Java Binding for JSON-LD

Martin Ledvinka [a]
Department of Computer Science, Faculty of Electrical Engineering,
Czech Technical University in Prague, Technicka 2, Prague 6 - Dejvice, Czech Republic

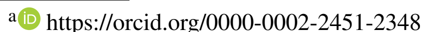
Keywords: JSON-LD, Java, REST API, Object Model.

Abstract: JSON-LD is an easy-to-understand and use data format with a Linked Data background. As such, it is one of the most approachable Semantic Web technologies. Moreover, it can bring major benefits even to applications not primarily based on the Semantic Web, especially regarding their interoperability. This work presents JB4JSON-LD – a software library allowing seamless integration of JSON-LD into REST APIs of Java Web applications without having to deal with individual nodes of the JSON-LD graph. The library is compared to existing alternatives and a demo application as well as a real-world information system are used to illustrate its use.

1 INTRODUCTION

JSON-LD (JSON for Linking Data) (Sporny et al., 2020) is an easy-to-use data serialization format. It is based on the widely popular JSON[1] format and, in essence, adds the possibility to connect information on how to interpret and identify the data to it (while staying compatible). Search engines such as Google recommend adding JSON-LD data to Web pages as it allows them to better understand the content of the page and enhance search results, ultimately improving the visit and interaction rate of the site.[2] Moreover, JSON-LD can be especially useful for information system interoperability because it helps to disambiguate the meaning of APIs and the data they work with (Bittner et al., 2006; Lanthaler and Gütl, 2012; Su et al., 2015; Xin et al., 2018).

Consider the plain JSON output of a terminology management tool shown in Listing 1. It could represent a detail of a term produced by the tool's REST API (Fielding, 2000). A term in this setting has a label, definition and can be put into a hierarchy based on its meaning.

[a]  https://orcid.org/0000-0002-2451-2348

[1] JavaScript Object Notation

[2] As described in the official documentation at https://developers.google.com/search/docs/appearance/structured-data/intro-structured-data, accessed 2023-09-14.

Listing 1: Example output of a terminology management tool in JSON.

```
{
  "id": "251",
  "label": "Locality",
  "definition": "...",
  "children": ["252", "253"]
}
```

For a consumer, this output has several issues that JSON-LD can fix:

1. The value of the `id` attribute (presumably identifier) is relevant only within the boundaries of the tool. Even worse, it is often relevant only within a certain table in the system's underlying database and the same identifier can be used for an instance of another type.
2. The meaning of the attributes can be ambiguous. One may, for example, question how the `label` attribute relates to the same attribute name in a music streaming service API (where a *record label* denotes a company that owns a particular piece of music). Similarly with the attribute `children`. It might be argued that this could be solved by better naming but such a solution often comes in hindsight when the damage is already done and applying it would break existing clients.
3. The string values by default do not contain any information about their language. Terminology management in particular often requires translations into multiple languages.
4. The output does not indicate the type of the object.

Ledvinka, M.
Java Binding for JSON-LD.
DOI: 10.5220/0012168500003584
In *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*, pages 207-214
ISBN: 978-989-758-672-9; ISSN: 2184-3252
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

Extending the JSON from Listing 1 with a JSON-LD context definition from Listing 2 considerably improves the semantic quality of the data.

Listing 2: Example JSON-LD context for a terminology management tool based on SKOS (Miles and Bechhofer, 2009).

```
{
  "id": "@id",
  "label": {
    "@id": "http://www.w3.org/2004/02/skos/core#prefLabel",
    "@container": "@language"
  },
  "definition": {
    "@id": "http://www.w3.org/2004/02/skos/core#definition",
    "@container": "@language"
  },
  "children": {
    "@id": "http://www.w3.org/2004/02/skos/core#narrower",
    "@type": "@id"
  }
}
```

Data modifications in Listing 3 further address the aforementioned problematic areas by changing the identifiers to IRIs[3] and adding type information.

