# Statement-centric API for manipulating RDF triples

**D. Tomaszuk**

*This paper describes an Application Programming Interface (API) for creating Resource Description Framework (RDF) triples. It proposes a specification in a general, programming language independent way to manage RDF data. This paper defines interfaces, methods and attributes using Interface Definition Language (IDL). The programming interfaces provide a statement-centric approach. This new API is developed in response to a demand from a wide range of users, who want create, update and delete RDF statements.*

## 1. Introduction

An RDF triple consists of a subject, a predicate, and an object. Following [1], let $U$ be the set of all URI references, $B$ an infinite set of blank nodes, $L$ the set of RDF plain literals, and $D$ the set of all RDF typed literals. $U$, $B$ and $L$ are pairwise disjoint. Let $O = U \cup B \cup L \cup D$ and $S = U \cup B$, then $T \subset S \times U \times O$ is set of all RDF triples.

This paper defines interfaces, methods and attributes using Interface Definition Language (IDL) [2]. The IDL describes an interface in a language-neutral way and programming language independent. An Application Programming Interface (API) is an abstraction that describes an interface for the interaction with a set of functions.

In this paper a new programming API that can be used to write RDF graphs is presented. It introduce an alternative mean of access to RDF triples using any programming languages. In Section 2 datatypes necessary to the main interface is presented. In the next section API to manage RDF data is proposed. In the Section 4 results of the experiments are presented. The paper is ended with the conclusions.

## 2. IDL definitions of basic datatypes

There are many libraries that provide various methods and attributes to manipulate RDF triples, which is not compatible, eg. Jena [3], JRDF [4] or RAP [5]. Besides, RDF abstract model could have several serialization formats, that may be used to interchange and store data. Considering the above difficulties this section proposes a new, universal and simple programming API specification to create, delete and update RDF statements. Another advantage of this API is that applications do not need to support additional formats such as XML or JSON to store and interchange data.

This section presents API provides a statement-centric view. This approach means that RDF data is manipulated as a set of RDF triples each consisting of a subject, predicate, and object. This approach differs from the resource-centric API that is closer to the RDF graph data model. The section proposes a hierarchy starting with Term interface with its descendants: Literal, PlainLiteral, TypedLiteral, Resource, URIReference and BlankNode interfaces.

### 2.1. Term interface

An Term is the abstract root type in the hierarchy of datatypes used in the RDF API. This interface contains methods to create URI reference, blank node and plain literal or typed literal. Listing 1 presents Term interface.

**Listing 1.** Term interface

```
abstract interface Term {
  Literal createLiteral ([in] string value, [in] [optional] string language);
  Literal createLiteral ([in] string value, [in] URIReference type);
```

```
  URIReference createURI ([in] String value);
  BlankNode createBlankNode ([in] String id);
  boolean removeLiteral ();
  boolean removeURI ();
  boolean removeBlankNode ();
  Literal updateLiteral ([in] string value, [in] [optional] string language);
  Literal updateLiteralValue ([in] string value);
  Literal updateLiteralLang ([in] [optional] string language);
  Literal updateLiteralType ([in] URIReference type);
  Literal updateLiteral ([in] string value, [in] URIReference type);
  URIReference updateURI ([in] String value);
  BlankNode updateBlankNode ([in] String id);
};
```

The createLiteral method can be overload. The first createLiteral method creates a new Literal that is a plain literal in RDF. The method has two parameters: value and language. The value parameter is a string type. It is the lexical value of this literal. The language parameter is a type string and it is optional. It is the language tag defined in [6]. The createLiteral method returns Literal. The second createLiteral method creates a new Literal that is a typed literal in RDF. The method has two parameters value and type. The value parameter is a string type. It is the lexical value of this literal. The type parameter is URI type. It is the datatype identified by a URI reference. The createLiteral method returns Literal.

The createURI method creates a new URI reference. The method gets value parameter. The value parameter should be an URIReference type. It is the lexical value of this URI. The createURI method returns new URI reference.

The createBlankNode method creates a new blank node. The method get id parameter. The id parameter sbould be a type string. It is the identifier of the blank node. The createBlankNode method returns new blank node.

The removeLiteral method remove existing literal. It returns boolean value – true when literal is deleted and false when it is not deleted (eg. when it is not exists). Close to that is removeURI and removeBlankNode methods.

The updateLiteral method can be overload. It modifies literal – first remove, then create new literal. It gets the same parameters to createLiteral method. The updateLiteralValues, updateLiteralLang and updateLiteralType are variations of updateLiteral method. The updateURI and updateBlankNode modify URI reference and blank node.

### 2.2. Literal, PlainLiteral and TypedLiteral interfaces

An Literal is an abstract type of RDFPlainLiteral and RDFTypedLiteral. It inherits from Term abstract interface. Listing 2 presents RDFLiteral interface.

**Listing 2.** Literal interface

```
abstract interface Literal : Term { };
```

PlainLiteral interface inherits from Literal abstract interface. Listing 3 presents PlainLiteral interface. The value attribute is the lexical value of this literal. It should be a string type. The language attribute is usually the two characters long language tag as defined by [6]. It is should also have a string type.

**Listing 3.** PlainLiteral interface

```
interface PlainLiteral : Literal {
  readonly attribute string value;
  readonly attribute string language;
};
```

TypedLiteral interface inherits from Literal abstract interface. Listing 4 presents RDFTypedLiteral interface. The value attribute is the lexical value of this literal. It should have a string type. The type attribute is the datatype reference to XML Schema [7] types. It should have an URIReference type.

