Anno4j - Idiomatic Access to the W3C Web Annotation Data Model

Emanuel Berndl(✉), Kai Schlegel, Andreas Eisenkolb, Thomas Weißgerber, and Harald Kosch

Chair of Distributed Information Systems,
University of Passau, Innstraße 43, 94032 Passau, Germany
{emanuel.berndl,kai.schlegel,andreas.eisenkolb,
thomas.weissgerber,harald.kosch}@uni-passau.de

Abstract. The Web Annotation Data Model proposes standardised RDF structures to form "Web Annotations". These annotations are used to express metadata information about digital resources and are designed to be shared, linked, tracked back, as well as searched and discovered across different peers. Although this is an expressive and rich way to create metadata, there exists a barrier for non-RDF and SPARQL experts to create and query such information. We propose Anno4j, a Java-based library, as a solution to this problem. The library supports an Object-RDF mapping that enables users to generate Web Annotations by creating plain old Java objects - concepts they are familiar with - while a path-based querying mechanism allows comprehensive information querying. Anno4j follows natural object-oriented idioms including inheritance, polymorphism, and composition to facilitate the development. While supporting the functionality of the Web Annotation Data Model, the library is implemented in a modular way, enabling developers to add enhancements and use case specific model alterations. Features like plugin functionality, transactions, and input/output methods further decrease the boundary for non-RDF experts.

Keywords: Semantic Web · Linked Data · Web Annotations · Java · Developer tool

1 Introduction

Annotating things in the web is more and more common and a desired feature in the internet of today. Users support comments to various media, tag people on pictures, support helpful links to different topics, and so on. Often times, this is done using RDF, the Resource Description Format [4], the de facto standard for interlinking resources in the Semantic Web. Therefore, the Web Annotation Data Group[1] recently issued a new version of the Web Annotation Data Model WADM [8] (derived from the Open Annotation Data Model OADM [7]).

[1] https://www.w3.org/annotation/.

H. Sack et al. (Eds.): ESWC 2016 Satellite Events, LNCS 9989, pp. 257–270, 2016.
DOI: 10.1007/978-3-319-47602-5_43

The central concept of their model is the **Annotation**, being used as a way to give further details, description, or information about another "thing", in general digital resources. Examples could be a comment or tag on a web page or image, or a blog post about a news article. One annotation is devised out of three core components: the **body**, which contains the actual content of the annotation, the **target** specifying the "thing" that the annotation is about, and an **annotation node** itself, which joins the body and the target, while supporting provenance information of the whole annotation. This splits the content from its context in a modular way, which fosters the fundamentals of the annotation model: the designed annotations are interoperable and can be used and interpreted at various different locations, offering the possibility to connect information and metadata beyond the boundaries of single enterprises, data silos, and/or applications. The resulting combined knowledge base increases the benefit as well as the degree of information for every participant.

This aligns heavily with the "purpose" of the Semantic Web, with its promising advantages of combined and interlinked data. However, there exists an initial hurdle to make oneself familiar with the Semantic Web technologies as well as Linked Data "rules" [2]. Often times, developers are very skilled in their own respective programming domain, but lack the knowledge of producing RDF and consuming Linked Data over SPARQL.

To overcome some of these shortcomings and lower the barrier of Semantic Web technologies, Anno4j[2] provides a Java library to directly map Java objects to and from the Web Annotation Data Model. The core contributions are as follows:

- An Object-relational-like mapping (ORM) provides idiomatic access to W3C Web Annotation Data Model following natural object-oriented idioms. The use of RDF is enhanced by the advantages and assets of Java.
- Support for use-case specific Web Annotation Data Model alterations and extensions.
- A library built on-top of OpenRDF Alibaba[3], allowing for a broad field of application.
- Path-based query criteria for extensive annotation search functionality.
- Developer-friendly Open-Source Apache V2 license.

The remaining paper is structured as follows. Section 2 briefly lists the core features of the library and Sect. 3 highlights related work in the context of the WADM and its applications. Sections 4, 5, and 6 will give more detailed information about persistence, querying, and insights about the internals of Anno4j. In Sect. 7, convenience and RDF features that have been added to the library are enlisted. Section 8 shows a use case of Anno4j in the MICO[4] project. Finally, Sect. 9 will conclude the paper by depicting future work and a planned roadmap.