Listing 3: Modified data sample from Listing 1 better utilizes JSON-LD features.

```
{
  "@context": {... defined in Listing 2 ...},
  "id": "http://onto.fel.cvut.cz/ontologies/slovník/datový/
       psp-2016/pojem/lokalita",
  "@type": "http://www.w3.org/2004/02/skos/core#Concept",
  "label": {
    "en": "Locality",
    "cs": "Lokalita"
  },
  "definition": {
    "en": "...",
    "cs": "..."
  },
  "children": [
    "http://onto.fel.cvut.cz/ontologies/slovník/datový/mpp
         -3.5-np/pojem/zastavitelná-lokalita",
    "http://onto.fel.cvut.cz/ontologies/slovník/datový/mpp
         -3.5-np/pojem/nezastavitelná-lokalita"
  ]
}
```

Considering the aforementioned issues with plain JSON:

1. Now the identifier is clearly denoted by the JSON-LD `@id` keyword. Moreover, if an IRI is used, it is globally valid and dereferencing it can give the client more information about the resource.
2. The added context unambiguously identifies properties corresponding to the attributes. The client can use these property identifiers to gather more information about the properties and possibly even restrictions on their values (for example, that a concept's children must be also concepts).
3. JSON-LD supports language-tagged strings out of the box.
4. JSON-LD allows classifying objects via the standard `@type` attribute.

Another useful feature of JSON-LD is the ability to reference objects only by their identifier. Plain JSON applications have to serialize an object multiple times if it is referenced in multiple places in the output. Or, they come up with ad hoc solutions that their client needs to know to be able to correctly interpret the output. In contrast, JSON-LD allows serializing the object only once and using its identifier to refer to it in all other places. This prevents issues with circular references and may reduce the size of the output and serialization time.

As can be seen, information systems may greatly benefit from supporting JSON-LD in their Web services, especially in conjunction with using an *ontology* to back the underlying domain model (Guarino et al., 2009; Lanthaler and Gütl, 2012). However, to be able to do so, developers need the right tools to integrate JSON-LD serialization and deserialization into the existing development stack. This paper introduces one such tool – JB4JSON-LD.

The remainder of this work is structured as follows: Section 2 introduces the library, including its features and architecture, Section 3 provides an overview of related work whereas Section 4 describes two demo applications. The paper is concluded in Section 5.

2 JB4JSON-LD

JB4JSON-LD (Java Binding for JSON-LD)[4] is a Java library for serializing and deserializing Java objects to/from JSON-LD. Its main goal is to provide JSON-LD capabilities to applications without requiring developers to work with JSON-LD directly. Instead, it relies on an application's object (domain) model and an explicit declarative mapping. This idea is based on the fact that information systems usually map data to an object model whose elements (classes) represent domain concepts like vocabulary, concept, or user (Booch, 1994). The system's API then translates between a data format such as JSON and instances of the object model classes. The goal of the library is to

[3]Internationalized Resource Identifiers

[4]https://github.com/kbss-cvut/jb4jsonld, accessed 2023-09-14.

allow a declarative description of this translation and not require the developer to implement the translation themselves. In the case of JB4JSON-LD, the mapping is specified via Java annotations.

2.1 Mapping with JOPA

To prevent the need to define a new set of annotations and because the library was (and still primarily is) used in applications that use the *JOPA* library for persistence (Křemen and Kouba, 2012),[5] JB4JSON-LD reuses the mapping specified by its annotations.

The example in Listing 4 shows a simple JOPA entity class representing a term with some basic attributes. To briefly explain:

- @OWLClass denotes the ontological class to which this entity class is mapped.
- @Id denotes the entity identifier.
- @OWLAnnotationProperty, @OWLObjectProperty, OWLDataProperty (not used in the example) denote mapping to ontological properties (Motik et al., 2012). Object properties are used to reference other objects, while data property values are literals. Annotation property values can be either.
- @Types are used to access additional classification besides the entity class's mapped type.

Listing 4: JOPA entity with annotation-based mapping. This mapping is utilized by JB4JSON-LD as well. MultilingualString is a JOPA class used to represent language-tagged strings.

```
@Namespace(prefix="skos",
    namespace="http://www.w3.org/2004/02/skos/core#")
@OWLClass(iri = "skos:Concept")
public class Term {
  @Id
  private URI id;

  @OWLAnnotationProperty(iri = "skos:prefLabel")
  private MultilingualString label;

  @OWLAnnotationProperty(iri = "skos:definition")
  private MultilingualString definition;

  @OWLObjectProperty(iri = "skos:narrower")
  private Set<Term> children;

  @Types
  private Set<String> types;

  // Other attributes, getters, setters
}
```

[5]Source code and documentation available at https://github.com/kbss-cvut/jopa, accessed 2023-09-14.

