and responsive web applications.

Alongside these advancements, the rise of TypeScript [68], a statically typed superset of JavaScript, further strengthened the web development ecosystem. By introducing static typing, TypeScript improved code quality, enhanced developer productivity, and provided robust tooling support, making it more feasible to build and maintain large-scale applications. These innovations have collectively equipped developers with the necessary tools to create sophisticated, feature-rich web applications that rival their desktop counterparts in functionality and performance.

However, the recent advancements in web development have significantly expanded the skill set required of web developers, who now must navigate a growing ecosystem of technologies. The increasing complexity of web applications demands a deeper understanding of software architecture principles and many advanced concepts such as state management, asynchronous programming, security best practices, and performance optimization. In this context, the adoption of Linked Data in web applications presents yet another challenge for developers due to its inherent complexity. Unlike conventional data management approaches, Linked Data requires developers to work with RDF, SPARQL, and other knowledge graph technologies, which introduce unfamiliar paradigms and demand a deeper understanding of semantic data structures. Therefore, to ease this transition, it is essential to provide robust developer tools that abstract much of the complexity while preserving the expressiveness and power of Linked Data.

To address the needs of modern web developers, we created LDkit, a novel Linked Data abstraction designed to provide a [data type-safe](#) way for interacting with Linked Data from within web applications. 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.

This paper [extends our previous work on LDkit \[18\]](#) and makes three key contributions to the development of Linked Data applications and adoption of Semantic Web technologies:

- [First, it elaborates on the architecture and design of LDkit, providing an in-depth discussion of data schema construction, as well as data querying and updating. The paper includes numerous code examples that demonstrate typical Linked Data use cases, illustrating LDkit’s internal mechanisms such as SPARQL query generation and the transformation between the RDF data model and TypeScript objects.](#)
- [Second, it presents enhancements to LDkit introduced since the original publication, addressing former limitations and incorporating feedback from the community. These enhancements focus on improving the developer experience, including usability refinements and solutions to common challenges in working with Linked Data.](#)
- [Third, it provides an empirical analysis of LDkit’s usage, offering insights into adoption trends, real-world application scenarios, and performance considerations. This analysis not only validates the effectiveness of LDkit but also identifies areas for future research and optimization.](#)

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 provides an overview of the design, implementation, usage examples, and integration of LDkit in web applications, followed by Section 4 that provides thorough overview of improvements in LDkit V2. Section 5 then discusses the developer experience aspects of LDkit, including comparison with related tools and current limitations. Section 6 evaluates performance of LDkit and discusses potential performance bottlenecks. Section 7 examines the usage of LDkit, presenting usage metrics and an analysis of its adoption in existing tools. Finally, Sections 8 and 9 outline directions for future research and provide concluding remarks.

## 2. Related Work

This section presents an overview of the existing approaches and technologies that are closely related to LDkit.

The first subsection focuses on common web application data abstraction solutions and introduces several key data access libraries that are considered state-of-the-art in web development. By examining these well-established solutions, we can infer a set of general qualities that a library such as LDkit should possess in order to achieve wide adoption, ensure usability, and integrate seamlessly with the web application development ecosystem.

The second subsection reviews existing RDF-specific JavaScript/TypeScript libraries, focusing on their capabilities, limitations and relevance to web application development. By examining these solutions, we aim to identify the gaps that remain unaddressed, especially when compared to traditional data abstraction tools.

The third subsection introduces alternative RDF querying approaches that leverage mainstream web application technologies, such as RESTful interfaces and GraphQL.

The fourth subsection examines RDF data documentation strategies and data descriptors, such as data shape definitions, and explores their potential use in Linked Data applications.

The last subsection provides an overview of the different JavaScript runtime environments that are relevant to the execution of LDkit and similar libraries. This technical background is important for understanding the broader context in which LDkit operates. Since LDkit is written in TypeScript and intended to be used in diverse web development environments, it is crucial to discuss the available runtime options, particularly how they affect TypeScript execution, development workflow, and the overall developer experience. Finally, this subsection also clarifies the runtime assumptions made in the paper.

This structure demonstrates how LDkit fits within the broader landscape of web development tools, while also illustrating the unique contributions it makes by integrating Linked Data into mainstream web technologies.

### 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* [59], *TypeORM* [67] or *Mongoose* [53]. Corresponding tools for graph databases are typically referred to as Object-Graph Mapping (OGM) or Object-Triple Mapping (OTM) [24] libraries, and include *Neo4j OGM* [54] for Java and *GQLAlchemy* [44] 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* [47].

*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 example of a driver-based abstraction heavily utilized in web applications is the *MongoDB Node.js Driver* [52].

1 Finally, *API-based Data Access* abstractions facilitate access to databases indirectly through APIs, such as REST- 1  
2 ful or GraphQL APIs. They provide client-side libraries that make it easy to fetch and manipulate data exposed by 2  
3 the APIs. Examples of API-based data access libraries include *tRPC* [65] and *Apollo Client* [40]. 3

4 Each style of abstraction caters to different needs and preferences. Ultimately, the choice of abstraction style 4  
5 depends on the project's specific requirements and architecture, as well as the database technology being used. 5  
6 There are, however, several shared qualities among these libraries that contribute to a good developer experience. 6

7 All of these libraries have *static type support*, which is especially beneficial for large or complex projects, where 7  
8 maintaining consistent types can significantly improve developer efficiency. Static types provide early error detec- 8  
9 tion during compile time rather than runtime, which reduces bugs and unexpected behavior in production since many 9  
10 common mistakes, such as assigning the wrong data type to variables or passing incorrect arguments to functions, 10  
11 are caught during development. In large projects, where many developers work on the same codebase, static types 11  
12 act as a form of self-documentation, improving code readability, maintenance, and developer collaboration. 12

13 Another aspect of the reviewed libraries, which is closely related to static types, is *good tooling support*: these 13  
14 libraries often provide integrations with popular development tools and environments. This support can include 14  
15 autocompletion, syntax highlighting, and inline error checking, further enhancing the developer experience and 15  
16 productivity. 16

17 Furthermore, most of these libraries offer a consistent API across different database systems, which simplifies 17  
18 the process of switching between databases or working with multiple databases in a single application. Finally, 18  
19 abstracting away low-level details allows developers to focus on their application's logic rather than dealing with 19  
20 the intricacies of the underlying database technology. 20  
21

## 22 2.2. JavaScript/TypeScript RDF libraries 22

23 JavaScript is a versatile programming language that can be utilized in various execution environments, such as 23  
24 web browsers, servers, or desktop. As Linked Data and RDF have gained traction in web development, several 24  
25 JavaScript libraries have emerged to work with RDF data. These libraries offer varying levels of RDF abstraction 25  
26 and cater to different use cases. 26  
27

28 Most of the existing libraries conform to the RDF/JS Data model specification [3], sharing the same RDF data rep- 28  
29 resentation in JavaScript for great compatibility benefits. Often, RDF libraries make use of the JSON-LD (JavaScript 29  
30 Object Notation for Linked Data) [31], a lightweight syntax that enables JSON objects to be enhanced with RDF 30  
31 semantics. JSON-LD achieves this by introducing the concept of JSON-LD *context*, which is a mechanism used to 31  
32 map terms in JSON data to concepts and entities in external vocabularies via RDF property and type IRIs<sup>1</sup>. This 32  
33 mapping allows for JSON objects to be interpreted as RDF graphs. 33

34 One of the most comprehensive projects is *Comunica* [37], a **modular** query engine for Linked Data, enabling 34  
35 developers to execute SPARQL queries over multiple heterogeneous data sources with extensive customizability. 35

36 *LDflex* [38] is a domain-specific language that provides a developer-friendly API for querying and manipulating 36  
37 RDF data with an expressive, JavaScript-like syntax. It makes use of JSON-LD contexts to interpret JavaScript 37  
38 expressions as SPARQL queries. While it does not provide end-to-end type safety, LDflex is one of the most versatile 38  
39 Linked Data abstractions that are available. Since it does not utilize a fixed data schema, it is especially useful for 39  
40 use cases where the underlying Linked Data is not well defined or known. 40

41 There are also several object-oriented abstractions that provide access to RDF data through JavaScript objects. 41  
42 *RDF Object* [60] and *SimpleRDF* [62] enable per-property access to RDF data through JSON-LD context mapping. 42  
43 *LDO (Linked Data Objects)* [51] leverage ShEx [29] data shapes to generate RDF to JavaScript interface, and static 43  
44 typings for the JavaScript objects. *OSM (Object-semantic mapping)* [57] utilizes proprietary model definition to map 44  
45 RDF data to model instances. *Soukai-solid* [64] provides OGM-like access to Solid Pods [63] based on a proprietary 45  
46 data model format. 46

47 Except for LDflex, the major drawback of all the aforementioned Linked Data abstractions is that they require 47  
48 pre-loading the source RDF data to memory. For large decentralized environments like Solid [63], this pre-loading 48  
49 49

---

50 <sup>1</sup>Internationalized Resource Identifier 50  
51 51

1 is often impossible, and we instead require discovery of data during query execution [34]. While these libraries offer  
2 valuable tools for working with RDF, when it comes to web application development, none of them provides the  
3 same level of type safety, tooling support and overall developer experience as their counterparts that target relational  
4 or document databases.

### 6 2.3. Alternative RDF querying approaches

8 In addition to dedicated RDF libraries, several attempts have been made to simplify the integration of Linked Data  
9 into web applications by leveraging technologies familiar to web developers and abstracting away the complexity  
10 of SPARQL querying.

11 One such project is GRLC [25], a tool that transforms a set of SPARQL queries into Linked Data Web APIs.  
12 This enables application developers to interact with Linked Data via a REST API, without requiring knowledge  
13 of SPARQL. Furthermore, GRLC automatically generates an OpenAPI [26] specification, which developers can  
14 leverage as documentation and use to generate additional application artifacts, such as TypeScript definitions.

15 GRLC offers benefits to both application developers and RDF data providers. Developers may deploy GRLC as a  
16 security layer to prevent exposure of SPARQL queries and endpoints to end users. RDF data providers, on the other  
17 hand, may host a GRLC server alongside an existing SPARQL endpoint to facilitate easier access for developers  
18 unfamiliar with SPARQL, in order to simplify data browsing and querying. However, this extra architectural layer  
19 may be impractical or undesirable in certain scenarios, and may add significant performance overhead.

20 In recent years, the GraphQL [45] interface has gained popularity as an alternative to REST interfaces, due to its  
21 flexible data retrieval, strongly typed schema, and the ability to group multiple REST requests into one. A notable  
22 element of this interface is the GraphQL query language, which is popular among developers due to its ease of use  
23 and wide tooling support. However, GraphQL uses custom interface-specific schemas, which are difficult to federate  
24 over, and have no relation to the RDF data model.

25 That is why, in the recent years, we have seen several initiatives [35] [36] [2] attempting to bridge the worlds  
26 of GraphQL and RDF, by translating GraphQL queries into SPARQL, with the goal of lowering the entry-barrier  
27 for writing queries over RDF. While these initiatives addressed the problems to some extent, there are still several  
28 drawbacks to this approach. Most notably, similar to GRLC, it requires the deployment of a dedicated server.

### 30 2.4. RDF Data descriptors

31 One of the key challenges in developing Linked Data applications is understanding what data is available within  
32 an RDF data source and how it can be explored effectively. While it is possible to investigate data directly using  
33 RDF data visualization tools or via SPARQL endpoint queries, this approach can be inefficient and opaque, particu-  
34 larly when dealing with unfamiliar or complex datasets. Consequently, various forms of documentation and data  
35 descriptors have been developed to facilitate comprehension, navigation, and use of RDF data.

36 Shape Expressions (ShEx) [29] and the Shapes Constraint Language (SHACL) [22] are two prominent schema  
37 languages designed to describe and validate RDF data structures. ShEx offers a compact and expressive syntax  
38 for defining graph patterns, enabling both validation and data documentation. SHACL, standardized by the W3C,  
39 employs an RDF-based syntax to define constraints and rules over RDF graphs, making it highly interoperable  
40 within the semantic web stack. These shape definitions may serve as means of accessing RDF data from web  
41 applications, as demonstrated by the aforementioned LDO library, which utilizes ShEx shapes to generate a type-  
42 safe RDF data access layer.

43 The JSON-LD *context* provides a form of RDF data description as well, albeit not as specific as ShEx or SHACL.  
44 For example, the LDflex library uses JSON-LD contexts to generate SPARQL queries and to map RDF data to  
45 JavaScript primitives.