[2] https://github.com/anno4j/anno4j.
[3] https://bitbucket.org/openrdf/alibaba.
[4] http://www.mico-project.eu/.

2 Overview

Anno4j is a Java-based library, that offers developers the opportunity to easily create and consume annotations conform to the WADM by writing Java POJOs. The framework has been designed in a modular and extensible fashion, allowing users to extend at every feature of Anno4j. The core functionalities of Anno4j are:

- **Persistence**: Simple Java objects provide the basis of persistence and can easily be created and persisted with a given Anno4j object (see Sect. 4).
- **Querying**: A `QueryService` object created by an Anno4j instance can be supported with different query criteria (formalised as LDPath expression) to query and consume respective annotations of the Anno4j database (see Sect. 5).
- **WADM implementations**: Built-in and predefined implementations for all basic classes proposed by the W3C Web Annotation Data Model.
- **Transactional behaviour**: By working with transactions, an Anno4j user can create sets of operations, that can either be fully integrated in the database (commit) or not at all (rollback). This ensures the consistency of the database (see Sect. 7.1).
- **Context awareness**: RDF databases are often using different contexts to divide their data into subgraphs. This feature is also possible in Anno4j, turning RDF triples into quads (see Sect. 7.2).
- **Plugin Extensions**: By supporting a plugin interface, users can define own RDF functions in combination with respective evaluation operators, which then can be used as custom querying criteria in order to even enhance the querying functionality and fine tune it to their particular use-case (see Sect. 7.3).
- **Input and Output**: Anno4j is able to both read and write annotations from and to different standardised serialisations, such as JSON-LD, TURTLE, N3-Triples, RDF/XML, etc. (see Sect. 7.4).

3 Related Work

Some approaches to create and query for Web Annotations already exist, and therefore can be considered related to our topic. Alongside the WADM, the Web Annotation Working Group also issued a protocol, namely the Web Annotation Protocol WAP [6]. The purpose is to provide a standard set of actions which are to be supported by both an annotation client and annotation server in order to cooperate smoothly. This enables the formerly discussed advantages of the WADM to be fully exploited, as it is not just desired to build up pairs of participants, but rather a client and server architecture that allows multiple users to consume the information of single data sources. The protocol makes use of the Linked Data Platform LDP [10] to define their core concept of an annotation container. Supported REST and HTTP functionality offers different methods to create and easily query annotations from a given container and provide information about the container itself. This allows to further familiarise a

client with the information or the structure that a given server will support. Listing 1.1 shows an exemplary POST request to create an annotation. The desired annotation content is supported as JSON, in this case a simple annotation that has a `oa:EmbeddedContent` (referring to the URI "http://www.w3.org/ns/oa#EmbeddedContent") body node containing the String (relationship `rdf:value`) "I like this page!". The target of the annotation is the homepage "http://www.example.com/index.html".

Listing 1.1. Example POST request to create an annotation using the WAP protocol

```
POST /annotations/ HTTP/1.1
Host: example.org
Content-Type: application/ld+json

{
  "@context": "http://www.w3.org/ns/oa",
  "@type": "oa:Annotation",
  "body": {
    "@type": "oa:EmbeddedContent",
    "value": "I like this page!"
  },
  "target": "http://www.example.com/index.html"
}
```

Before the WAP has been fully published, similar approaches have been issued in order to give the annotations of the WADM (or in this case formerly known as the Open Annotation Data Model OA [7]) a new facet of interoperability by making them shareable more easily over the web. In [5] Pyysalo et al. describe a minimal web interface using REST, that allows users to create and share OA annotations. They implement two core components, the "OA Store" as client and the "OA Explorer" as server respectively, that can further be enhanced with two components for validation and format conversion.

In comparison, the REST-based approaches offer some advantages. As it is a protocol, there exists the freedom of implementing only parts of the specification, allowing the WAP-conform server or client to be adapted to a specific use case and thusly be more lightweight. The REST interface enables the server to be used as a platform independently, annotations are queried as well as created using JSON. On the other hand, all RDF instances in Anno4j are present as a POJO, allowing them to be used further in object-oriented ways, without the need of up-front parsing. Anno4j is also able to read and generate different serialisations of the RDF objects if needed.