When serializing an object, JB4JSON-LD uses Java Reflection (Forman and Forman, 2004) to access its fields to get the values to serialize and annotations to determine the attributes to serialize them to. Conversely, when a JSON-LD input is deserialized, JB4JSON-LD determines the target Java class based on the input type and the requested Java class and, for each attribute of the input JSON object, looks for Java fields with a corresponding mapping.

Listing 2 in Section 1 shows a JSON-LD context corresponding to the Term class from Listing 4.

2.2 Features

This section highlights the most important features of JB4JSON-LD.

2.2.1 Multiple JSON-LD Forms

The JSON-LD specification defines multiple semantically equivalent syntactic forms of JSON-LD documents (Sporny et al., 2020). While the serialization output form of JB4JSON-LD is defined (compacted JSON-LD with/without context), to support as wide variety of input as possible, JB4JSON-LD does not restrict the form of the JSON-LD input it deserializes. Instead, it internally transforms the input to the expanded form (so that it always has the same structure) and then proceeds with deserialization.

2.2.2 Polymorphism Support

Polymorphism in programming is usually meant in the sense that a set of types share a common ancestor that can be used in their place (Booch, 1994). In the context of JB4JSON-LD, polymorphism is interesting for deserialization. In this case, the requested result type may have subtypes and JB4JSON-LD will determine which of these subtypes should be actually instantiated based on the input classification (this behavior is configurable).

Consider the class hierarchy in Figure 1. The application may request deserialization of type Resource. However, if the type of the JSON-LD object on input corresponds to Dataset, JB4JSON-LD will actually return an instance of Dataset. JSON libraries such as *Jackson*[6] support the same behavior. However, because plain JSON does not support object classification, it has to be configured in a library-specific way.

[6]Arguably the most popular JSON mapping library for Java. https://github.com/FasterXML/jackson, accessed 2023-09-14.

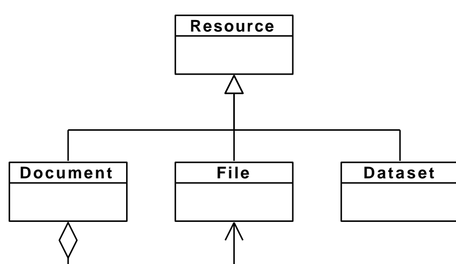


Figure 1: Simple hierarchy classes to showcase polymorphism.

2.2.3 Object References

As already mentioned in Section 1, JSON-LD allows referencing an object by its identifier on repeated encounters instead of always serializing it in full. JB4JSON-LD supports this standard feature out of the box. In contrast, plain JSON mapping libraries such as Jackson have to simulate this behavior by assigning identity to objects and referencing this identity when the object is encountered again. However, their clients have to be configured to handle this provider-specific implementation.

2.2.4 Multilingual Strings and Typed Values

Language-tagged strings are a feature JSON-LD offers to support internationalization of applications and data. JB4JSON-LD supports string internationalization by mapping the MultilingualString class, which represents strings with language tags (see Listing 2 for language-tagged string data and the corresponding entity class in Listing 4).

Typed values then allow extending the limited set of data types supported by JSON (boolean, number, string, array, object, null) by stating their lexical form together with a type declaration. A typical example would be temporal values such as date, time, and datetime, which can be naturally expressed in Java, but not in plain JSON. Listing 5 shows such a typed value (the type is declared the context object).

Listing 5: A short example of how typed values are represented in JSON-LD.

```
{
  "@context": {
    "created": {
      "@id": "http://purl.org/dc/terms/created",
      "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
    },
  },
  "created": "2023-06-06T04:00:00.000Z"
}
```

2.2.5 Custom Serializers and Deserializers

When the built-in serialization and deserialization capabilities of JB4JSON-LD are not enough or not suitable for a particular use case, it is possible to register custom serializers and deserializers the library should use instead. When invoked, these custom components receive an object representing the current serialization/deserialization context and are expected to return the appropriate result.

2.3 Architecture

The architecture of JB4JSON-LD decouples JSON-LD production and ingestion from serialization and deserialization. The goal is to prevent client applications from being locked to a single JSON processing library (such as the aforementioned Jackson). Integration modules can be implemented for different JSON processing libraries. Figure 2 illustrates the component structure of the library and its Jackson integration module as well as the flow of the data.