46 Finally, the Vocabulary of Interlinked Datasets (VoID) [1] provides a standardized way to describe metadata about  
47 RDF datasets, including structural metadata that describe the structure and schema of datasets. This is particularly  
48 useful for tasks such as querying and data integration. The SPARQL Editor [75] uses VoID description present in  
49 the triplestore to provide autocomplete features when composing SPARQL queries.  
50  
51

1 These data description technologies are typically employed in back-end systems or data pipelines, with the possible  
2 exception of JSON-LD, which benefits from a comprehensive web-based library<sup>2</sup>. Consequently, the available  
3 tooling for these technologies predominantly targets server-side technologies and programming languages. How-  
4 ever, thanks to the general support of WebAssembly across modern web platforms, it has become feasible to lever-  
5 age tooling originally developed for backend environments within web runtime contexts. Notable WebAssembly-  
6 enabled projects include Rudof [23], a Rust library for handling RDF data models and shapes, and Oxigraph [73], a  
7 graph database that implements the SPARQL standard.

8 Despite the portability advantages offered by WebAssembly, its integration into web applications must be ap-  
9 proached with caution. The size of WebAssembly binaries and the startup latency they introduce can negatively  
10 impact the responsiveness of browser-based applications (for instance, the Oxigraph WASM binary is 3.5 MB).  
11 For these reasons, it may be more appropriate to deploy WebAssembly-based libraries in server-side JavaScript  
12 environments.

### 13 2.5. JavaScript runtime environments 14

15 JavaScript code can be executed in a variety of environments. The most widely used runtime is perhaps  
16 Node.js [55], which pioneered usage of JavaScript outside of web browsers. Node.js benefits from a large ecosystem  
17 of libraries and tools, that is facilitated by the NPM [56] package registry, the largest software registry in the world.  
18 One of the drawbacks of Node.js is that it cannot execute TypeScript files directly – in order to achieve that, Type-  
19 Script **must first be** transpiled into JavaScript, **either as a build step, or through just-in-time (JIT) transformation**  
20 **using** Node.js wrappers like ts-node [66].

21 Insufficient TypeScript support, antique CommonJS module system and absence of comprehensive development  
22 toolchain in Node.js gave rise to alternative JavaScript runtimes in recent years.

23 The most prominent of them is Deno [42]. The Deno runtime natively supports TypeScript, JSX and modern  
24 ECMAScript features with zero configuration. It is built on web standards, and includes essential tools to build, test  
25 and deploy applications. Even though Deno is backwards compatible with Node.js code and supports NPM package  
26 registry, Deno can import modules from any location on the web via URL, like GitHub, a personal webserver or  
27 a CDN like esm.sh [43]. Recently, the Deno team introduced JSR [46], a modern open-source JavaScript package  
28 registry that is runtime agnostic and supports TypeScript natively.

29 Bun [41] is another JavaScript runtime. Similar to Deno, it is an all-in-one JavaScript and TypeScript toolkit for  
30 bundling and testing applications. Bun is designed as a drop-in replacement for Node.js, and includes an NPM-  
31 compatible package manager.

32 Web browsers represent the traditional platform for JavaScript execution, arguably the most important one when  
33 it comes to user-facing applications. Same as Node.js, web browsers do not support TypeScript and require script  
34 transpilation. Additionally, because some of the interfaces provided by Node.js or other runtimes do not have direct  
35 equivalents in browser environments, they need to be polyfilled to ensure full functionality. Polyfills allow devel-  
36 opers to emulate missing features, providing a seamless and consistent experience across server and client-side  
37 executions.

38 The examples in the rest of [the paper](#) assume the usage of TypeScript and Node.js runtime, since it is predomi-  
39 nantly used at the time of writing, but they are easily adaptable (with minimal changes) to other runtime environ-  
40 ments as well.

## 41 3. LDkit 42

43 LDkit is a Linked Data OGM toolkit that provides a [type-safe data](#) abstraction layer for interacting with Linked  
44 Data from within web applications. It [enables RDF](#) data query and update over a variety of data sources. It is written  
45 in TypeScript and designed to be used on [the client or server](#). In this section, we provide a high level perspective of  
46 LDkit design philosophy, architecture, capabilities and discuss some of its most important components.

47 <sup>2</sup><https://github.com/digitalbazaar/jsonld.js> 48

### 3.1. Design philosophy

The development of LDkit was driven by the following design principles, which evolved from the initial requirements [18] to fully realized features:

#### P1 Embraces Linked Data **heterogeneity**

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 [5]. 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 [16]. LDkit embraces this heterogeneity by supporting the querying of Linked Data from various data sources, such as SPARQL endpoints and Solid pods, and various RDF representations.

#### P2 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 translation between the simplified application model and the underlying data representation. In LDkit, the data model (called *schema*) is represented by a simple TypeScript object that contains a definition of a data shape - a class of entities and their properties - eventually utilized to query RDF data. [An example of a simple schema of a book is available in Listing 1.](#) LDkit schemas are based on JSON-LD context format, and simple JSON-LD contexts can usually be easily transformed to schemas. As such, schemas are easy to create and maintain, and are easily separable from the rest of the application so that they can be shared as standalone artifacts. Even though the Schema is a simple structure, it offers comprehensive RDF data abstraction [including data types specification, arity of values specification \(optional value, single value, array of values\), or nesting one schema in another one.](#) Schemas are discussed in more detail in Section 3.3.

```
1 import { dcterms, xsd } from "ldkit/namespaces";
2
3 const BookSchema = {
4   "@type": dcterms.BibliographicResource,
5   "title": dcterms.title,
6   "publicationDate": {
7     "@id": dcterms.issued,
8     "@type": xsd.date,
9   },
10 } as const;
```

Listing 1: Example of a simple *book* schema, including a title, and a publication date.

#### P3 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 is 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*.

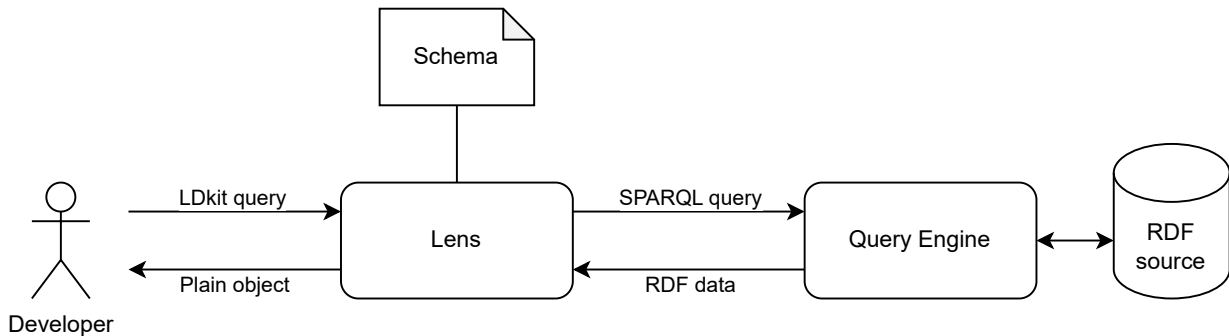


Fig. 1. Basic operation of LDkit.

#### P4 Aims to provide positive developer experience

LDkit achieves a good developer experience by focusing on several key aspects. First, LDkit provides an **ORM-like programming interface that should feel familiar to developers new to the framework, making the learning curve more manageable**. Second, the toolkit leverages 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 is compatible with popular web application libraries and frameworks, allowing developers to incorporate LDkit into their existing workflows easily. By focusing on these aspects, LDkit creates 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.

#### P5 Adheres to existing Web standards and best practices

LDkit adheres 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.

### 3.2. Design fundamentals

In this section, we provide a high-level overview of how LDkit works, of its capabilities and discuss some of its most important components.

The primary objective of LDkit is to provide TypeScript native abstraction to RDF data. It achieves that using the *Lens*<sup>3</sup> component that **provides programming interface to query RDF using simple LDkit Query language**. Using *Schema*, Lens translate the simple queries into SPARQL queries, which are then executed over a target RDF data source using *Query engine*. The Query engine returns retrieved RDF data back to Lens; the data is then decoded using *Schema* again to TypeScript native objects that are ready to be handled by the developer. The process is depicted in Figure 1.

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.*

<sup>3</sup>This should not be confused with the *lens* concept from functional programming used to access and update parts of a data structure in an immutable and composable way. The naming of the *LDkit Lens* component is purely coincidental and bears no relation to the functional programming concept.

```

1  import { createLens } from "ldkit";
2  import { dbo, xsd } from "ldkit/namespaces";
3
4  const PersonSchema = {
5    "@type": dbo.Person,
6    "name": dbo.birthName,
7    "birthDate": {
8      "@id": dbo.birthDate,
9      "@type": xsd.date,
10   },
11 } as const;
12
13 const Persons = createLens(PersonSchema, {
14   sources: ["https://dbpedia.org/sparql"]
15 });
16
17 const adaIri = "http://dbpedia.org/resource/Ada_Lovelace";
18 const ada = await Persons.findByIri(adaIri);
19
20 console.log(ada.name); // "The Hon. Augusta Ada Byron"
21 console.log(ada.birthDate); // Date object of 1815-12-10
22
23

```

Listing 2: LDkit usage example

The example in Listing 2 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.

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.

Listing 3 illustrates how exactly the data schema corresponds to the original RDF data and resulting TypeScript primitives, and presents an example of equivalent data models in both domains.

```

42 // TTL data model
43 @prefix dbo: <http://dbpedia.org/ontology/> .
44 @prefix dbr: <http://dbpedia.org/resource/> .
45 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
46
47 dbr:Ada_Lovelace a dbo:Person ;
48   dbo:birthName "The Hon. Augusta Ada Byron" ;
49   dbo:birthDate "1815-12-10"@xsd:date .
50
51 // TypeScript type inferred from the schema

```

```

1 11 type PersonType = {
2     $id: string;
3     name: string;
4     birthDate: Date;
5 };
6
7 // Resulting JavaScript object equivalent to the RDF data
8 const ada: PersonType = {
9     $id: "http://dbpedia.org/resource/Ada_Lovelace",
10    name: "The Hon. Augusta Ada Byron",
11    date: new Date("1815-12-10"),
12 };

```

Listing 3: Example of equivalent data models, and an equivalent TypeScript type.

### 3.3. Schema

The *schema* is the key concept in LDkit; understanding of the term is essential for efficient use of the library.

On a conceptual level, a data schema is a definition of a data shape through which the RDF data are queried, and how the data are eventually transformed to JavaScript primitives. It is similar to a data model for standard ORM libraries. Through schema, the library users describe a class of entities and their properties to be retrieved from an RDF source.

Listing 4 includes a TypeScript definition of the schema object, including RDF types and properties specification. Any object that satisfies the *Schema* type is a valid LDkit schema.

```

1  type Schema = {
2     "@type"?: Iri | Iri[]; // Optional type restriction
3     [key: string]: Iri | Property;
4 }
5
6  type Property = {
7     "@id": Iri;
8     "@type"?: DatatypeIri; // Supported data type, e.g. xsd:date
9     "@schema"?: Schema; // Nested schema
10    "@optional"?: true;
11    "@array"?: true;
12    "@multilang"?: true;
13    "@inverse"?: true;
14 }

```

Listing 4: TypeScript definition of LDkit schema.

#### 3.3.1. RDF Type Definition

A schema may include a `@type` definition, that is, a specification of one or more RDF types (specified by IRIs) of the entities to be queried. LDkit uses this information as a restriction and considers only the subjects that have all of the specified types. The type definition is optional – if the user omits it, then only the shape of properties is considered for querying data. Listing 5 includes examples of type definitions.

```
1 const MySchemaWithTwoTypes = {
2   "@type": [
3     "http://example.com/ontology/MyClass",
4     "http://example.com/ontology/MyOtherClass",
5   ],
6 } as const;
7
8 const MySchemaWithoutType = {
9   title: dcterms.title // http://purl.org/dc/terms/title
10 } as const;
```

Listing 5: Examples of type definitions in schema.

### 3.3.2. Properties Definition

The schema lets developers define general shape of the data by specifying properties of entities, their data type (a data primitive based on XSD [28], or a nested schema), and arity.

A schema typically includes a map of multiple data properties, that is, a mapping from simplified names to RDF predicates specified by IRI.

