JOPA: Efficient Ontology-based Information System Design

Martin Ledvinka[1] and Bogdan Kostov[1] and Petr Křemen[1]

Czech Technical University in Prague, Czech Republic
martin.ledvinka@fel.cvut.cz, bogdan.kostov@fel.cvut.cz,
petr.kremen@fel.cvut.cz

Abstract. Creating applications on top of linked data and ontologies brings many difficulties. The applications are either generic (and thus not appealing to end-users), or bound to ontology structure, change of which breaks the application. We present JOPA, a tool that formalizes the contract between the application and the ontology, combining advantages of both worlds. JOPA is a persistence framework for Java applications, providing formalized object-ontological mapping, transactions, access to multiple repository contexts, and producing linked data. The system is demonstrated on a real use-case of a reporting tool that we develop for the aviation industry.

1 Introduction

Keeping object-oriented applications aligned with underlying ontology commitments is a challenging task. Various ontology access libraries have been introduced over the last decade (these are briefly discussed in Section 2.1). Basically, developers either use too low-level API that is verbose and hard to use and maintain, or a high-level API that tries to map ontology structure to the object model which is necessarily lossy and dependent on the ontology structure.

We introduce the Java OWL Persistence API (JOPA), a tool that benefits from both approaches – it provides an object-ontological mapping, but also constructs to access all property values as well as inferred property values in an analogous manner. Furthermore, to allow easy maintenance of the application access to evolving ontologies, a formal contract between an object-oriented application and the ontology is set up. The formal contract consists of a set of integrity constraints describing the fixed part of the ontology relevant for the application, as introduced in [9]. An advantage of this explicit contract is that it allows rechecking ontology compliance with the application upon data update (a third-party change), i.e. *before* the application itself tries to access the data. Integrity constraint violation signalizes to the application designer the need for formal contract redesign (or even ontology redesign).

This demo shows the features of JOPA on a simplified application for aviation safety reporting. The full application is designed and implemented in cooperation with several stakeholders in the Czech aviation industry, including Air Navigation Services of the Czech Republic, or Prague Airport, for their future use.

2 Ontology Access using JOPA

Let us briefly discuss first the JOPA framework itself and then delve into description of the example application and how it uses JOPA's features.

2.1 Application Access to Ontologies

There are two main approaches to application access to ontologies.

A *generic* one, where data in ontologies are manipulated without any assumptions about their nature. Such approach is represented for example by OWL API [7] or Sesame API [2]. This approach is suitable mostly for generic applications like ontology editors, because its use for domain-specific business logic requires a lot of boilerplate code.

A *domain-specific* approach to ontology access makes use of *object-ontological mapping* (OOM), which maps ontological constructs to concepts of the object-oriented paradigm. OOM enables the application to be written in object-oriented style, which is by far the most widespread programming paradigm nowadays. Frameworks exploiting OOM are for example Empire [5] or AliBaba [1].

More thorough discussion of both approaches can be found in [9] or [11].

2.2 JOPA features

JOPA tries to take the best of both the *domain-specific* and *generic* approaches. It employs a formally defined object-ontological mapping, while providing a (limited) access to the more dynamic aspects of ontologies. Let us now briefly describe the main distinguishing features of JOPA. More detailed explanation of its architecture and features can be found in [9], [10] and [11].

Formal OOM In contrast to ad hoc mapping used by Empire or AliBaba, the object-ontological mapping in JOPA is based on a formally defined contract between the ontology and the object model. This contract is described by a set of OWL *integrity constraints* [13], which provide a closed-world view of a part of otherwise open-world assuming ontology. The OOM does not attempt to provide a complete mapping of OWL to Java, so for example only named classes and properties are supported.

Explicit inferred knowledge JOPA provides explicit access to inferred knowledge in the object model. Inferred statements cannot be treated as asserted ones on the object level, because they cannot be directly changed. Therefore, JOPA enables the developer to explicitly mark attributes as inferred, which means they may contain inferred knowledge and are thus read-only.