4 Persistence

Anno4j's persistence implementation follows a very simple mechanism. Associated with a local in-memory store or a supported remote SPARQL endpoint, the

persistence features allow users to generate RDF information by creating plain old Java objects - one direction of the ORM. In the following section, this process is illustrated with the example of an image detection, that expresses that "something" has been detected on a given image with a certain confidence. Figure 1 shows the image that will be utilised in the examples, while Fig. 2 depicts an exemplary Web Annotation that states that "Barack Obama" has been detected to "85 %".

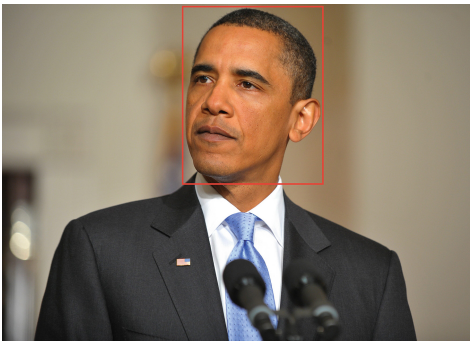

Fig. 1. Picture of Barack Obama. The red rectangle illustrates the area a face was found in

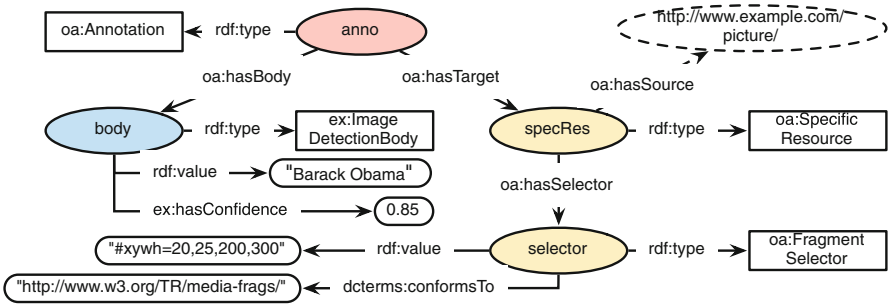

Fig. 2. Resulting RDF graph created with the code shown in Listing 1.2. Round and coloured nodes represent RDF node instances, a rectangle symbolises an RDF class, and rounded rectangles are RDF literals

Listing 1.2 shows a simple workflow example of how to create RDF information with Anno4j Java objects. All required RDF nodes are created independently, using the same Anno4j instance. Various fields are set and the nodes are connected at the end. All associated RDF nodes, relationships, and properties are created automatically and the respective triples are persisted at the registered

triplestore. RDF class information is generated automatically when creating a respective instance, relationships and properties are associated through the setters. The resulting triples that are created after the workflow shown in Listing 1.2 can be seen in Fig. 2.

Listing 1.2. Exemplary creation and initialisation of an object

```
// Anno4j object initialised
Anno4j anno4j = new Anno4j();

// Create the nodes
Annotation annotation =
    anno4j.createObject(Annotation.class);

ImageDetectionBody body =
    anno4j.createObject(ImageDetectionBody.class);
body.setDepicts("Barack Obama");
body.setConfidence(0.85);

FragmentSelector selector =
    anno4j.createObject(FragmentSelector.class);
selector.setConformsTo("http://www.w3.org/
                TR/media-frags/");
selector.setValue("#xywh=1345,25,1050,1325");

SpecificResource specRes =
    anno4j.createObject(SpecificResource.class);
specRes.setSource("http://www.example.com/picture");

// Connect the nodes
annotation.setBody(body);
specRes.setSelector(selector);
annotation.addTarget(specRes);
```

Every RDF node is implemented as an interface class in Anno4j. When creating an instance of it, the proxy pattern is applied and a proxy object of the respective interface is generated. By implementing the supported interfaces, subclasses can be created and the information is also reflected in the RDF graph. Anno4j already comes with built-in and predefined implementations for most of the classes proposed by the Web Annotation Data Model. This enables developers to start right away with the core functionality of producing annotations. However, use-case specific alterations and enhancements are easy to integrate by extending the persistence layer with their own respective classes.

The main element to do so is the Java annotation `@Iri`, which is used at interface level of the given Java interface and on each setter/getter pair. The annotation at interface level sets the RDF class (relationship `rdf:type`) of the

respective RDF object, while an assigned `@Iri` at setter **and** getter method will create the respective predicate attached to the RDF object.