In addition, each property definition may include one or more property modifiers, further specifying how the data should be queried and outputted. LDkit schema supports the following modifiers:

- **@id** (*required*)  
The RDF predicate IRI.
- **@type**  
The RDF datatype of the property value. If omitted, defaults to `xsd:string`.
- **@schema**  
Nested subschema. It may contain the definition in place, or a reference to another JavaScript object containing LDkit schema specification, thanks to the composability properties of schema. Alternative to `@type`.
- **@optional**  
If set, indicates that the property is optional. By default, properties are considered required, and LDkit will only query entities having such properties, or will require property values when creating or updating an entity.
- **@array**  
If set, indicates to treat the property as having multiple values. By default, properties are considered single-value only. If there are multiple values, only one of them is accepted – the first one encountered in the dataset.
- **@multilang**  
If set, LDkit will treat the property as language-enabled, that is, it will transform literal values annotated by `@language` tags to a key-value map of languages and their respective literal values.
- **@inverse**  
If set, indicates that the property is in an inverse relation, which is useful to represent incoming links. Normally, the properties are matched using `<?entity ?property ?value>` pattern, but if the inverse attribute is set, the matching is done using `<?value ?property ?entity>` instead. The attribute is equivalent to JSON-LD `@reverse` keyword or ShExJ `inverse` keyword.

The modifiers may be combined together as required. For example, a property with the `@array` and `@multilang` flags will have all its literal values transformed to a key-value map, the *key* being the `@language` annotation, and the *value* being an array of literals belonging to the particular language.

The `@type` property includes a datatype IRI. LDkit supports two-way conversion between commonly used RDF data types and TypeScript native types. Both the data and TypeScript types are adequately converted. For example,

a property of type `xsd:date` is converted to a TypeScript `Date` object. The complete reference of supported data types, including a mapping to resulting TypeScript types, is available in the LDkit documentation<sup>4</sup>.

In addition to the built-in types, LDkit may be extended by custom datatypes handlers. We discuss this in more detail in Section 4.

While property values of data entities are usually literals, in some cases it may be useful to query for named nodes (e.g. IRIs of linked entities) instead of literal property values. To address that, we introduced a special datatype `ldkit:IRI` from the LDkit ontology [17].

Finally, for developer convenience, the schema supports a shorthand property notation, where instead of a complex property definition using a key-value object, the developer provides only its predicate IRI. In such case, LDkit assumes that the property is required, of type `xsd:string`, and has a single value (not an array).

### 3.3.3. Schema nesting, composition, recursion and handling complex types

The LDkit schema support is designed with two important constraints. First, the schema must be finite to prevent infinite recursion in the resulting data. Second, it must be possible to retrieve all the data necessary to populate the schema using a single SPARQL query, in order to apply data constraints directly in the query (e.g. ensure existence of all required properties).

Due to these constraints, although LDkit supports schema nesting and composition, a schema cannot reference itself, even transitively. This restriction is enforced syntactically in TypeScript, thereby preventing users from inadvertently creating invalid recursive schemas.

However, Linked Data often includes entities that reference others of the same type, and potentially the same schema. Such cases can be addressed through application level recursion. At the schema level, instead of referencing the complete schema recursively, users may retrieve only the IRIs of related entities, or use a subschema. Listing 6 illustrates a typical use case involving `foaf:knows` predicate where one person may refer to another. In this example, instead of referencing the entire `PersonSchema`, the query retrieves only IRIs of referenced entities of a RDF type `foaf:Person`. These IRIs can then be used in subsequent queries that apply the main schema.

```
1 import { foaf } from "ldkit/namespaces";
2
3 const PersonIdSchema = {
4   "@type": foaf.Person,
5 } as const;
6
7 const PersonSchema = {
8   ...PersonIdSchema, // Composition; same as "@type": foaf.Person,
9   name: foaf.name,
10  knows: {
11    "@id": foaf.knows,
12    "@schema": PersonIdSchema, // Nested schema
13    "@array": true,
14  },
15 } as const;
```

**Listing 6:** Simulating schema recursion using a base identification schema. Using this schema, LDkit query retrieves IRIs of other persons that a person knows; these IRIs may be subsequently recursively used to query for more data.

Another common use case in Linked Data is the handling of complex data types, for example, those where the range of a property is the union or the intersection between two classes. There are two alternative strategies to address this scenario.

<sup>4</sup><https://ldkit.io/docs/features/supported-data-types>

To illustrate these strategies, consider the `schema:author` property from the Schema.org [30] ontology, whose range is defined as either `schema:Person` or `schema:Organization`.

The first approach, demonstrated in Listing 7, is similar to the recursion example discussed earlier. It uses an intermediary schema to resolve RDF types of the linked entity. The application can then query additional data based on the resolved RDF types, ideally using dedicated schemas for *Person* and *Organization*.

```

1 import { ldkit, rdf, schema } from "ldkit/namespaces";
2
3 const TypesSchema = {
4   types: { // Results in an array of all rdf:type values for the resource
5     "@id": rdf.type,
6     "@type": ldkit.IRI,
7     "@array": true,
8   },
9 } as const;
10
11 const CreativeWorkSchema = {
12   "@type": schema.CreativeWork,
13   author: {
14     "@id": schema.author,
15     "@schema": TypesSchema, // Can be a schema:Person or schema:Organization
16   },
17 } as const;

```

**Listing 7:** Handling complex data types using intermediary types resolution. Using this schema, LDkit query retrieves an IRI of the author of a *CreativeWork* including all associated RDF types. This information may be used during application runtime to query for more data, using separate schemas for *Person* or *Organization*.

The second strategy involves creating a synthetic schema that contains properties that belong to either *Person* or *Organization* entities, using the `@optional` property attribute. This is possible to achieve since LDkit does not require `rdf:type` specification on the schema level. The example synthetic schema is demonstrated in Listing 8.

```

1 import { ldkit, rdf, schema } from "ldkit/namespaces";
2
3 const PersonOrOrganizationSchema = {
4   "givenName": {
5     "@id": schema.givenName,
6     "@optional": true, // Exists only for schema:Person
7   },
8   "familyName": {
9     "@id": schema.familyName,
10    "@optional": true, // Exists only for schema:Person
11  },
12  "legalName": {
13    "@id": schema.legalName,
14    "@optional": true, // Exists only for schema:Organization
15  },
16  "address": {
17    "@id": schema.address, // Common property
18  },
19 } as const;

```

```

1 20
2 21 const CreativeWorkSchema = {
3 22   "@type": schema.CreativeWork,
4 23   author: {
5 24     "@id": schema.author,
6 25     "@schema": PersonOrOrganizationSchema,
7 26   },
8 27 } as const;

```

Listing 8: Handling complex data types using intermediary types resolution. Using this schema, LDkit query retrieves an IRI of the author of a *CreativeWork* including all associated RDF types. This information may be used during application runtime to query for more data, using separate schemas for *Person* or *Organization*.

### 3.4. Querying data

In Listing 2, we showed how LDkit can map a particular data entity specified by IRI from RDF to TypeScript model. When building web applications that browse data, a more advanced interface is required, so that the user can efficiently browse and utilize the data. For example, traditional ORM solutions provide a way to retrieve all entities, lookup entities based on particular criteria, or paginate results to support large datasets. LDkit strives to support the same use cases. A key differentiator of LDkit is the ability to retrieve previously unknown data entities without knowing their resource IRI upfront. In contrast, other existing RDF-based solutions typically require a starting IRI or having the entire dataset loaded into memory.

To support such advanced querying needs, LDkit introduces a custom query language that abstracts SPARQL and enables developers to perform filtering and pagination without writing SPARQL manually. This query language provides a familiar and declarative syntax akin to traditional ORMs. Listing 9 shows an example of how to fetch persons named "Alan", and limiting the total number of results to ten. It also shows the particular SPARQL query that is created and executed by LDkit under the hood.

```

30 1 const persons = await Persons.find({
31 2   where: {
32 3     name: {
33 4       $strStarts: "Alan",
34 5     },
35 6   },
36 7   take: 10,
37 8 });
38 9
39 10 // Corresponding SPARQL query
40 11 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
41 12 PREFIX dbo: <http://dbpedia.org/ontology/>
42 13 PREFIX ldkit: <https://ldkit.io/ontology/>
43 14
44 15 CONSTRUCT {
45 16   ?iri rdf:type ldkit:Resource .
46 17   ?iri dbo:birthName ?iri_0 .
47 18   ?iri dbo:birthDate ?iri_1 .
48 19 }
49 20 WHERE {
50 21   {
51 22     SELECT DISTINCT ?iri

```

```

1 23     WHERE {
2 24         ?iri rdf:type dbo:Person .
3 25         ?iri dbo:birthName ?iri_0 .
4 26         FILTER (STRSTARTS(?iri_0, "Alan")) .
5 27         ?iri dbo:birthDate ?iri_1 .
6 28     }
7 29     LIMIT 10
8 30     OFFSET 0
9 31 }
10 32 ?iri dbo:birthName ?iri_0 .
11 33 FILTER (STRSTARTS(?iri_0, "Alan")) .
12 34 ?iri dbo:birthDate ?iri_1 .
13 35 }

```

Listing 9: List 10 persons with name that starts with "Alan".

The LDKit query may contain a *where* clause that lets the user restrict the values of specific properties. It uses `FILTER` expressions and built-in SPARQL functions to achieve that.

LDKit allows various search and filtering operations that are data-type specific, most of those operations are realized using built-in SPARQL functions. There is support for general comparisons<sup>5</sup>, string functions<sup>6</sup>, and array functions<sup>7</sup>. In addition, users can specify a custom *filter* expression that is inserted directly into the resulting SPARQL query. In addition, most of these filtering operations can be mixed together for a single entity property, so that users can e.g. query for a date range ("*Find all persons born after 1950 and before 1990*").

For pagination, there are *take* and *skip* parameters, that correspond to the `LIMIT` and `OFFSET` in the SPARQL query.

In order to support this kind of query specification, the resulting SPARQL queries are quite complex. The queries can be broken down into three parts:

1. First, a set of entities represented by their IRIs must be established. This is done using a `SELECT` subquery that checks for required properties defined in schema, and employs filtering and pagination.
2. Second, for each IRI found, a graph corresponding to the defined schema is matched, including optional properties. Filtering must be applied on this level as well in order to yield correct results.
3. Third, the graph is finalized using `CONSTRUCT` query, and a special type `ldkit:Resource` is added for each IRI so that it is clear which of the IRIs contained in the resulting set corresponds to the entities found.

In summary, LDKit enhances data browsing by offering retrieval of data entities without known IRIs and does not require the entire dataset to be in memory. It features advanced search and filtering capabilities using SPARQL, allowing efficient exploration of large datasets. Users can filter data based on properties, use data-type specific operations, execute custom SPARQL expressions, and apply pagination, making it versatile for querying any kind of data.

### 3.5. Updating data

LDKit provides means to *insert*, *update* and *delete* RDF data in a developer friendly way, centered around *schema* definition, similar as reading data. The read operations convert RDF data to TypeScript plain objects and primitives. The update operations work exactly in the opposite way - the input are plain objects holding information about entities, which are then converted to RDF and SPARQL Update queries. An example of creating a new entity is shown in Listing 10.

<sup>5</sup>equals, not, gt, lt, gte, lte

<sup>6</sup>contains, strStarts, strEnds, regex, langMatches

<sup>7</sup>in, notIn

```

1  await Persons.insert({
2    $id: "http://dbpedia.org/resource/Alan_Kay",
3    name: "Alan Curtis Kay",
4    birthDate: new Date("1940-05-17"),
5  });
6
7  // Corresponding SPARQL query
8  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
9  PREFIX dbo: <http://dbpedia.org/ontology/>
10 PREFIX dbr: <http://dbpedia.org/resource/>
11
12 INSERT DATA {
13   dbr:Alan_Kay a dbo:Person .
14   dbr:Alan_Kay dbo:birthName "Alan Curtis Kay"@en .
15   dbr:Alan_Kay dbo:birthDate "1940-05-17"^^xsd:date .
16 }

```

Listing 10: Add a Person to a data source, with a preferred language set to "en".

When updating the data, LDkit **assumes** that the data present in the data source correspond to the defined data schema, and the update operations are designed so that the data entities remain sound, that is, if the developer modifies an entity through LDkit, the resulting RDF data always correspond to the defined schema. For example, it is not possible to delete a required property of an entity. With that said, it is always the responsibility of the developer to make sure to use provided interface appropriately, e.g. make sure not to *insert* an entity if it already exists, or not to *update* an entity that does not exist yet. For performance reasons, LDkit does not check for data integrity in the data source.