Serialization expects the library integration to implement a generator of the JSON string. The serializer traverses the specified object graph, using an implementation of the *Visitor* pattern (Gamma et al., 1995) to build a representation of a JSON tree that is then written out via the provided JSON generator. The serialization algorithm thus makes two traversals – one transforms the input object graph to an abstract representation of a JSON tree and the other writes out this abstract representation into actual JSON.

Deserialization assumes that the integration will provide it with a Java representation of the input JSON-LD in the expanded form (Sporny et al., 2020). This means that all values are represented by lists, and objects are maps where keys are property identifiers. Deserialization is then a matter of a single traversal.

Both serialization and deserialization keep track of already visited objects so that there is no risk of infinite recursion on circular references as well as the possibility of reusing the object references.

3 RELATED WORK

JSON-LD is not the only approach attempting to extend the capabilities of JSON. JSON Schema[7] is a language allowing annotation and validation of JSON documents. While it can be used to describe the structure of the data, its main use is in its validation (Pezoa et al., 2016). In a sense, JSON Schema is to JSON

[7] https://json-schema.org/, accessed 2023-09-14.

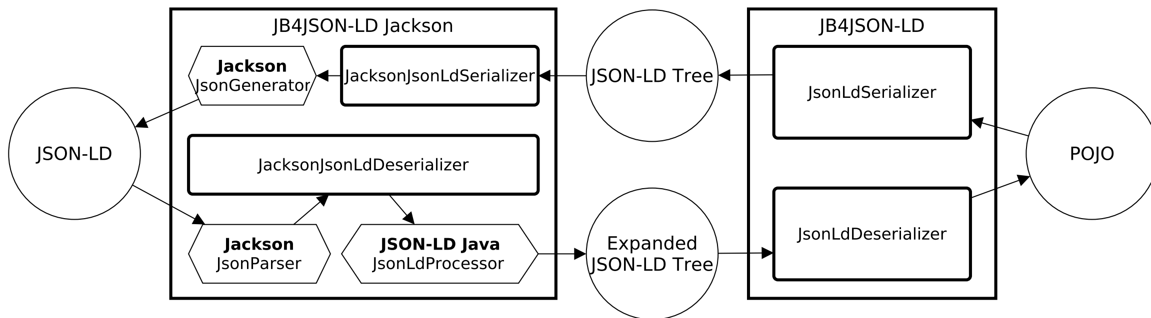


Figure 2: A simplified diagram of the architecture of JB4JSON-LD and its Jackson integration module. Items with bold border represent parts of the library, circles represent data artifacts, hexagons represent components of external libraries. The arrows illustrate the flow of data. Element size does not indicate complexity.

what SHACL (Kontokostas and Knublauch, 2017) is to RDF. JSON Schema, with certain extensions, has also been proposed as an ontology modeling language (Angele and Angele, 2021).

However, the main focus of this work is JSON-LD. There are multiple ways of using JSON-LD in an application. Since JB4JSON-LD is a Java library, the primary focus of the following will be on Java.

The most basic way of processing JSON-LD in an application is viewing it as yet another serialization of RDF (Cyganiak et al., 2014). In this case, existing RDF-access libraries like *Jena* (Carroll et al., 2004) or *RDF4J* (Broekstra et al., 2002) (and their equivalents in other programming languages, such as *RDFLib*[8] in Python) can be used to produce and consume JSON-LD. Nevertheless, such libraries are arguably not suitable for building a domain-specific Web service API because their primary purpose is general data access and storage.

A more high-level approach would be to use JSON-LD manipulation libraries such as *JSONLD-Java*[9] or *Titanium*[10] to map domain objects to nodes in a JSON-LD graph. JSONLD-Java supports (at the time of writing this paper) version 1.0 of the JSON-LD specification, whereas Titanium already supports version 1.1 (Sporny et al., 2020). Again, other programming languages have similar solutions, for example, *json-ld.net*[11] in C#, *json-ld*[12] in Ruby or *PyLD*[13] in Python.