Listing 1.3 shows an example of the implementation of a body interface that is used to convey detected things on a given image. It was already used in the previous example in Listing 1.2. In order to implement the required information of the example shown in Fig. 2, the body node needs to have a specific type (relationship `rdf:type` with the value `ex:ImageDetectionBody`) - defined with the `@Iri` Java Annotation at line 2, and two properties for the detected thing and the confidence about it (properties `rdf:value` in lines 7 and 10, and `ex:hasConfidence` in lines 13 and 16 respectively). Note that the interface extends the `Body` interface, an interface supported for body classes by Anno4j, in order to have the desired behaviour.

Listing 1.3. Exemplary body implementation with one field "depiction"

```
1  // Set the type of the class
2  @Iri(EX.IMAGE_DETECTION_BODY)
3  public interface ImageDetectionBody extends Body {
4
5      // Setters and getters required for the RDF
           predicates
6      @Iri(RDF.VALUE)
7      void setDepicts(String depiction);
8
9      @Iri(RDF.VALUE)
10     String getDepicts();
11
12     @Iri(EX.CONFIDENCE)
13     void setConfidence(Double confidence);
14
15     @Iri(EX.CONFIDENCE)
16     Double getConfidence();
17 }
```

5 Querying

Besides persisting Web Annotations, Anno4j also provides ways to query the annotations or only subparts of them that fit specific needs. On the one hand, Anno4j provides convenient mechanisms to directly query e.g. for all annotation bodies with a particular type or Anno4j Java class. On the other hand, Anno4j offers more expressive ways, using the path-based query language LDPath[5], to define query criteria to reduce the effort for non-SPARQL experts. LDPath, which is similar to XPath, allows a more compact and inline definition of criteria in contrast to the verbose pattern-based query language SPARQL. A

[5] http://marmotta.apache.org/ldpath/.

fluent interface API supports readability and comprehensible query definition. A collection of individual criteria defines the desired characteristic of the resulting annotations. Hereby, multiple criteria are combined with a logical `AND` operation. Helpful functions are supported, that can be used to create key query criteria LDPath expressions via Java methods (such as the type of the body node, the selection features, etc.). This further enhances the usability for beginners and non-SPARQL experts.

Listing 1.4 shows an example using LDPath to define two different criteria for annotations. In this example we are searching for annotations which satisfy the conditions that the annotation body should be a `dctypes:StillImage` and Barack Obama is depicted on the image. The `.execute(Class type)` method does not only define the class of the returned objects, it also does define the starting point of the query and LDPath expressions in the RDF graph. If no type is supported, the standard case is to query for `oa:Annotation`.

Listing 1.4. Anno4j Query Example

```
List<Annotation> annotations = queryService
    .addCriteria("oa:hasBody[is-a dctypes:StillImage]")
    .addCriteria("oa:hasBody/rdf:value","Barack Obama")
    .execute(Annotation.class);
```

Although Anno4j uses LDPath as syntax for query criteria, there is no need for a special LDPath-capable RDF endpoint, because LDPath criteria are translated to an equivalent valid SPARQL 1.1 query. This allows developers to reuse generic SPARQL 1.1 endpoints for their use-cases. Besides basic path criteria, Anno4j also supports a wide range of different LDPath condition types[6]:

- Forward and reverse path conditions
- Resolving of namespace abbreviations
- Recursive pathing like OneOrMore(+) or ZeroOrMore(*)
- Comparison methods like equal, greater, or lower for conditions
- Union of multiple paths
- Type or datatype conditions
- Logical combination of conditions
- Custom functions (see Sect. 7.3).

After execution of the translated SPARQL query against the specified endpoint, all query results are automatically transformed to corresponding annotated Java objects. This abstraction layer allows developers to easily work with RDF information in contrast to constructing complex SPARQL queries and parsing the SPARQL results.

6 Internals

At its core, Anno4j builds upon the OpenRDF Alibaba library (former Elmo codebase) which provides simplified RDF store abstractions and combines the

[6] For further and detailed description of the LDPath criteria, please refer to the LDPath specification at http://marmotta.apache.org/ldpath/language.html.

flexibility and adaptivity of RDF with the powerful object-oriented programming model of Java. It is able to map Java objects to and from RDF resources in a non-intrusive manner that enables developers to work with resources stored in a SPARQL endpoint. Nested properties of RDF resources are lazy evaluated to avoid unnecessary fetching of unused information for faster and efficient query evaluation. Considering Listing 1.3, lazy evaluation implies that the RDF value for `depicts` is not fetched from the SPARQL endpoint until the `.getDepicts()`-method of the respective Java object is called.