In order to add new data, users need to specify a full entity (or entities) to insert, including all the required properties. For update operation, it is only needed to provide a subset of entity properties - only the ones that are supposed to change. Listing 11 shows an example of an update query.

```

1  await Persons.update({
2    $id: "http://dbpedia.org/resource/Alan_Turing",
3    name: "Not Alan Turing",
4  });
5
6  // Corresponding SPARQL query
7  PREFIX dbo: <http://dbpedia.org/ontology/>
8  PREFIX dbr: <http://dbpedia.org/resource/>
9
10 DELETE {
11   dbr:Alan_Turing dbo:birthName ?v1 .
12 }
13 INSERT {
14   dbr:Alan_Turing dbo:birthName "Not Alan Turing"@en .
15 }
16 WHERE {
17   dbr:Alan_Turing dbo:birthName ?v1 .
18 }

```

Listing 11: Modify a name of a Person.

### 3.5.1. The singularization problem

One of the peculiarities when working with Linked Data, or with graph data sources in general, is that more often than not, a particular property of an entity may have multiple values, and sometimes the list of values may be quite big. The majority of existing RDF abstractions deal with this issue in a Linked Data way - simply considering any property in data to have multiple values. That is however not good enough developer experience for a lot of scenarios, especially if one needs to rely on a particular data model, or has the data under their control. However, some of the available libraries provide means to define this behavior.

To better understand the issue, we provide examples of singularization in the GraphQL-LD and LDflex libraries.

The GraphQL-LD [35] library treats all properties as arrays by default, but allows for `@single` or `@plural` directives to be added inside the queries to indicate which fields should be singularized and which ones should remain plural. Listing 12 gives an example of how to retrieve single or multiple labels of an entity.

```

1 # RDF dataset
2 <https://example.com/subject> rdfs:label "A", "B", "C" .
3
4 # JSON-LD context
5 {
6   "@context": {
7     "label": { "@id": "http://www.w3.org/2000/01/rdf-schema#label" },
8   },
9 };
10
11 # GraphQL-LD query to retrieve all labels of the subject
12 query @single {
13   label
14 }
15
16 # GraphQL-LD query to retrieve a single label of the subject
17 query @single {
18   label @single
19 }

```

Listing 12: Singularization in GraphQL-LD.

The LDflex [38] library adopts a different approach by providing a fluid JavaScript interface for interacting with Linked Data, allowing users to traverse data similarly to local objects. When a user retrieves a property using `await`, a singular value is returned, whereas iterating over a property using `for await` yields all available values. Listing 13 shows how to print a single or multiple values of the entity.

```

1 // RDF dataset and JSON-LD context are the same as in the GraphQL-LD example
2
3 // LDflex path pointed on the subject entity
4 const subject = path.create({
5   subject: dataFactory.namedNode("https://example.com/subject"),
6 });
7
8 // Print one label
9 console.log(`${await subject.label}`);
10
11 // Print all labels
12 for await (const label of subject.label) {

```

```

13 console.log(`${label}`);
14 }

```

Listing 13: Singularization in LDflex.

### 3.5.2. Singularization in LDkit

LDkit supports singularization definition directly within the model. In the data schema, users can specify which properties are restricted to single value (the default), and which may contain multiple values. This approach enhances the developer experience by ensuring data integrity and providing tailored methods for reading and writing properties based on whether they represent a singular value or an array.

Furthermore, as we mentioned earlier, some arrays may be quite large, raising the question of how to update such lists efficiently. The LDkit library addresses this challenge by offering developers a [mechanism](#) to *set* array-like properties to a specific list of values, *add* new values, or *remove* existing ones. This functionality enables efficient updates to large datasets without requiring the entire dataset to be transmitted. An example of such an update is shown in Listing 14.

```

1 const PersonKnowsSchema = {
2   "@type": foaf.Person,
3   knows: {
4     "@id": foaf.knows,
5     "@type": ldkit.IRI, // A special type for named node
6     "@array": true,
7   },
8 } as const;
9 const PersonFriends = createLens(PersonKnowsSchema, options);
10
11 await PersonFriends.update({
12   $id: "http://dbpedia.org/resource/Ada_Lovelace",
13   knows: {
14     $add: ["http://dbpedia.org/resource/Alan_Turing"],
15     $remove: ["http://dbpedia.org/resource/Alan_Kay"],
16   },
17 });
18
19 // Corresponding SPARQL query
20 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
21 PREFIX dbr: <http://dbpedia.org/resource/>
22
23 DELETE {
24   dbr:Ada_Lovelace foaf:knows dbr:Alan_Kay .
25 }
26 INSERT {
27   dbr:Ada_Lovelace foaf:knows dbr:Alan_Turing .
28 }
29 WHERE {
30   # No conditions specified
31 }

```

Listing 14: Working with arrays, modify friends list of a Person.

### 3.6. 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 [33] 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 [13].

LDkit is fully compatible with Comunica-based query engines. Comunica [37] provides access to RDF data from multiple sources and various source types, including Solid pods, RDF files, Triple/Quad Pattern Fragments, and HDT files.

### 3.7. Converting data between RDF and TypeScript

LDkit implements a runtime RDF decoding mechanism that transforms RDF triples into structured TypeScript objects, according to a user-defined schema. This decoding is performed by the `Decoder` component, which interprets the RDF graph through a schema-driven approach. The algorithm supports language-tagged literals, nested entities, optional and repeated properties, and type validation, enabling reliable and predictable transformation of RDF data into a TypeScript object aligned with the schema.

The input of the decoding algorithm is an RDF graph, a schema, and an optional preferred language settings. The RDF graph is a set of triples (subject S, predicate P, object O) serialized in the following tree structure, grouping predicates by subject, and objects by predicate:

```
{ "S" : { "P" : [ O ] } }
```

The decoding process operates in several stages, formally described in Algorithm 1.

1. **Resource identification:** All RDF subjects in the graph are scanned for an *rdf:type* including *ldkit:Resource*. Only those nodes are considered as root nodes for decoding.
2. **Per-node decoding:** Each resource node is decoded based on the schema, which maps property keys to RDF predicates and type constraints.
3. **Property handling:** Each property is evaluated based on its constraints.
  - **Required vs Optional:** Required properties must be present; missing values raise errors<sup>8</sup>.
  - **Multilingual literals:** If `@multilang` attribute is set, terms are grouped by language.
  - **Arrays:** Properties marked with `@array` are decoded into lists.
  - **Nested schemas:** Properties with `@schema` are recursively decoded into sub-objects.
4. **Language preference:** If a preferred language is specified, literals in that language are prioritized.
5. **Value resolution:** RDF terms are mapped to TypeScript values using built-in or custom data types converters.

To facilitate end-to-end type safety, in addition to the runtime type conversion provided by the `Decoder` component, LDkit includes a compile-time TypeScript type conversion through the `SchemaInterface` type helper. This utility takes the data schema as input and infers a TypeScript type that describes the shape of a single data entity corresponding to the schema. This shape is equivalent to the one produced by the `Decoder` component.

The resolution algorithm employed by `SchemaInterface` closely mirrors that of the runtime `Decoder` in terms of semantic interpretation of schema definitions, including handling of optional properties, arrays, language maps, and value coercion. However, because it operates at compile time, it must express this logic entirely within

<sup>8</sup>LDkit issues SPARQL queries that enforce this constraint, so the input data will always be valid, e.g. data missing required properties will not be retrieved at all in the first place.

```

1  Input: RDF graph  $G$ , schema  $S$ , preferred language  $P$ 
2  Output: List of decoded TypeScript objects
3
4   $output \leftarrow []$ 
5  foreach  $(iri, predicates) \in G$  do
6  |   if  $ldkit:Resource \in predicates[rd\!f:type]$  then
7  |   |    $output[n] \leftarrow DecodeNode(iri, S)$ 
8
9  Function  $DecodeNode(iri, S)$  :
10 |    $nodeData \leftarrow G[iri]$ 
11 |    $result \leftarrow \{\$id : iri\}$ 
12 |   foreach  $(key, property) \in S$  do
13 |   |   if  $key = @type$  then
14 |   |   |   continue
15 |   |    $result[key] \leftarrow DecodeProperty(iri, nodeData, property)$ 
16 |   return  $result$ 
17
18 Function  $DecodeProperty(iri, nodeData, property)$  :
19 |    $terms \leftarrow nodeData[property[@id]]$ 
20 |   if  $terms = \emptyset$  then
21 |   |   return  $property[@optional] ? (property[@array] ? [] : null) : \mathbf{error}$ 
22 |   if  $property[@multilang]$  then
23 |   |   if  $property[@array]$  then
24 |   |   |   return key-value map of  $terms$ , where key is a  $@language$  and value is an array of literals
25 |   |   else
26 |   |   |   return key-value map of  $terms$ , where key is a  $@language$  and value is a literal
27 |   if  $property[@array]$  then
28 |   |   if  $property[@schema]$  then
29 |   |   |   return  $terms$  mapped with  $term \rightarrow DecodeNode(term, property[@schema])$ 
30 |   |   else
31 |   |   |   return  $terms$  mapped to an array of their literal values
32 |   if  $property[@schema]$  then
33 |   |   return  $DecodeNode(first\ valid\ term\ in\ terms, property[@schema])$ 
34 |   if  $P$  is set then
35 |   |   return  $first\ literal\ in\ terms\ with\ matching\ language, or\ first\ literal\ if\ no\ match$ 
36 |   return  $first\ literal\ or\ named\ node\ in\ terms$ 

```

**Algorithm 1:** Decoding RDF Graph to TypeScript Objects

the TypeScript type system, leveraging advanced TypeScript programming features, including mapped types, conditional types, template literal types, and recursive type inference<sup>9</sup>.

The `SchemaInterface` helper is integrated in the `Lens` component interface and facilitates the data-specific auto-completion during development time.

Since LDkit supports not only querying but also updating RDF data, conversion from TypeScript back to RDF is also required. This functionality is provided by the `Encoder` component, which effectively inverts the operation performed by the `Decoder`. It takes as input a set of data entities along with a schema and produces a corresponding set of RDF triples.

However, the encoding algorithm is not entirely symmetrical, as updating RDF data involves more complexity than reading it. Data insertion is relatively straightforward: a data entity is transformed to a set of RDF triples, which

<sup>9</sup>Detailed description of the `SchemaInterface` utility is beyond the scope of this paper; it is best to be studied directly from the source code at <https://github.com/kareklima/ldkit/blob/main/library/schema/interface.ts>

are then added to the data store using the `INSERT` operation of SPARQL Update query. In contrast, data updates are more complex, as they often require replacing existing values in the data store (e.g., when assigning a new value to a required property), or removing them (e.g., when deleting the value of an optional property). Consequently, a typical update operation comprises both triple patterns to be deleted and new triples to be inserted. To support this, the `Encoder` component also generates the corresponding deletion patterns, where the subject is the IRI of the entity being updated, the predicate is the property IRI, and the object is a variable. This pattern enables the removal of any existing values for the specified properties prior to the insertion of new values.

### 3.8. 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*, *Query engine*, *Decoder* and *Encoder* already presented, LDkit also provides several additional components and utilities that facilitate development with Linked Data.

The `QueryBuilder` component automatically generates SPARQL queries for basic Create, Read, Update, and Delete (CRUD) operations based on the data schema. This includes query construction for inserting new entities, retrieving data by IRI or property filters, updating property values, and deleting entities or specific statements.

LDkit also offers a general-purpose, type-safe SPARQL query builder that allows developers to construct arbitrary SPARQL queries programmatically using a fluent interface.

Finally, LDkit also includes *Namespaces* definitions for popular Linked Data vocabularies, such as Dublin Core [6], FOAF [7] or Schema.org [30].

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

### 3.9. Implementation

LDkit is implemented in TypeScript, and requires TypeScript [version 5.5 or higher](#) to be used effectively in applications. For execution in a server environment, LDkit requires at least [Node v20.19.3](#) or [Deno v2.1](#). The source code is available under the [MIT license](#) on GitHub [50] and [Zenodo](#) [21].

Following the standard practices, LDkit is published as an NPM package [49] and as a Deno module [48]. At the time of writing, the latest release is at version [2.4.0](#).