Using the aforementioned libraries for mapping to/from domain objects still requires rather a lot of boilerplate code. Instead, one would like to declaratively describe the mapping and use a serializer/deserializer to automatically transform the data to/from JSON-LD. There exist several solutions besides the one described in this paper. *jackson-jsonld*[14] is a module for working with JSON-LD for the Jackson library and it was actually the original inspiration for JB4JSON-LD. It defines its own set of mapping annotations and is easy to integrate into an application. However, the last commit in the library Git repository is from September 2017, indicating that it may not be maintained anymore. Another similar library is *hydra-java*[15]. hydra-java is a part of a larger ecosystem of modules for enriching Web applications with machine-readable hypermedia based on the Hydra vocabulary (Lanthaler, 2021). The modules allow generating API responses containing data in JSON-LD as well as links to other relevant endpoints and data in accordance with the HATEOAS[16] principles (Fielding, 2000). The latest commit in the library repository is from March 2019. *calamus*[17] in Python offers similar functionality, where mapping is declared via special classes called schemas. *Pinto*[18] is a generic mapper of Java objects to/from RDF. It uses the RDF4J Model API for output and input, so it supports most RDF syntax formats, including JSON-LD. Its integration into an application's REST API is, due to the lack of integration with Jackson or similar libraries, slightly more complicated. Unfortunately, the last

[8]https://rdflib.dev/, accessed 2023-09-14.

[9]https://github.com/jsonld-java/jsonld-java, accessed 2023-09-14.

[10]https://github.com/filip26/titanium-json-ld, accessed 2023-09-14.

[11]https://github.com/linked-data-dotnet/json-ld.net, accessed 2023-09-14.

[12]https://github.com/ruby-rdf/json-ld, accessed 2023-09-14.

[13]https://github.com/digitalbazaar/pyld, accessed 2023-09-14.

[14]https://github.com/io-informatics/jackson-jsonld, accessed 2023-09-14.

[15]https://github.com/dschulten/hydra-java, accessed 2023-09-14.

[16]Hypermedia as the Engine of Application State

[17]https://github.com/SwissDataScienceCenter/calamus, accessed 2023-09-14.

[18]https://github.com/stardog-union/pinto, accessed 2023-09-14.

commit in Pinto's repository is from December 2019, indicating that it is yet another abandoned project.

3.1 Comparison

This section compares all the high-level JSON-LD mapping libraries – JB4JSON-LD (presented by this paper), jackson-jsonld, hydra-java, Pinto and calamus – in terms of the features described in Section 2.2, plus any additional relevant capabilities provided by any of the libraries. An overview of the comparison is shown in Table 1, the following text provides details.

3.1.1 Multiple JSON-LD Forms

jackson-jsonld and hydra-java serialize objects to compacted JSON-LD with context. hydra-java is able to read only data in the same form. In contrast, jackson-jsonld is able to deserialize also expanded JSON-LD and compacted JSON-LD without context (JSON attributes using full property IRIs). Pinto relies on the serialization capabilities of the underlying RDF4J implementation, which produces expanded JSON-LD by default. Thanks to the reliance on RDF4J, it is also able to read any valid form of JSON-LD input for deserialization. Similarly, calamus also outputs JSON-LD without context and is able to deserialize various forms of JSON-LD.

3.1.2 Polymorphic Deserialization

None of the other tested libraries are able to utilize polymorphism when deserializing JSON-LD input. JB4JSON-LD not only supports this feature, but also allows configuring its precise behavior (whether supertypes or subtypes should be preferred).

3.1.3 Object References

hydra-java supports object references both in serialization and deserialization. However, this behavior has to be configured by using the Jackson JsonIdentityInfo annotation.[19] Without it, hydra-java fails to serialize circular object references with a stack overflow exception. jackson-jsonld suffers from the same issue, but using JsonIdentityInfo disables JSON-LD serialization completely and the output is in plain JSON. Neither Pinto nor calamus are able to handle circular references and produce a stack overflow exception both for serialization and deserialization.

[19] https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations#object-references-identity, accessed 2023-09-14.

3.1.4 Multilingual Strings, Typed Values

jackson-jsonld, hydra-java, and calamus support neither multilingual strings nor typed values. One interesting discovery is that jackson-jsonld actually does not handle the JSON-LD `@type` attribute and, without instructing Jackson to ignore unknown properties, deserialization fails with `@type` not being recognized as a known attribute. Pinto does support datatype specification for attributes, but does not support language-tagged strings.

3.1.5 Custom Serializers and Deserializers

jackson-jsonld and hydra-java themselves do not support custom (de)serializers. However, since both libraries are based on Jackson, which does support (de)serializer customization, it is possible to overcome this deficiency. Nevertheless, such an approach does not, for example, provide access to the current JSON-LD context and cannot thus be considered a fully implemented feature. Pinto supports custom (de)serializers via the so-called *codecs* – classes that provide both serialization and deserialization of values of a specific type. As far as it was possible to determine from the rather scarce documentation of calamus, it does not support custom serializers and deserializers.