As mentioned before, Anno4j uses LDPath syntax to define query criteria. Internally, LDPath criteria is automatically transformed to valid SPARQL 1.1 syntax. Some LDPath expressions can be directly mapped to similar SPARQL 1.1 property path expressions (e.g. path selection, inverse path selection "^", or alternative path "|"). LDPath expressions which can't be directly mapped to SPARQL 1.1 are translated to similar SPARQL constructs (e.g. datatype or "is-a" tests are mapped to SPARQL `FILTER` constructs). To support extensibility, the translation process follows the Interpreter software pattern. Hence there exists a specific interpreter for each LDPath expression. This allows developers to easily integrate custom LDPath expressions, such as function predicates, test functions, and filters, as well as register a query interpreter which transforms the new query element to valid SPARQL 1.1 to ensure full compatibility with generic SPARQL endpoints.

7 Extensions

The basic functionality of the library Anno4j in order to query and persist RDF via Java POJOs has been covered in Sects. 4 and 5. Next to this, several additional features have been implemented, partly supporting the usability of the library in terms of convenience features as well as some features that give a richer RDF feature support. The following section will give insights into those features, which are namely: transactional behaviour (see Sect. 7.1), subgraphs and contexts (see Sect. 7.2), plugin extensibility (see Sect. 7.3), and input/output functionality (see Sect. 7.4).

7.1 Transactions

The Anno4j library features a transactional behaviour, allowing the user to work in an atomic fashion. By creating sets of actions that either are completely executed (**commit**) or not at all (**rollback**), the database is always at a consistent state. A crash in the midst of a work procedure does not create an unclean or untraceable state of data. The basic behaviour (if no `Transaction` object is used) is set to auto-commit, so every action is persisted at the respective database automatically. A transaction itself has to be **started** and ended, which means a **commit** or **rollback**. Listing 1.5 shows an example that creates, begins, and ends a transaction while showing the possibility to create objects and a QueryService. This shows, how persistence and querying operations can be added to that respective set of actions.

Listing 1.5. Use of a `Transaction` in Anno4j.

```
1  Anno4j anno4j = new Anno4j();
2
3  Transaction transaction =
       anno4j.createTransaction();
4  transaction.begin();
5
6  // Create and query using the Transaction object
7  Annotation annotation =
       transaction.createObject(Annotation.class);
8  QueryService qs = transaction.createQueryService();
9
10 transaction.commit(); / transaction.rollback();
```

7.2 Contexts and Subgraphs

A convenience feature of RDF is the use of contexts. Contexts allow users to split their whole RDF graph into smaller contextualised subgraphs. Therefore, RDF triples are turned into so-called **quads**, which have a fourth component after subject, predicate, and object that implements the URI of the subgraph the triple is to be contained in. On the one hand, if no context is defined, the default context is used. On the other hand, context can be utilised in one of two ways in Anno4j:

- Anno4j instance level: Two out of the four possible methods to create an object (`Anno4j.createObject( ... )`) support an additional URI parameter standing for the context. Creating an object this way will insert it in the respective subgraph.
- Transaction level (see Sect. 7.1 for transactions): Every `Transaction` object supports a `.setAllContexts(String uri)` method, which defines the subgraph that the transaction is to write to and read from.

Listing 1.6 shows two examples using a context in Anno4j. Line 1 defines a new URI for a respective subgraph, while line 4 creates an `Annotation` object in that subgraph. Line 7 creates a `Transaction` object and line 8 changes its context to the defined context `uri`.