Types and properties Besides mapping properties to attributes, JOPA also provides access to the more dynamic parts of the ontology. Namely, every instance can contain a set of ontological types (`@Types` field), to which the individual represented by this instance belongs. It can also contain a map of property values, which are not mapped by the object model. This gives, although limited, access to the ontological structure which is not directly compiled into the object model.

Separate storage access By separating the actual storage access into the *OntoDriver* layer, JOPA enables the application to easily switch between different storages. Such change thus comprises merely modifying a few lines in a configuration file. Similarly, Empire [5] uses pluggable storage access components.

JPA features JOPA was inspired by the JPA specification [8] for object-relational mapping in Java. As such, it supports transactional processing, caching, cascading. JOPA also supports executing SPARQL [6] and SPARQL Update [3] statements and mapping their results directly to entities. While the API of JOPA is inspired by JPA, it is not exactly the same. This is because it tries to take into account features specific to ontologies, like contexts and support for types and unmapped properties. Empire, on the other hand, goes even further and does actually implement a subset of the JPA specification.

Contexts Some ontological storages support the notion of RDF *named graphs*, which enable data to be further structured. JOPA enables the application to exploit this feature both on object and attribute level.

2.3 Demo Application

The demo application showcases all of the features described in Section 2.2. The application is build for a use case in aviation safety. When a safety manager/aviation authority performs a safety audit, a checklist of several questions guides him/her through the audit agenda. The questions are linked to expected answers and whenever the actual answer does not match the expected one, it signalizes a possible safety issue.

The audit scenario is only a small part of a much larger field of aviation safety, which we are currently tackling in one of our projects[1]. The whole domain is described by a documentation ontology, which is based on the *unified foundational ontology* (UFO) [4].

For the purposes of our application, we create a set of integrity constraints [13], which restricts a part of the documentation ontology in order to make it suitable for an object-oriented application.

In the demo, a user can create audits, which are documented by reports. Every report contains a set of records, which are question-answer pairs. The records can be classified to express whether the answer was satisfactory or not.

From Ontology to Object Model To give a glimpse of the design process, take for example the portion of the documentation ontology $\mathcal{O}_D$ shown in Table 1. A set $\mathcal{S}_{IC}$ of integrity constraints provides a closed-world view on $\mathcal{O}_D$ for the purpose of our application. When an integrity constraint is violated, the ontology becomes incompatible with the application.

[1] For JOPA, this is actually its second deployment. An early prototype was used in a tool called *StruFail* in the domain of structural failures of buildings.

$$
\begin{array}{r|l}
\mathcal{O}_D = \{ Event \sqsubseteq Entity, & \mathcal{S}_{IC} = \{ Report \sqsubseteq \forall documents \cdot Event, \\
Report \sqsubseteq Entity, & \quad Report \sqsubseteq (= 1\, documents), \\
Person \sqsubseteq Agent, & \quad Report \sqsubseteq \forall hasAuthor \cdot Person, \\
\top \sqsubseteq \forall hasAuthor \cdot Agent, & \quad Report \sqsubseteq (= 1\, hasAuthor), \\
Report \sqsubseteq \exists documents \cdot Entity, & \quad Report \sqsubseteq \exists documents \cdot Entity, \\
documents \equiv isDocumentedBy^{-} \} & \quad Audit \sqsubseteq \forall isDocumentedBy \cdot Report \}
\end{array}
$$

Table 1. $\mathcal{O}_D$ represents an excerpt of the documentation ontology used in the demo application. $\mathcal{S}_{IC}$ depicts a set of OWL integrity constraints used as a contract between the application (its object model) and the ontology.

Based on $\mathcal{O}_D$ and $\mathcal{S}_{IC}$, transformation to the object-oriented paradigm yields a model shown in Figure 1. The actual object model is generated from the integrity constraints by the *OWL2Java* tool, which is a part of JOPA.