3.1.6 Additional Features

JB4JSON-LD, hydra-java, and Pinto allow exposing Java enum constants as resources with their own IRIs. This can be seen as a representation of the OWL `ObjectOneOf` construct (Motik et al., 2012).

JB4JSON, jackson-jsonld, and hydra-java are also able to handle context term conflicts, i.e., two attributes of the same name (in different Java classes) mapped to different properties. Such situations are handled by utilizing JSON-LD embedded contexts (Sporny et al., 2020). Pinto and calamus do not use the JSON-LD context for serialization at all, so no term conflicts can occur. On the other hand, not using context makes them incompatible with clients that support only plain JSON.

calamus provides one unique feature (among the other compared libraries) – it allows validating the serialization output/deserialization input against the specified ontology. If the data are not valid, it provides a report indicating the offending properties.

Table 1: Overview of the JSON-LD library comparison. Criteria are mainly based on the features discussed in Section 2.2. Each criterion can have a value of ✓ for satisfaction, or × for dissatisfaction.

Feature	JB4JSON-LD	jackson-jsonld	hydra-java	Pinto	calamus
Multiple JSON-LD forms	✓	✓	×	✓	✓
Polymorphic deserialization	✓	×	×	×	×
Object references	✓	×	✓	×	×
Multilingual strings, typed values	✓	×	×	×✓	×
Custom (de)serializers	✓	×	×	✓	×
Enum mapping	✓	×	✓	✓	×
Embedded contexts	✓	✓	✓	×	×

4 DEMO

There are multiple Web applications utilizing JB4JSON-LD. This paper introduces two of them – a simple demo showcasing the library and a real-world information system.

4.1 JOPA JSON-LD

JOPA JSON-LD[20] is one of several demo projects associated with JOPA. This particular application is a trivial example derived from a medical trial management tool developed at the author's research group (Klíma, 2018). It demonstrates, besides several other JOPA features, the ability to easily create a JSON-LD-compatible application REST API with the help of JB4JSON-LD.

The demo shows that both JSON and JSON-LD can be simultaneously supported by an application using JB4JSON-LD, the appropriate format selected via HTTP content negotiation (Fielding, 2000). The configuration is trivial and can be found in the WebAppConfig class. Instructions on how to run the demo are available in the linked GitHub repository.

4.2 TermIt

TermIt[21] (Ledvinka et al., 2020) is a SKOS (Miles and Bechhofer, 2009)-compatible terminology manager built with Semantic Web technologies – one of them being JSON-LD. Similarly to the aforementioned demo application, TermIt's REST API supports both JSON and JSON-LD, and JSON-LD is actually used in communication between TermIt's frontend, written in TypeScript, and the backend. Nevertheless, other applications integrated with TermIt may (and often do) choose to use JSON rather than JSON-LD as a data format.

Experience shows that while for singular objects the JSON-LD context adds a slight data overhead, for more complex data structures (like hierarchies of terms), JSON-LD is able to decrease the amount of transferred data thanks to using only identifiers instead of serializing a single object multiple times. Support for polymorphism also greatly simplifies data processing both on client and server side (multiple kinds of resources use the same general logic).

A demo instance of TermIt is available online[22] and credentials *demo/demo* can be used to log into the application.

5 CONCLUSIONS

This paper has presented JB4JSON-LD, a library for mapping Java objects to/from JSON-LD based on an explicit mapping using annotations. It has been argued that JSON-LD can bring major benefits to Web application APIs, especially concerning their interoperability, and that JB4JSON-LD is a library that allows adding support for this data format easily.

A set of the most important features was described and a comparison with similar software libraries w.r.t. these features was provided. This comparison showed that JB4JSON-LD is the only library to cover of them. Two demo applications – one a simple, dedicated example, the other a real-world information system – were introduced to showcase the library's potential.

One of the main future tasks of JB4JSON-LD development is adding support for the JSON-LD 1.1 standard, because the current version is tied to JSON-LD 1.0 due to its reliance on JSONLD-Java. In addition, providing more powerful and precise ways to customize the serialization/deserialization process are

[20] https://github.com/kbss-cvut/jopa-examples/tree/master/jsonld, accessed 2023-09-14.