Listing 1.6. Setting a context for an Anno4j and Transaction object.

```
1  URI uri = new URIImpl("http://www.somePage.com/");
2
3  // Create an Item in the uri context
4  Annotation annotation =
       anno4j.createObject(Annotation.class, uri);
5
6  // Create a Transaction object and define its
       context to uri
```

```
7 Transaction transaction =
      anno4j.createTransaction();
8 transaction.setAllContexts(uri);
```

7.3 Plugin Functionality

By implementing a plugin for Anno4j, users are given the opportunity to add their own querying logic to the library. This is formalised and implemented by introducing an LDPath function in combination with the corresponding logic behind it. Using this, not an actual relationship in the RDF graph is requested, but rather semantic information between given entities are utilised, like for example the target and its selector in order to evaluate the corresponding function. When applied, the query logic is evaluated at the point of time the query is executed. Because of this, the size of the result can be confined beforehand, rather than "doing the logic by hand" afterwards on a bigger result set. In order to implement a plugin, the user has to define the LDPath function expression (`QueryExtension`), as well as the querying logic (`QueryEvaluator`). An exemplary expression (taken from SPARQL-MM [3]) can be seen in Listing 1.7. Integrating that criteria could lead to a result set of only those annotations, that detected both an elephant and a lion, standing next to each other with the elephant found left besides the lion.

Listing 1.7. Exemplary plugin expression in an LDPath criteria.

```
1 QueryService qs = anno4j.createQueryService();
2
3 qs.addCriteria("sparqlmm:leftBesides("elephant","lion")");
```

7.4 Input and Output

To improve the usability of the library, a small extension to the ORM has been implemented. Users can parse their RDF triples formulated in various RDF serialisations to create the respective Java objects, as well as write their Java objects as serialised RDF triples. Among the available serialisations are rdf/xml, ntriples, turtle, n3, jsonld, rdf/json, etc.

In order to read a given RDF annotation (available as Java String), an `ObjectParser` object is needed. Its `.parse(String annotation, String uri, RDFFormat format)` requires the **annotation** as String, a **uri** for namespacing, and the supported **format**. It will then return a Java list of the parsed Annotations. Important to note is, that all RDF nodes that are to be parsed need to be supported as respective Anno4j interfaces. An ObjectParser will keep its parsed annotations locally, a respective call to the `.getAnnotations()` method will return a list of them. Listing 1.8 shows an example reading a turtle annotation.

Listing 1.8. Reading an annotation from a given turtle serialisation.

```
String TURTLE = "@prefix oa:
    <http://www.w3.org/ns/oa#> ." +
"@prefix ex: <http://www.example.com/ns#> ." +
"@prefix dctypes: <http://purl.org/dc/dcmitype/>
    ." +
"@prefix rdf:
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    ." +

"ex:anno1 a oa:Annotation;" +
"  oa:hasBody ex:body1;" +
"  oa:hasTarget ex:target1.";

URL url = new URL("http://example.com/");

ObjectParser objectParser = new ObjectParser();
List<Annotation> annotations =
    objectParser.parse(TURTLE, url,
    RDFFormat.TURTLE);

objectParser.shutdown();
```

To write a given Anno4j Java object to respective RDF serialisations, the `ResourceObject` interface (which every Anno4j object descends from) supports a `.getTriples(RDFFormat format)` method, which returns the representation of the object as a Java String in the supported **format**. Listing 1.9 shows an example that writes a given annotation as turtle triples.

Listing 1.9. Writing a given Java item as turtle RDF serialisation.

```
Annotation annotation =
    anno4j.createObject(Annotation.class);
...
String itemAsTurtle =
    annotation.getTriples(RDFFormat.TURTLE);
```

8 Application

Anno4j was developed within the MICO project [9] and was tailored to the specific project requirements. The MICO project deals with the semantic analysis of multimedia objects in order to create a rich metadata background for the analysed data. This is done to narrow the semantic gap and make the multimedia be more useful to machines. By combining different analysis procedures on the same item, hidden semantics can be found whereby an even wider knowledge

base can be created. To do this, the whole analysis process of the MICO project is based on a single instance, called the **MICO platform**. It combines data and metadata storage, an orchestration unit (the MICO broker [1]), recommendation features, a fully integrated persistence API, and the possibility to combine different (local and external) extractors to form workflows and thereby jointly analyse the supported content.

For all the various extraction results with their formats a unified metadata model was necessary. The MICO Metadata Model MMM[7] is an extension to the WADM. The intermediary and final results are persisted in the form of Web Annotations, while the MMM introduces an RDF structure to connect the various result annotations. This allows to combine different analyses of the same input multimedia item, workflow outcomes are combined and provenance can be traced back.