To make adoption easy for new developers, comprehensive documentation, API reference and code examples are available at <https://ldkit.io> or linked from the GitHub repository. This resource includes several examples of fully working demo applications covering both Node and Deno environments, and using vanilla JavaScript, React [61] or Preact [58] frameworks.

LDkit includes a comprehensive suite of over 200 unit tests and integration tests to verify the functionality and interactions of each component within the framework. By rigorously testing the library across various use cases and edge cases, LDkit ensures stability and dependability with each new release, facilitating seamless updates and enhancements while minimizing the risk of introducing regressions.

LDkit is actively developed and maintained by a group of researchers at Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Czechia.

## 4. LDkit V2 improvements

The release of LDkit 2.0 addressed several limitations of its predecessor [18], and since then LDkit has continued to evolve, introducing further refinements and capabilities to support a broader range of application development scenarios. In this section, we examine the major changes in the library.

#### 4.1. LDkit query language

The most significant improvement in LDkit V2 is the introduction of the custom query language that enables filtering and pagination of data entities without requiring knowledge of the SPARQL language. This query language has been thoroughly described in Section 3.4. In V1, users had to provide either a full custom SPARQL query, or at minimum, a custom WHERE clause to issue advanced queries and precisely define the desired results. While the custom SPARQL queries remain supported, the new query language offers a more effective abstraction over SPARQL and functions similarly to traditional ORMs, making it more accessible to application developers.

#### 4.2. Efficient large array manipulation

Another major enhancement introduced in LDkit V2 concerns the manipulation of large arrays, or to be precise, multi-valued properties in RDF datasets. In earlier versions, modifying these array-like structures – such as adding or removing individual items – required either passing the whole updated array to LDkit, or specifying the RDF triples to insert and delete from the underlying datastore. This approach was not only error-prone but also placed a considerable cognitive and technical burden on developers, particularly those unfamiliar with RDF syntax and semantics.

LDkit V2 simplifies this process by introducing a high-level interface for array operations, that was described in Section 3.5. Developers can now perform incremental updates on array-like properties using concise and declarative commands, without the need to manage the RDF graph state manually. This abstraction aligns with the design principles of modern application frameworks, where collection manipulation is a routine task that should not involve low-level data operations. This improvement reduces the likelihood of inconsistencies or unintended side effects in the RDF graph.

#### 4.3. Custom data types

In addition to built-in datatypes, LDkit supports custom two-way conversion based on datatype IRI between RDF literals and TypeScript native primitive or complex types. This is useful when working with complex data formats, such as geometrical points, dates, monetary values, or domain-specific representations that require special parsing and serialization.

In order to support custom types, including end-to-end type safety, developers must provide an explicit TypeScript type definition by augmenting the LDkit's `CustomDataTypes` interface and register conversion functions to translate between RDF literal values and TypeScript native values.

To demonstrate the usage of custom datatypes, Listing 15 illustrates how to store a complex TypeScript object in RDF using JSON serialization.

```
1 import { registerDataHandler } from "ldkit";
2
3 // Custom object shape that we want to store as RDF
4 type Coordinates = { x: number, y: number };
5
6 // TypeScript compile time type conversion using the datatype IRI
7 declare module "ldkit" {
8   interface CustomDataTypes {
9     ["http://example.org/Coordinates"]: Coordinates;
10  }
11 }
12
13 // Actual runtime data conversion
14 registerDataHandler(
15   "http://example.org/Coordinates", // datatype IRI
```

```
1 16 (literalValue: string) => JSON.parse(literalValue) as Coordinates,
2 17 (nativeValue: Coordinates) => JSON.stringify(nativeValue),
3 18 );
4 19
5 20 // Example schema using the newly registered datatype IRI
6 21 const ThingSchema = {
7 22   "@type": "http://example.org/ThingWithCoordinates",
8 23   "coordinates": {
9 24     "@id": "http://example.org/hasCoordinates",
10 25     "@type": "http://example.org/Coordinates",
11 26   },
12 27 } as const;
13 28
14 29 // Resulting TypeScript type of entities queried using the schema
15 30 type Thing = {
16 31   $id: IRI;
17 32   coordinates: Coordinates;
18 33 };
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```

Listing 15: Example of a custom data object stored in RDF using JSON serialization.

#### 4.4. Schema generators

LDkit provides experimental schema generators that transform existing Linked Data definitions into TypeScript schemas compatible with LDkit. These tools are available via the LDkit CLI and support generating code directly from JSON-LD contexts or ShEx shapes.

Although the generators do not fully support the complete feature set of JSON-LD or ShEx – potentially omitting or simplifying complex validation rules, advanced constraints, and specialized constructs – the output schemas can nevertheless serve as a robust starting point for LDkit-based applications.

##### 4.4.1. Using the CLI

The LDkit CLI is included in the NPM `ldkit` package, and as such it can be invoked via `npx` or installed<sup>10</sup>.

The general command syntax:

```
npx ldkit <command> <method> <input>
```

– `<command>`: One of `context-to-schema`, `shexc-to-schema` or `shexj-to-schema`

– `<method>`: Defines how the input is provided. Possible values:

\* `url` – The input is a URL pointing to the resource.

\* `file` – The input is a path to a local file.

\* `arg` – The input is passed directly as a string argument.

– `<input>`: The actual input data, such as URL, depending on the selected method.

The generators produce TypeScript code that can be used directly in LDkit projects. The CLI outputs the schemas to `stdout` which can be redirected to a file. For example, the following command transforms a JSON-LD context from a local file and writes the resulting LDkit schema to a `schema.ts` file:

```
npx ldkit context-to-schema file ./context.jsonld > schema.ts
```

<sup>10</sup>Instructions how to install the CLI for Node and Deno are available online: <https://ldkit.io/docs/components/schema-generators>

#### 4.4.2. JSON-LD

The `context-to-schema` command transforms the `@context` entry from a JSON-LD 1.1 document to a compatible LDkit schema.

Supported JSON-LD features:

- property datatypes set via `@type` keyword
- containers of type `@language`, `@set` and `@list`
- `@reverse` properties
- nested `@context` entries

Since JSON-LD contexts do not support specifying property cardinality, all properties are treated as *required* by default in the LDkit schema, which may require manual adjustment. Nested context entries are transformed to explicit LDkit schemas, meaning that a single JSON-LD context may yield multiple LDkit schemas that can be used independently, if needed.

Example of a JSON-LD context transformation:

```
npx ldkit context-to-schema url https://ldkit.io/examples/person.jsonld
```

#### 4.4.3. ShEx

LDkit supports generating schemas from ShEx 2.1 using two commands – `shexc-to-schema` and `shexj-to-schema` that perform ShExC and ShExJ (JSON representation) conversion, respectively.

Supported ShEx features:

- explicit property types
- property types inferred from enumerations (value sets)
- property cardinalities, represented as optional and/or array LDkit properties
- expressions with choices, represented as optional properties
- inverse properties
- nested shapes (both explicit and anonymous)
- simplified AND / OR shapes logic
- reuse of named triple expressions

Explicit ShEx shapes result in corresponding LDkit schemas in the generated TypeScript code. Since ShEx language is more expressive than LDkit schemas, some of the ShEx rules need to be simplified – most notably, LDkit schema does not support alternative choices of any kind, that is, it cannot accurately represent constraints such as *Schema S contains either property A, or property B*. Such rules would be converted to the equivalent of *Schema S contains an optional property A and an optional property B*, meaning that the resulting LDkit schema is more permissive. Nevertheless, the converted schemas should work well enough with data that are valid according to the original ShEx schema.

Example of a ShExC schema transformation:

```
npx ldkit shexc-to-schema url https://ldkit.io/examples/person.shex
```

#### 4.5. Other improvements

LDkit V2 includes built-in support for new RDF data serializations – N-Triples, N-Quads, and Trig. These formats complement the already supported Turtle and RDF/JSON serializations, increasing interoperability with a wider range of Linked Data sources and tools.

Additionally, V2 introduced support for inverse property relations in the schema using the `@inverse` property attribute, discussed in Section 3.3. In V1, the same result could only have been achieved by specifying custom SPARQL queries.

## 5. Developer Experience

Ensuring a positive developer experience is crucial for the adoption and success of any programming tool. This section explores the various ways in which LDkit improves the workflow for developers, making it easier for them to use Semantic Web and Linked Data technologies in their applications.

### 5.1. Comparison with similar tools

Table 1 presents a comparison of LDkit with other similar software based on key features. The comparison includes LDflex, which is the most widely used; LDO, which offers the most advanced TypeScript integration and type safety; and GraphQL-LD. Each of these libraries employs a different approach to querying Linked Data and offers a distinct developer experience. Together, they represent the state of the art in the development of web applications based on Linked Data.

Feature	LDkit	LDflex	Linked Data Objects (LDO)	GraphQL-LD
<b>Language Support</b>	TypeScript	JavaScript	TypeScript	TypeScript
<b>Data Modeling</b>	Utilizes schemas based on JSON-LD context for defining data shapes	Does not use fixed data schemas; flexible property access	Employs ShEx to define and validate data shapes	Does not use fixed data schemas; flexible property access via GraphQL queries
<b>Type Safety</b>	End-to-end type safety with TypeScript types automatically inferred from the property RDF datatypes specified in schema	No built-in type safety	Provides static typings through TypeScript interfaces generated from ShEx shapes	No built-in type safety
<b>Query Mechanism</b>	Generates SPARQL queries based on defined schemas	Uses JavaScript-like expressions interpreted as SPARQL queries	Translates object manipulations into RDF/JS Dataset queries	Uses GraphQL queries translated into SPARQL queries
<b>Update Mechanism</b>	Generates SPARQL UPDATE queries based on defined schemas	Uses JavaScript-like expressions interpreted as SPARQL queries	Translates object manipulations into RDF/JS Dataset queries	Not supported
<b>Environment Compatibility</b>	Browsers – via build tool Node.js – native Deno – native	Browsers – via build tool Node.js – native Deno – via NPM compatibility layer	Browsers – via build tool Node.js – native Deno – via NPM compatibility layer	Browsers – via build tool Node.js – native Deno – via NPM compatibility layer
<b>Integration with Frameworks</b>	Promise-based API; serializable results	Iterative promise-based API; multiple queries needed to read a complex data entity	Promise-based API; React components for Solid connectivity	Promise-based API; serializable results
<b>Data Source Compatibility</b>	Supports multiple distributed data sources	Supports multiple distributed data sources	Requires pre-loading RDF data into memory	Supports multiple distributed data sources
<b>Developer Experience</b>	Focuses on providing a clear API with strong tooling support	Aims to simplify Linked Data interactions with a familiar syntax	Offers an object-like approach with ShEx-based typings	Familiar GraphQL syntax makes it accessible to GraphQL developers

Table 1

Comparison of LDkit, LDflex, Linked Data Objects (LDO), and GraphQL-LD

### 5.2. Key differentiators

LDkit provides a simple way of Linked Data model specification 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

1 requirements. The schema syntax is based on JSON-LD context, and as such it assumes its qualities: it is self-  
2 explanatory and easy to create, and can be reused, nested, and shared independently of LDkit.

3 The *Lens* interface for reading and writing **Linked Data should** feel familiar even to developers new to RDF, as it is  
4 inspired by interfaces of analogous model-based abstractions of relational databases. **LDkit provides tooling support**  
5 **by incorporating end-to-end data type safety**, giving developers instantaneous feedback in the form of autocomplete  
6 or error highlighting within their development environment. The development experience is further enhanced by the  
7 fact that the typings of data entities are inferred directly from the defined *schema* during development time, without  
8 the need to generate additional TypeScript artifacts or provide explicit type information, which is the case for all  
9 other similar tools.

10 Other Linked Data libraries, like LDflex or LDO, make use of the JavaScript *Proxy* object to allow virtual access  
11 to data and override some of their properties. While that approach may be effectively used for developer-friendly  
12 paradigms like fluent interfaces, it may be problematic when working with complex data. The reason for that is that  
13 proxied objects cannot be in principle cached, serialized, printed, or sent from server to client. LDkit on the other  
14 hand opts for representing the data always as JavaScript plain objects and primitives that inherently share all the  
15 aforementioned properties. This approach is implemented by the most advanced ORMs, such as Prisma.

### 17 5.3. Integration and Compatibility

18  
19 LDkit **adheres to selected** W3C standards and recommendations, ensuring compatibility and integration capabil-  
20 ities within the Semantic Web and Linked Data ecosystem. The core of the toolkit is based on the RDF/JS data  
21 model [3] and query [33] specifications, which standardizes the representation of RDF in JavaScript. To interact  
22 with data sources, LDkit utilizes SPARQL Query Language [13] and SPARQL Update [10]; its built-in query en-  
23 gine interacts with SPARQL endpoints using SPARQL Protocol [8].

24 One of the key features of LDkit is its Promise-based API, which aligns with modern asynchronous programming  
25 practices in JavaScript. As a **result**, LDkit can be easily integrated with existing frontend and backend frameworks,  
26 such as React, Angular or **Express**. Whether developers are building complex single-page applications or more  
27 traditional multi-page websites, **LDkit enables RDF** data integration and manipulation across different parts of an  
28 application.

29 Further extending its versatility, LDkit is compatible with multiple JavaScript runtimes, including Node.js, Deno,  
30 and Bun. This compatibility ensures that LDkit can be used in a variety of execution environments—from server-  
31 side applications, serverless and edge workers, to client-side interfaces. This wide range of applicability allows  
32 developers to deploy LDkit in diverse scenarios, whether they are building backend services, interactive client-side  
33 applications, or applications requiring low-latency responses at the edge.

### 35 5.4. The expressivity/complexity trade-off

36  
37 In this section, we examine how the use of LDkit affects expressivity and complexity inherent in Linked Data  
38 applications.

39 LDkit provides a schema-based abstraction over RDF data. Using this schema, LDkit reduces the complexity of  
40 RDF data querying by automatically generating SPARQL queries, fetching data from a data source, and transform-  
41 ing it into TypeScript native types. This preserves the expressivity of the data model through semantic mapping in  
42 the schema, while eliminating the need for direct SPARQL querying and RDF data mapping in developer code.

43 However, building the schema can be challenging, as it requires knowledge of data ontologies and possibly the  
44 data itself. To reduce schema creation complexity, LDkit provides converters that enable the reuse of existing ShEx  
45 shapes or JSON-LD contexts to generate an LDkit schema.

46 When using LDkit, the expressivity of data querying is deliberately reduced through its interface, based on the  
47 assumption that typical application use cases do not require the full degree of expressivity offered by SPARQL.  
48 Instead of SPARQL, developers use an ORM-like programming interface along with simplified query expressions  
49 to filter data.

50 For less typical use cases, LDkit offers an advanced interface that allows developers to specify custom SPARQL  
51 queries to execute against the data source or explicitly define triples to insert into or remove from the data source.

Even in these cases, complexity is still reduced, as LDkit handles query execution and potentially data retrieval and transformation. Additionally, LDkit includes a type-safe SPARQL query builder with a fluent interface that helps to create syntactically correct queries.

In summary, LDkit simplifies RDF data access by reducing complexity while maintaining sufficient expressivity for typical and advanced Linked Data use cases.

### 5.5. Current limitations

To date, the greatest limitation of LDkit remains the inherent complexity of the SPARQL queries it generates to read and update data in RDF data sources. Since the complexity of such queries is directly proportional to the complexity of the data schemas, the more properties developers add to the schema, the less performant the LDkit becomes.

Section 6 presents performance tests of LDkit using various schemas, ranging from simple to complex. These tests provide insight into the expected performance and help identify the threshold of schema complexity at which LDkit continues to operate efficiently. In cases where performance issues arise, we recommend refactoring the data schemas by removing unnecessary properties, splitting the schemas into multiple simpler ones, or reducing schema nesting. In many instances, executing several simpler SPARQL queries can be much faster than running a single complex one.

## 6. Performance

In our previous work [18], we evaluated the performance of LDkit using three typical scenarios involving data queries of increasing complexity, executed via the DBpedia SPARQL endpoint<sup>11</sup>. The experiments showed that the overall performance of LDkit is primarily determined by the execution speed of the SPARQL query engine. The library itself maintained stable performance without substantial degradation, even as the complexity of the scenarios increased. In other words, the majority of the processing time is attributable to the execution of the SPARQL query.

While the evaluation demonstrated that LDkit's performance is adequate for use in Linked Data-based applications and confirmed our assumption that increasing data schema complexity affects performance, it did not provide detailed insight into the extent of this impact – for example, the performance degradation associated with adding a single new property to the data schema.

To address this limitation, we designed a new performance test that would better indicate the impact of complexity of data schemas. A synthetic dataset was constructed comprising 10,000 entities, each containing 10 simple properties, 10 array properties and 10 object properties. Each array property holds three literal values, and each object property includes a sub-entity with three simple properties. To ensure consistency, UUIDv4 random string values were assigned to all properties. The resulting dataset contains a total of 910,000 explicit RDF statements.

The test scenario involves querying all entities in the dataset over 10 iterations, retrieving 1000 entities in each iteration. To accurately measure the performance overhead introduced by LDkit, each query was executed twice: first using LDkit, and then directly against the SPARQL endpoint using the same SPARQL query<sup>12</sup>. To reduce the influence of outliers, the minimum and maximum execution times were discarded, and the remaining values were averaged to yield the mean query time in milliseconds per 1,000 entities.

The sole test parameter – and the only variable across the test runs – is the data schema used for retrieving the data. To evaluate the limits of LDkit, we employed five distinct data schema types:

- **A**: includes only simple properties with singular values,
- **B**: includes only object properties, with each sub-entity containing three simple properties with singular values,
- **C**: includes only array properties, each containing three distinct values,
- **AB**: a combination of A and B, incorporating both simple properties and sub-entities,

---

<sup>11</sup><https://dbpedia.org/sparql>

<sup>12</sup>Caching and indexing in the triplestore was disabled to ensure accurate results

– **ABC**: a combination of A, B, and C, encompassing all property types

The number of properties in each schema type ranged from one to ten. For the composite schema types AB and ABC, this range applies to each individual property type. Listing 16 presents an example of the ABC(1) schema type along with the corresponding entity.