[21] Source code available at https://github.com/kbss-cvut/termit, accessed 2023-09-14.

[22] https://kbss.felk.cvut.cz/termit-demo, accessed 2023-09-14.

also planned, for example, per-attribute string literal format configuration support.

ACKNOWLEDGEMENTS

The author would like to thank Dr. Petr Křemen for his invaluable feedback.

REFERENCES

Angele, K. and Angele, J. (2021). JSON towards a simple Ontology and Rule Language. In Soylu, A., Nezhad, A. T., Nikolov, N., Toma, I., Fensel, A., and Vennekens, J., editors, *Proceedings of the 15th International Rule Challenge, 7th Industry Track, and 5th Doctoral Consortium @ RuleML+RR 2021 co-located with 17th Reasoning Web Summer School (RW 2021) and 13th DecisionCAMP 2021 as part of Declarative AI 2021 Leuven, Belgium (virtual due to Covid-19 pandemic), 8 - 15 September, 2021*. CEUR-WS.org.

Bittner, T., Donnelly, M., and Winter, S. (2006). *Ontology and Semantic Interoperability*, chapter 6. CRC Press, Boca Raton, Florida.

Booch, G. (1994). *Object-oriented Analysis and Design with Applications (2nd Ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: Implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference (Alternate Track Papers & Posters)*, pages 74–83.

Cyganiak, R., Wood, D., and Lanthaler, M. (2014). RDF 1.1 Concepts and Abstract Syntax. W3C recommendation, W3C. http://www.w3.org/TR/rdf11-concepts/, accessed 2023-09-14.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Forman, I. R. and Forman, N. (2004). *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Guarino, N., Oberle, D., and Staab, S. (2009). *What Is an Ontology?*, chapter 1, pages 1–17. Springer-Verlag Berlin Heidelberg.

Klíma, T. (2018). Semantic manager for prospective clinical trials. B.S. thesis, Czech Technical University in Prague. https://dspace.cvut.cz/handle/10467/76531?locale-attribute=en, accessed 2023-09-14. The thesis is in Czech.

Kontokostas, D. and Knublauch, H. (2017). Shapes constraint language (SHACL). W3C recommendation, W3C. https://www.w3.org/TR/shacl/, accessed 2023-09-14.

Křemen, P. and Kouba, Z. (2012). Ontology-Driven Information System Design. *IEEE Transactions on Systems, Man, and Cybernetics: Part C*, 42(3):334–344.

Lanthaler, M. (2021). https://www.hydra-cg.com/spec/latest/core/. W3C draft, Hydra W3C Community Group. https://www.hydra-cg.com/spec/latest/core/, accessed 2023-09-14.

Lanthaler, M. and Gütl, C. (2012). On Using JSON-LD to Create Evolvable RESTful Services. In *Proceedings of the Third International Workshop on RESTful Design*, WS-REST '12, page 25–32, New York, NY, USA. Association for Computing Machinery.

Ledvinka, M., Křemen, P., Saeeda, L., and Blaško, M. (2020). TermIt: A Practical Semantic Vocabulary Manager. In *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 1: ICEIS*, pages 759–766, Setúbal, Portugal. INSTICC, SciTePress.

Miles, A. and Bechhofer, S. (2009). SKOS Simple Knowledge Organization System Reference. W3C Recommendation, W3C. http://www.w3.org/TR/skos-reference, accessed 2023-09-14.

Motik, B., Parsia, B., and Patel-Schneider, P. F. (2012). OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C recommendation, W3C. https://www.w3.org/TR/owl2-syntax/, accessed 2023-09-14.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 263–273, Republic and Canton of Geneva, CHE. International World Wide Web Conferences Steering Committee.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Champin, P.-A., and Lindström, N. (2020). JSON-LD 1.1 A JSON-based Serialization for Linked Data. W3C Recommendation, W3C. https://www.w3.org/TR/json-ld11/, accessed 2023-09-14.

Su, X., Riekki, J., Nurminen, J. K., Nieminen, J., and Koskimies, M. (2015). Adding semantics to internet of things. *Concurrency and Computation: Practice and Experience*, 27(8):1844–1860.

Xin, J., Afrasiabi, C., Lelong, S., Adesara, J., Tsueng, G., Su, A. I., and Wu, C. (2018). Cross-linking BioThings APIs through JSON-LD to facilitate knowledge exploration. *BMC Bioinformatics*, 19(30).