Hence, it was a requirement for the extractor implementer to both persist and query the metadata using (template supported, but still) verbose and complex SPARQL queries. Additionally, when requesting data from the MICO platform, it had to be interpreted "by hand" first. At this point, Anno4j solved a lot of problems at both sides. By replacing the SPARQL queries, the extractor experts could now both persist and query their respective data using POJOs of the programming language they are familiar with and their extractor is written in. Additionally, querying with LDPath expressions was easier to adopt and had less lines of code than the SPARQL queries. Next to that, when querying data from the respective triple store, having POJOs rather than the answer of a SPARQL query, the result data could be used right away and did not require any further up-front parsing or manual object creation.

9 Conclusion

This paper introduced the library Anno4j, which enables Java developers to create and consume RDF annotations. Those annotations are conform to the WADM, allowing them to be shared and exchanged between different locations. Anno4j features simple persistence of RDF objects via Java objects, its querying functionality is based on LDPath, supporting a wide range of combinable path criteria to form a powerful annotation consumption tool. Both features can be extended easily, so developers can fine tune their respective Anno4j instance to their needs.

Anno4j is available under Apache V2 license at Github[8]. Future work on this library will include the attempt to provide the functionality of Anno4j to other programming languages. First proof-of-concepts show positive results for a C++ mapping using Java Native Interfaces. This would allow developers to write their software natively in C++ but still use Anno4j for a convenient creation and querying of Web Annotations. Other extensions like SPARQL-MM querying allow Anno4j to be used in a broader spectrum.

[7] http://mico-project.bitbucket.org/vocabs/mmm/2.0/documentation/.
[8] https://github.com/anno4j/anno4j.

The current application of the Anno4j library has led to various lessons learned, as different projects make use of Web Annotations in conjunction with Anno4j to make a more convenient use of the annotations. However, as nearly every application of today is web-oriented, a web facet could lift Anno4j to a broader use case. This requirement is exactly tailored to the WAP specification, so future steps will include a layer on top of Anno4j, allowing it to be used as a WAP-conform server. Additionally, as the querying mechanism of Anno4j in combination with LDPath is manifold, we intend to extend the WAP requirements for our implementation to deliver more comprehensive querying.

Acknowledgments. The presented work was developed within the MICO project partially funded by the EU Seventh Framework Programme, grant agreement number 610480.

References

1. Aichroth, P., Sieland, M., Cuccovillo, L., Köllmer, T.: The mico broker: an orchestration framework for linked data extractors. In: Joint Proceedings of the 4th International Workshop on Linked Media and the 3rd Developers Hackshop (LiME 2016, SemDev 2016), Co-located with 13th Extended Semantic Web Conference ESWC 2016 Heraklion, Crete, Greece, 30 May 2016
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. In: Semantic Services, Interoperability and Web Applications: Emerging Concepts, pp. 205–227 (2009)
3. Kurz, T., Schlegel, K., Kosch, H.: Enabling access to linked media with SPARQL-MM. In: Proceedings of the 24nd International Conference on World Wide Web (WWW2015) Companion (LIME15) (2015)
4. Manola, F., Miller, E.: RDF primer. W3C Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/
5. Pyysalo, S., Campos, J., Cejuela, J.M., Ginter, F., Hakala, K., Li, C., Stenetorp, P., Jensen, L.J.: Sharing annotations better: RESTful open annotation. In: ACL-IJCNLP 2015, p. 91 (2015)
6. Sanderson, R.: WAP web annotation protocol. W3C Working Draft, W3C (2015). https://www.w3.org/TR/annotation-protocol/
7. Sanderson, R., Ciccarese, P., de Sompel, H.V.: OADM open annotation data model. W3C Community Draft, W3C, February 2013. http://www.openannotation.org/spec/core/
8. Sanderson, R., Ciccarese, P., Young, B.: WADM web annotation data model. W3C Working draft, W3C, October 2015. https://www.w3.org/TR/annotation-model/
9. Schlegel, K., Berndl, E., Granitzer, M., Kosch, H., Kurz, T.: A platform for contextual multimedia data: towards a unified metadata model and querying. In: Proceedings of the 15th International Conference on Knowledge Technologies and Data-Driven Business, i-KNOW 2015, pp. 1:1–1:8. ACM, New York (2015). http://doi.acm.org/10.1145/2809563.2809586
10. Speicher, S., Arwe, J., Malhotra, A.: LDP linked data platform 1.0. W3C Recommendation, W3C, February 2015. https://www.w3.org/TR/ldp/