```

1 // PREFIX x: https://x/
2 const ABC1Schema = {
3   "@type": x.Entity,
4   simpleProperty1: x.simpleProperty1,
5   arrayProperty1: {
6     "@id": x.arrayProperty1,
7     "@array": true,
8   },
9   objectProperty1: {
10    "@id": x.objectProperty1,
11    "@schema": {
12      "@type": x.SubEntity,
13      simpleProperty1: x.simpleProperty1,
14      simpleProperty2: x.simpleProperty2,
15      simpleProperty3: x.simpleProperty3,
16    }
17  },
18 };
19
20 // Corresponding example entity
21 const ABC1Entity = {
22   "$id": "https://x/77a1d8ca-3105-4b08-a14d-8e5ff337748a",
23   simpleProperty1: "072c77b0-2a6f-4a5d-a0d1-b7d3aba016e7",
24   arrayProperty1: [
25     "076a2976-9a2b-4d5b-ab0b-72e7e0754816",
26     "120e16f2-e1d1-4f75-8db0-88c6cbe41f15",
27     "5214cc6e-390e-4b9d-8834-49108b50a218",
28   ],
29   objectProperty1: {
30     "$id": "https://x/1bebc97c-f1f2-42e7-8fe4-66666b349117",
31     simpleProperty1: "aefdbb61-a58b-41e1-b3a7-713a79c9d9a4",
32     simpleProperty2: "70c4ad24-94c8-4c62-9200-3e12d24a1aa2",
33     simpleProperty3: "738fb0b2-d61c-4e4f-9bbe-26726cd86235",
34   },
35 };

```

Listing 16: Example of an ABC schema type containing one simple property, one array property and one object property, along with the corresponding example entity.

Figure 2 presents the test results, showing the average time required to query 1,000 entities using the different data schema types. The tests were conducted on a PC with an Intel CPU @ 2.40 GHz, 8 GB RAM, running Windows 10. To minimize the impact of network latency, a local installation of GraphDB version 11.0 was used as the triplestore. The test scripts, the dataset generator, execution instructions, and raw results are publicly available on GitHub<sup>13</sup>.

<sup>13</sup><https://github.com/kareklima/ldkit/tree/main/performance/benchmark>

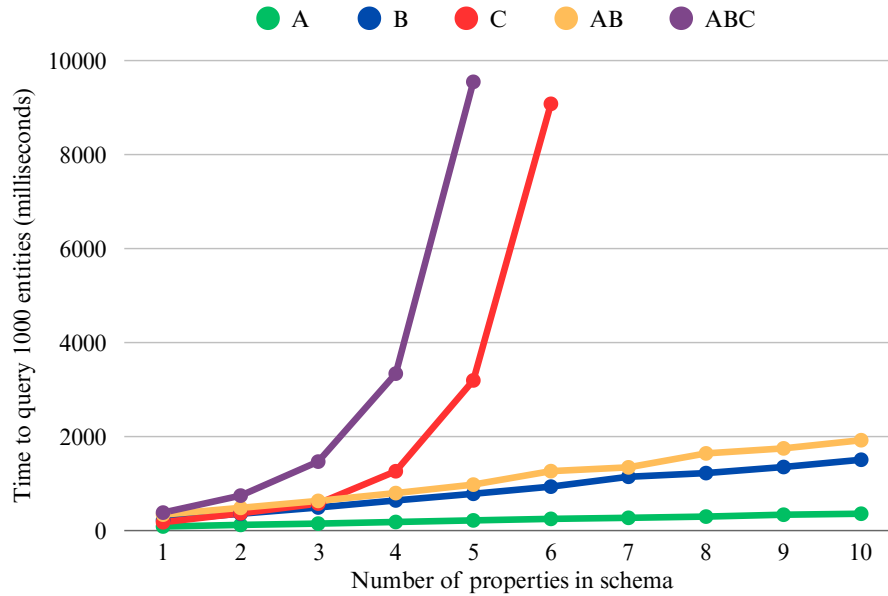


Fig. 2. Performance test results of LDkit showing execution time for five different schema types based on the number of properties the schemas include. Schema A includes only simple properties with one value, schema B includes properties with sub-entities (each of those containing three simple properties), and schema C includes array properties containing three values. Schemas AB and ABC are the combinations, for example, Schema AB with five properties includes five simple properties and five properties with sub-entities.

The results demonstrate that LDkit performs efficiently when handling simple properties or properties with sub-entities. For instance, for a schema with ten simple properties, the average execution time to query 1,000 entities was 374 ms. Similarly, for a schema of type AB, comprising 10 simple properties and 10 sub-entities, the average execution time was 1923 ms, which is considered satisfactory. However, the results clearly indicate that array properties have a significant negative impact on performance. When the number of array properties exceeds four, the execution time surpasses two seconds, rendering such schemas impractical for use in web applications.

The findings also help identify the potential limits of LDkit schema complexity and suggest strategies to enhance performance in web applications. If performance is suboptimal, one viable approach is to refactor the schema by extracting some or all array properties into separate schemas, which can then be queried independently using the same entity IRI.

Finally, comparing the execution time of LDkit with direct SPARQL endpoint queries, the average overhead introduced by LDkit was 7.47% per test case. Figure 3 illustrates the overhead in various test schemas. This overhead is more pronounced for simple schemas and lower for more complex ones, especially those containing multi-value properties. As the SPARQL query execution time increases, LDkit benefits from the streaming nature of result processing, enabling it to process much of the data in parallel as it is received from the SPARQL endpoint.

### 6.1. Independent comparative performance evaluation

The performance of LDkit was also independently evaluated [11] together with six other RDF data abstraction libraries, including LDflex and LDO. Although the evaluation has not been published or peer reviewed at the time of writing, the dataset and results are publicly available on GitHub<sup>14</sup>.

<sup>14</sup><https://github.com/herminiogg/dmaog-paper-evaluation/tree/master/StatisticalAnalysis>

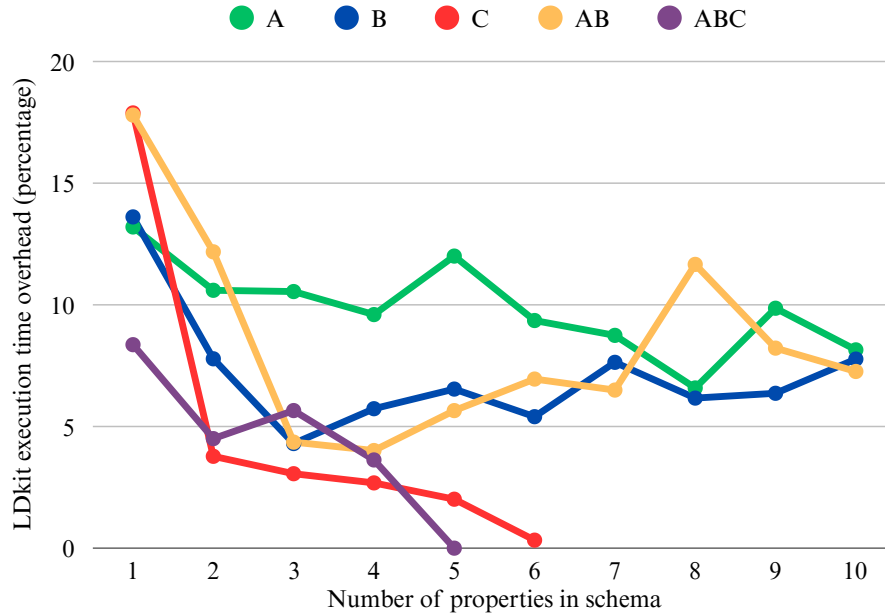


Fig. 3. Performance test results of LDkit showing execution time overhead as the percentage increase in execution time compared to direct SPARQL endpoint queries. Schemas A, B, C, AB, and ABC are the same as in Figure 2: Schema A includes only simple properties with one value, schema B includes properties with sub-entities (each of those containing three simple properties), and schema C includes array properties containing three values. Schemas AB and ABC are the combinations, for example, Schema AB with five properties includes five simple properties and five properties with sub-entities.

The repository provides a detailed description of the performance scenarios, including the steps required to reproduce the tests.

The evaluation assessed library performance across two use cases:

- fetching a set of entities from an RDF data source, and
- fetching a set of entities that satisfy a specific condition.

Statistical analysis<sup>15</sup> indicates that LDkit outperforms all other evaluated libraries and demonstrates significantly better performance than both LDflex and LDO.

## 7. Usage

In this section, we present overview of practical adoption and application of LDkit across various platforms and projects. Understanding the extent of LDkit's utilization not only validates its effectiveness but also demonstrates its impact within the developer community. We explore this through *Usage metrics*, which quantifies its integration into development environments, and *In-use analysis*, which provides qualitative insights from real-world applications and developer experiences.

<sup>15</sup><https://herminiogarcia.com/dmaog-paper-evaluation/StatisticalAnalysis/analysisInR.html>

## 7.1. Usage metrics

The adoption and popularity of LDkit can be partly quantified by several key metrics commonly used within the open-source community.

Since the source code of LDkit is hosted on GitHub, we can inspect some of the statistics that the platform provides. As of June 2025, the LDkit repository gained 65 stars from other GitHub users, and there are 12 other public repositories that depend on LDkit directly. There seems to be no transitive dependencies, which is an indicator that LDkit is being used in applications and not in libraries, exactly as intended.

We can gather additional insight from the NPM platform [49] where the Node.js version of LDkit is being hosted. The public NPM statistics indicate that the total number of downloads of the LDkit package between February 1st 2024 and February 1st 2025 was 11,819, averaging more than 227 downloads per week. Similar to GitHub, NPM does not indicate any dependent packages, confirming the assumption that LDkit is being used in user applications, not in libraries or similar tools.

As we mentioned earlier, LDkit is also available as a Deno module [48]. Unfortunately, the Deno platform does not provide download statistics or dependencies information, therefore we cannot easily gauge the adoption of LDkit for this particular JavaScript runtime.

It is important to note that while these metrics offer valuable insights, they come with certain limitations. For instance, LDkit's usage on platforms like Deno is not tracked, and projects hosted in private GitHub repositories, or other source code platforms such as Bitbucket or GitLab are not included in these metrics. This lack of comprehensive tracking can lead to underestimations of the actual usage figures. Additionally, the metrics from GitHub and NPM might not fully capture the toolkit's impact and adoption due to these platforms' inherent tracking limitations and the fact that they do not encompass all usage scenarios or the variety of developer environments. Moreover, the number of downloads reported by NPM may be overestimated, as a portion of these downloads could be attributed to automated processes such as bots, rather than actual human users.

Thus, while the data from GitHub stars, project dependencies, and NPM downloads are informative, they should be considered indicative rather than exhaustive. These metrics, albeit with their imperfections, still provide a snapshot of LDkit's presence and relevance in the field of Linked Data application development.

## 7.2. In-use analysis

This section presents qualitative analysis of selected projects that use LDkit: *Dutch Digital Heritage Network*, *Assembly Line for conceptual models*, *Dataspecer*, *TypeSPARQ*, *Schema Forge*, and *LDE – Linked Data Engine*. Two of these projects include contributions from the authors of this paper, as noted in the respective subsections below.

### 7.2.1. *Netwerk Digitaal Erfgoed*

Netwerk Digitaal Erfgoed (NDE) [72], or the Dutch Digital Heritage Network, is a collaborative initiative in the Netherlands focused on improving access to and preservation of digital heritage collections from Dutch museums, archives, libraries, and other cultural heritage institutions.

The main goal of NDE is to make the rich digital heritage of the Netherlands accessible, sustainable, and interconnected for users, researchers, and the public. It aims to foster cooperation among heritage institutions and promote the use of open standards, linked data, and sustainable digital practices.

A core component of NDE is The Network of Terms [76] – a search engine for finding terms in terminology sources, such as thesauri, classification systems and reference lists. Given a textual search query, the engine searches one or more terminology sources in real-time and returns matching terms, including their labels and URIs, aggregating the results to the SKOS data model.

This project uses LDkit to query JSON-LD datasets, using Comunica as a query engine. The authors created a complex data schema that employs very advanced LDkit usage. It includes numerous data properties with various data types and leverages features such as nested schemas, arrays, optional properties, and multilingual support.

### 7.2.2. Assembly Line for conceptual models

LDkit is used in a project for the Czech government<sup>16</sup> that aims to build a set of web applications for distributed modeling and maintenance of government ontologies<sup>17</sup>. The ensemble is called *Assembly Line (AL)* [19]. 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 [12]. 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<sup>18</sup> in SKOS (business glossary) and OWL (ontology). The AL tools have to process this native 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.

*Disclaimer: authors of the paper contributed to this project.*

### 7.2.3. Dataspecer

Dataspecer [32], a tool for management and modeling of data specifications based on a domain ontology. Using Dataspecer, the users can generate technical artifacts such as data schemas, e.g., in JSON Schema or XML Schema, and human-readable documentation for a specific dataset based on the provided ontology while maintaining the semantic mapping from the generated artifacts to the ontology. This significantly eases the task of developing data specifications and keeping the corresponding technical artifacts consistent in the process.

Dataspecer provides support for the implementation of artifact generators for any target format, including LDkit. Using Dataspecer, users can automatically generate ready-to-use LDkit data schemas based on generic data specification. These schemas are generated as TypeScript files that are ready to use in an LDkit-based application, speeding the development time considerably.

*Disclaimer: authors of the paper contributed to this project.*

### 7.2.4. TypeSPARQ

TypeSPARQ [69] is a dynamic and user-friendly service that simplifies the process of exploring and extracting data from SPARQL endpoints. With TypeSPARQ, users can easily navigate and understand the schema of a SPARQL endpoint, thanks to its graphical interface that provides a visual representation of the endpoint's schema.

Additionally, TypeSPARQ enables users to visually select data structures or subsets of ontologies and directly generate a starter LDkit application from those selections. This includes the specification of the endpoint, data schemas, and Lens components necessary for the development.

As a result, using TypeSPARQ enables application developers to bootstrap their Linked Data applications easily, even without previous knowledge of RDF and related technologies. TypeSPARQ produces a ready-to-use starter template that provides an end-to-end type-safe access to selected data in the target SPARQL endpoint. This accessibility significantly lowers the barrier to entry for new developers in the field of Semantic Web and Linked Data, allowing them to focus more on application logic and less on the complexities of RDF data handling. This approach not only simplifies the initial setup process but also accelerates the development cycle, making it more approachable and manageable for developers of all skill levels.

### 7.2.5. Schema Forge

Schema Forge [74] is a low-code, pattern-driven schema engineering tool designed to facilitate the interactive exploration and authoring of RDF-based ontologies. It supports the ingestion of ontologies serialized in formats

---

<sup>16</sup><https://slovník.gov.cz>

<sup>17</sup><https://github.com/datagov-cz/sgov-assembly-line> is the umbrella repository that refers to the repositories of individual tools (in Czech)

<sup>18</sup><https://github.com/datagov-cz/ssp> (in Czech)

1 such as Turtle and JSON-LD, enabling users to navigate classes, properties, and relationships through a dynamic 1  
2 graphical interface, including visualization of ontology structures. 2

3 To enable dynamic retrieval of ontology elements, Schema Forge integrates LDkit for querying Linked Data 3  
4 resources. Specifically, it utilizes LDkit's querying capabilities to obtain class hierarchies and enumerate subclass 4  
5 relationships within the connected dataset, thereby ensuring type-safe and schema-aware interaction with RDF data. 5

6 While Schema Forge is currently in an early stage of development and lacks some features required for 6  
7 production-level deployment, it provides a functional foundation for ontology exploration and editing. The project 7  
8 appears to be under active development, with a publicly available roadmap indicating planned features and enhance- 8  
9 ments. 9

#### 10 7.2.6. LDE – Linked Data Engine 10

11 The Linked Data Engine (LDE) [71] is a modular suite of Node.js libraries that are designed to support the 11  