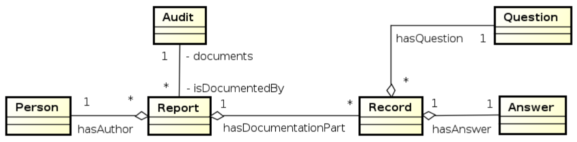

Fig. 1. Object model of the demo application. Due to space restrictions, $\mathcal{O}_D$ and $\mathcal{S}_{IC}$ capture only the Audit – Report – Person part of the model.

Demo Application Overview [2]

To list all audits and reports, a SPARQL query is used, whose results are directly mapped to the corresponding entities. While every report is related to an audit by an explicit assertion, the inverse relation is inferred, as it is not necessary to maintain both directions in the relationship. All operations on reports are cascaded to the records they contain, so for example when a report is persisted, all its records are persisted automatically as well. The same holds for the record-answer relationship. Questions are managed separately, because they can be reused by multiple reports.

Record classification is performed by adding the record individuals into OWL classes using the `@Types` field. In addition to the mapped attributes, every audit and record can also be enhanced with values of unmapped properties.

The demo application supports two storages - a Sesame storage and OWL files accessed by OWL API. The Sesame storage supports contexts, which is utilized by having the reports' authors stored in a dedicated context. The OWL API storage, on the other hand, is used by Pellet [12] to provide additional inferred knowledge. In our instance, it enables to show reports for each audit by exploiting the inverse *isDocumentedBy* property.

[2] The demo application can be found at http://onto.fel.cvut.cz/eswc2016, its source codes are available at https://github.com/kbss-cvut/jopa-examples.

3 Conclusions

We have discussed the difficulties of application access to ontologies and presented the JOPA framework as a possible solutions to these issues. We have demonstrated its viability as a persistence solution for ontology-based applications on a simplified demo application (a much more complex version of which is currently being evaluated by project partners in the Czech Republic), which nonetheless exploits most of the distinguishing features of JOPA.

Development of the aviation safety application has also shown us some shortcomings of JOPA, mainly its lack of support for OWL class subsumption (inheritance) and referential integrity. We plan to address these in our future work.

Acknowledgment This work was supported by grant No. GA 16-09713S Efficient Exploration of Linked Data Cloud of the Grant Agency of the Czech Republic and by grant No.SGS16/229/OHK3/3T/13 Supporting ontological data quality in information systems of the Czech Technical University in Prague.

References

1. AliBaba. online, `https://bitbucket.org/openrdf/alibaba/`
2. Broekstra, J., Kampman, A., van Harmelen, F.: 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. pp. 54–68 (2002)
3. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update. Tech. rep., W3C (2013)
4. Giancarlo Guizzardi: Ontological Foundations for Structural Conceptual Models. Ph.D. thesis, University of Twente (2005)
5. Grove, M.: Empire: RDF & SPARQL Meet JPA. semanticweb.com (April 2010), `http://semanticweb.com/empire-rdf-sparql-meet-jpa_b15617`
6. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. Tech. rep., W3C (2013)
7. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web – Interoperability, Usability, Applicability (2011)
8. JCP: JSR 317: Java^{TM} Persistence API, Version 2.0 (2009)
9. Křemen, P., Kouba, Z.: Ontology-Driven Information System Design. IEEE Transactions on Systems, Man, and Cybernetics: Part C 42(3), 334–344 (May 2012), `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6011704`
10. Ledvinka, M., Křemen, P.: JOPA: Developing Ontology-Based Information Systems. In: Proceedings of the 13th Annual Conference Znalosti 2014 (2014)
11. Ledvinka, M., Křemen, P.: JOPA: Accessing Ontologies in an Object-oriented Way. In: Proceedings of the 17th International Conference on Enterprise Information Systems (2015)
12. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (June 2007)
13. Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity Constraints in OWL. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010), `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1931`