12 development and execution of Linked Data applications and data processing pipelines. 12

13 The project encompasses a broad set of functionalities, including dataset modeling, downloading dataset distribu- 13  
14 tions, and retrieving dataset descriptions from DCAT registries. It also facilitates the deployment of local SPARQL 14  
15 endpoints for testing purposes, as well as the import of data dumps into local SPARQL endpoints for querying. 15

16 LDE is currently under active development. A public roadmap outlines upcoming features, including a pipeline 16  
17 builder tool to query, transform and enrich Linked Data. 17

18 At present, LDE utilizes LDkit to query dataset descriptions from DCAT-AP 3.0 registries. The implementation 18  
19 includes a custom DCAT namespace, a dataset schema built upon this namespace, and leverages LDkit filtering and 19  
20 pagination capabilities. 20  
21

## 22 8. Future Work 22

23 While LDkit is a production-ready toolkit suitable for developing most common Linked Data-based applications, 23  
24 several aspects could be improved in future iterations. 24

25 First, LDkit is currently RDF graph-agnostic when reading and writing data. To enhance flexibility, we plan to 25  
26 introduce support for named graph constraints in LDkit resources, allowing users to specify particular graphs for 26  
27 their data. A key challenge in this implementation is determining how to handle resources that span multiple graphs. 27  
28 This raises the question whether to enforce graph constraints at the resource level or at the property level. 28  
29

30 Second, LDkit users would benefit from support for RDF containers and collections, enabling the retrieval of 30  
31 ordered or unordered lists of entities – a common requirement in application development. Since LDkit relies on 31  
32 SPARQL queries for data retrieval and updates, effective support for these RDF structures must first be implemented 32  
33 in SPARQL to ensure efficient integration into LDkit and similar tools. While SPARQL 1.1 introduced support for 33  
34 property paths, making it somewhat easier to work with ordered lists, it still leaves a lot to be desired. 34  
35

36 Finally, to support the development of user-friendly applications, LDkit must be able to query data fast. As previ- 36  
37 ously discussed, more complex data schemas result in more complex SPARQL queries, leading to longer execution 37  
38 times within query engines. Addressing this challenge is nontrivial, as SPARQL performance issues stem from 38  
39 graph-based data model, inefficient storage or indexing, and the computational cost of joins and path queries. Given 39  
40 these constraints, there are limits to the performance improvements that can be reasonably expected from SPARQL 40  
41 query engines in the future. Instead, performance optimizations could be implemented at the LDkit level by leverag- 41  
42 ing knowledge of the data schema. This could involve decomposing complex SPARQL queries into smaller queries, 42  
43 deferring their execution, or supporting the retrieval of partial results. 43  
44

## 45 9. Conclusion 45

46 Web application developers face considerable challenges, primarily due to the vast array of technologies they need 46  
47 to master. The addition of Linked Data into the web development mix introduces yet another layer of complexity 47  
48 with its distinct set of technologies and standards. This exacerbates the learning curve and can detract from a positive 48  
49 50  
51

development experience. Historically, the Semantic Web community has struggled to keep pace with the rapid evolution of application development standards, and there has been a notable scarcity of robust, production-ready frameworks or libraries that adequately support developers in integrating Semantic Web technologies smoothly. Consequently, despite the potential benefits of Semantic Web technologies, their adoption within the broader web development community remains limited.

LDkit 2.0 represents a major step forward in lowering the barrier to the adoption of Linked Data and Semantic Web technologies. 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 specifically designed to simplify the complexities associated with querying RDF data, while still providing developers complete control over semantic data mapping through its schema-based approach. By abstracting the intricacies of SPARQL queries and RDF data handling, LDkit allows developers to focus on the logic of their applications rather than getting bogged down by the underlying data structure complexities. The framework offers a schema-driven approach that enables precise control over how data is mapped and manipulated within applications. This means developers can define custom data models that directly correspond to their application needs, ensuring that the integration of semantic data is both seamless and efficient.

In conclusion, we believe that LDkit is a valuable contribution to the Linked Data community, providing a **comprehensive** abstraction of Linked Data technologies. Throughout this paper, we have presented evidence to support this claim, demonstrating how LDkit addresses specific web development needs and its utility in real-world scenarios. We have illustrated its impact by presenting both quantitative and qualitative insights from existing projects that currently utilize LDkit, including those that further simplify the use of Semantic Web technologies for developers. Based on this evidence, we are confident that LDkit will foster further adoption of Linked Data in web applications.

### Acknowledgements

This research was supported by SVV project number 260 698.

Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N)

### References

- [1] K. Alexander, R. Cyganiak, M. Hausenblas and J. Zhao, Describing Linked Datasets with the VoID Vocabulary, 2011. <https://www.w3.org/TR/void/>.
- [2] K. Angele, M. Meitinger, M. Bußjäger, S. Föhl and A. Fensel, Graphsparql: A GraphQL interface for linked data, in: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 778–785.
- [3] T. Bergwinkl, M. Luggen, elf Pavlik, B. Regalia, P. Savastano and R. Verborgh, RDF/JS: Data Model Specification, 2022. <https://rdf.js.org/data-model-spec/>.
- [4] T. Berners-Lee, J. Hendler and O. Lassila, The semantic web, *Scientific American* **284**(5) (2001), 34–43.
- [5] C. Bizer, T. Heath and T. Berners-Lee, Linked data - the story so far, *International Journal on Semantic Web and Information Systems* **5**(3) (2009), 1–22.
- [6] D.U. Board, DCMI Metadata Terms, DCMI, 2020.
- [7] D. Brickley and L. Miller, FOAF Vocabulary Specification, 2014.
- [8] L. Feigenbaum, G.T. Williams, K.G. Clark and E. Torres, SPARQL 1.1 Protocol, *W3C Recommendation* **21** (2013).
- [9] M. Fowler, *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*, Addison-Wesley, 2012.
- [10] P. Gearon, A. Passant and A. Polleres, SPARQL 1.1 Update, W3C, 2013.
- [11] H.G. González, DMAOG Paper examples and performance measurement scripts, 2025. <https://herminiogarcia.com/dmaog-paper-evaluation/>.
- [12] G. Guizzardi, A. Botti Benevides, C.M. Fonseca, D. Porello, J.P.A. Almeida and T. Prince Sales, UFO: Unified foundational ontology, *Applied Ontology* **17**(1) (2022), 167–210.
- [13] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C, 2013.
- [14] O. Hartig, An overview on execution strategies for Linked Data queries, *Datenbank-Spektrum* **13** (2013), 89–99.
- [15] T. Heath and C. Bizer, *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, pp. 1–136.
- [16] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres and S. Decker, An empirical survey of linked data conformance, *Journal of Web Semantics* **14** (2012), 14–44.
- [17] Karel Klíma, LDkit Ontology, 2025.

- [18] K. Klíma, R. Taelman and M. Nečaský, LDkit: Linked Data Object Graph Mapping Toolkit for Web Applications, in: *International Semantic Web Conference*, Springer Nature Switzerland, 2023, pp. 194–210. ISBN 978-3-031-47243-5. [https://doi.org/10.1007/978-3-031-47243-5\\_11](https://doi.org/10.1007/978-3-031-47243-5_11).
- [19] K. Klíma, M. Blaško, P. Křemen, M. Nečaský, M. Ledvinka, A. Binderová, M. Švagr and F. Kopecký, Assembly Line: a tool for collaborative modeling of ontologies in public administration, in: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, IEEE, 2023, pp. 24–29.
- [20] K. Klíma and D. Beeke, *kareklima/ldkit*: 1.0.0, Zenodo, 2023. doi:10.5281/zenodo.7905469.
- [21] K. Klíma and D. Beeke, *kareklima/ldkit*: 2.4.0, Zenodo, 2025. doi:10.5281/zenodo.15485481.
- [22] H. Knublauch and D. Kontokostas, Shapes Constraint Language (SHACL), 2017. <https://www.w3.org/TR/shacl/>.
- [23] J.-E. Labra-Gayo, A. Iglesias-Préstamo, D. Martín-Fernández and M.-A. Arnaud, *rudof*: A Rust Library for handling RDF data models and Shapes, in: *Proceedings of the Posters and Demos Track at the 23rd International Semantic Web Conference (ISWC 2024)*, CEUR Workshop Proceedings, Vol. 3828, Baltimore, USA, 2024. ISSN 1613-0073.
- [24] M. Ledvinka and P. Křemen, A comparison of object-triple mapping libraries, *Semantic Web* **11**(3) (2020), 483–524.
- [25] A. Meroño-Peñuela and R. Hoekstra, *grlc* Makes GitHub Taste Like Linked Data APIs, in: *The Semantic Web: ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 – June 2, 2016*, Springer, 2016, pp. 342–353. ISBN 978-3-319-47602-5. [https://doi.org/10.1007/978-3-319-47602-5\\_48](https://doi.org/10.1007/978-3-319-47602-5_48).
- [26] OpenAPI Initiative, OpenAPI Specification, 2025. <https://spec.openapis.org/oas/latest.html>.
- [27] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn and G. Tummarello, *Sindice.com*: A document-oriented lookup index for open linked data, *International Journal of Metadata, Semantics and Ontologies* **3**(1) (2008), 37–52.
- [28] D. Peterson, S. Gao, A. Malhotra, C.M. Sperberg-McQueen and H.S. Thompson, XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, 2012. <https://www.w3.org/TR/xmlschema11-2/>.
- [29] E. Prud'hommeaux, I. Boneva, J.E. Labra Gayo and G. Kellog, Shape Expressions Language (ShEx) 2.1, 2019. <https://shex.io/shex-antics/>.
- [30] Schema.org, Schema.org: Vocabulary, 2011, Accessed: yyyy-mm-dd.
- [31] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler and N. Lindström, JSON-LD 1.1, *W3C Recommendation, Jul* (2020).
- [32] Š. Stenclák, M. Nečaský, P. Škoda and J. Klímek, *Dataspecer*: A model-driven approach to managing data specifications, in: *European Semantic Web Conference*, Springer, 2022, pp. 52–56.
- [33] R. Taelman and J. Scazzosi, RDF/JS: Query Specification, 2023. <https://rdf.js.org/query-spec/>.
- [34] R. Taelman and R. Verborgh, Evaluation of Link Traversal Query Execution over Decentralized Environments with Structural Assumptions, *arXiv preprint arXiv:2302.06933* (2023).
- [35] R. Taelman, M. Vander Sande and R. Verborgh, GraphQL-LD: linked data querying with GraphQL, in: *ISWC2018, the 17th International Semantic Web Conference*, 2018, pp. 1–4.
- [36] R. Taelman, M. Vander Sande and R. Verborgh, Bridges between GraphQL and RDF, in: *W3C Workshop on Web Standardization for Graph Data. W3C*, 2019.
- [37] R. Taelman, J. Van Herwegen, M. Vander Sande and R. Verborgh, *Comunica*: a Modular SPARQL Query Engine for the Web, in: *Proceedings of the 17th International Semantic Web Conference*, 2018. <https://comunica.github.io/Article-ISWC2018-Resource/>.
- [38] R. Verborgh and R. Taelman, *LDflex*: a read/write linked data abstraction for front-end web developers, in: *The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II 19*, Springer, 2020, pp. 193–211.
- [39] Angular, 2024. <https://angular.io/>.
- [40] Apollo Client, 2024. <https://github.com/apollographql/apollo-client>.
- [41] Bun, 2024. <https://bun.sh/>.
- [42] Deno, 2024. <https://deno.com/>.
- [43] esm.sh, 2024. <https://esm.sh/>.
- [44] GraphQLAlchemy, 2024. <https://github.com/memgraph/qlalchemy>.
- [45] GraphQL, 2024. <https://graphql.org/>.
- [46] JSR - the JavaScript Registry, 2024. <https://jsr.io/>.
- [47] Knex.js, 2024. <https://knexjs.org/>.
- [48] LDkit Deno module, 2024. <https://deno.land/x/ldkit>.
- [49] LDkit NPM package, 2024. <https://www.npmjs.com/package/ldkit>.
- [50] LDkit source code on GitHub, 2024. <https://github.com/kareklima/ldkit>.
- [51] Linked Data Objects (LDO), 2024. <https://github.com/o-development/ldo>.
- [52] MongoDB Node.js Driver, 2024. <https://github.com/mongodb/node-mongodb-native>.
- [53] Mongoose, 2024. <https://mongoosejs.com/>.
- [54] Neo4j OGM, 2024. <https://github.com/neo4j/neo4j-ogm>.
- [55] Node.js, 2024. <https://nodejs.org/>.
- [56] NPM, 2024. <https://www.npmjs.com/>.
- [57] Object-semantic mapping (OSM), 2024. <https://github.com/doga/object-semantic-mapping>.
- [58] Preact, 2024. <https://preactjs.com/>.
- [59] Prisma, 2024. <https://www.prisma.io/>.
- [60] RDF Object, 2024. <https://github.com/rubensworks/rdf-object.js>.
- [61] React, 2024. <https://react.dev/>.

1	[62] SimpleRDF, 2024. <a href="https://github.com/simplerdf/simplerdf">https://github.com/simplerdf/simplerdf</a> .	1
2	[63] Solid Project, 2024. <a href="https://solidproject.org/">https://solidproject.org/</a> .	2
3	[64] Soukai Solid, 2024. <a href="https://github.com/NoelDeMartin/soukai-solid">https://github.com/NoelDeMartin/soukai-solid</a> .	3
4	[65] tRPC, 2024. <a href="https://github.com/trpc/trpc">https://github.com/trpc/trpc</a> .	4
5	[66] ts-node, 2024. <a href="https://www.npmjs.com/package/ts-node">https://www.npmjs.com/package/ts-node</a> .	5
6	[67] TypeORM, 2024. <a href="https://typeorm.io/">https://typeorm.io/</a> .	6
7	[68] TypeScript, 2024. <a href="https://www.typescriptlang.org/">https://www.typescriptlang.org/</a> .	7
8	[69] TypeSPARQ, 2024. <a href="https://github.com/Jkuzz/sparql-explorer">https://github.com/Jkuzz/sparql-explorer</a> .	8
9	[70] Vue, 2024. <a href="https://vuejs.org/">https://vuejs.org/</a> .	9
10	[71] Linked Data Engine (LDE), 2025. <a href="https://github.com/ldengine/ldengine">https://github.com/ldengine/ldengine</a> .	10
11	[72] Netwerk Digitaal Erfgoed (NDE), 2025. <a href="https://www.netwerkdigitaal erfgoed.nl/">https://www.netwerkdigitaal erfgoed.nl/</a> .	11
12	[73] Oxigraph, 2025. <a href="https://github.com/oxigraph/oxigraph">https://github.com/oxigraph/oxigraph</a> .	12
13	[74] Schema Forge, 2025. <a href="https://github.com/gongni220284/pp2">https://github.com/gongni220284/pp2</a> .	13
14	[75] SPARQL Editor, 2025. <a href="https://github.com/sib-swiss/sparql-editor">https://github.com/sib-swiss/sparql-editor</a> .	14
15	[76] The Netwerk of Terms, 2025. <a href="https://github.com/netwerk-digitaal-erfgoed/network-of-terms">https://github.com/netwerk-digitaal-erfgoed/network-of-terms</a> .	15
16		16
17		17
18		18
19		19
20		20
21		21
22		22
23		23
24		24
25		25
26		26
27		27
28		28
29		29
30		30
31		31
32		32
33		33
34		34
35		35
36		36
37		37
38		38
39		39
40		40
41		41
42		42
43		43
44		44
45		45
46		46
47		47
48		48
49		49
50		50
51		51