**Listing 4.** TypedLiteral interface

```
interface TypedLiteral : Literal {
readonly attribute string value;
readonly attribute URIReference type;
};
```

### 2.3. Resource, URIReference and BlankNode interfaces

An Resource is an abstract type of URIReference and BlankNode. It inherits from Term abstract interface. Listing 5 presents Resource interface.

**Listing 5.** Resource interface

```
abstract interface Resource : Term { };
```

URIReference interface inherits from Literal abstract interface. Listing 6 presents URIReference interface. The value attribute is the lexical representation of the URI reference. It should have a string type.

**Listing 6.** URIReference interface

```
interface URIReference : Resource {
readonly attribute string value;
};
```

BlankNode interface inherits from Literal abstract interface. Listing 7 presents BlankNode interface. The id attribute is identifier of the blank node. It should have a string type.

**Listing 7.** BlankNode interface

```
interface BlankNode : Resource {
  readonly attribute string id;
};
```

## 3. IDL definition of interface to modify RDF statement

This section proposes an Statement interface, which defines the data structure to represent an RDF triples. One Statement object should consists of single subject, predicate and object. It is the main interface that uses datatypes interfaces such as: Term, Literal, PlainLiteral, TypedLiteral, Resource, URIReference, BlankNode. Listing 8 present this interface in IDL. RDF graph should be collection RDF statement. This proposal do not care of graph consistency, it is dedicated at a higher level because this API provides only flat triples approach.

**Listing 8.** Statement interface

```
interface Statement {
  readonly attribute Resource subject;
  readonly attribute URIReference predicate;
  readonly attribute Term object;
  Statement createStatement ([in] Resource subject,
    [in] URIReference predicate, [in] Term object);
  boolean removeStatement ();
  Statement updateStatement ([in] Resource subject,
    [in] URIReference predicate, [in] Term object);
  Statement updateSubject ([in] Resource subject);
  Statement updatePredicate ([in] URIReference predicate);
  Statement updateObject ([in] Term object);
  string toNTriples ();
  string toJSON ();
  string toString ();
  boolean equals ([in] Statement otherStatement );
};
```

The Statement interface has three attributes: subject, predicate and object. It corresponds to RDF triple. The subject attribute is a subject resource of the Statement. This attribute should have Resource type. It means that subject should be URI reference or blank node. The predicate attribute is a predicate URI reference of the Statement. This attribute should have URIReference type. The object attribute is an object term of the Statement. This attribute should have Term type. It means object should be URI reference, blank node or

literal. Possible datatypes mapped to rdf terms for subject, predicate and object are defined in section 1.

The createStatement method creates a new RDF triple. The method has three input parameters: subject, predicate and object. It returns a new Statement object. The removeStatement method remove existing RDF triple. It returns boolean value – true when Statement object is deleted and false when it is not deleted (eg. when RDF triple is not exists).

The updateStatement method can be overload. It modifies statement – first remove, then create. It gets the same parameters to createStatement method. The updateSubject, updatePredicate and updateObject are variations of updateStatement method.

The toNTriples method returns the N-Triples representation of the statement as defined by [8]. The toJSON method returns the RDF/JSON representation of the statement as defined by [9]. These methods return flat triples. The toString method formats the object's data in a sensible way and returns a string.

Usually programming languages can not simply test two RDF Nodes for equivalence using general constructs such as == operator, so there is equals method to do it. This method returns true if both statements are equivalent.

## 4. Experimental results

Speed criterion, which is adopted to measure the proposed API, is time of parsing files that are storing objects serialization. Main tests are executed on typical desktop computer with two 1600MHz CPUs and 2GB RAM. The test program load files with RDF triples. It is implemented in Python. The program compares the objects serialization based on the proposed statement-centric API and Jena [3], JRDF [4] or RAP [5] N-Triples serialization [8]. The proposed API serialization is through the standard library cPickle module. It is an objective evaluation of the different serialization in libraries such as Jena, JRDF or RAP in the same Python environment.

The result is average of three the best return values from one hundred tests of parsing files. The tests are performed with ten increasing number of RDF triples, set $\{1000, 2000, ..., 10000\}$.

The results show that times of parsing grows close to linear function. The results show that times of parsing grows close to linear function but the native serialization based on proposed API grows definitely slower. The best results belong to the proposal.

## 5. Conclusion

The problem of how to manipulate RDF triples has produced many proposals. We believe that my API is an interesting approach. My application programming interface provides opportunities for object serialization to any storage model. Serialized objects can also be sent and executed in different environments. Another advantage of this approach is the speed of serialization because it is a native solution as opposed to other serialization, which need intermediate layers. Another benefit of this API is that it is close to the RDF model, and it is programming language independent way to manage RDF data. The proposal can be implemented in any programming languages which is object-oriented. It can be used to access to triplestores in low layer.

We realize that some further work on this issue is still necessary extended proposed API to iterating through RDF statements and SPARQL querying [10] for triples.

## Acknowledgements

The author would like to thank Ivan Herman from the World Wide Web Consortium. Professor Henryk Rybinski's comments and support were invaluable.

## References


[1] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. World Wide Web Consortium, 2004.

[2] L. Heaton, OMG IDL Syntax and Semantics, Object Management Group, 2002.

[3] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne and K. Wilkinson. Jena: implementing the semantic web recommendations. International World Wide Web Conference, Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, 2004.

[4] A. Fnewman. Getting Started with JRDF. 2010.

[5] R. Oldakowski, C. Bizer, D. Westphal. RAP: RDF API for PHP. In Proceedings International Workshop on Interpreted Languages, MIT Press, 2004.

[6] A. Phillips and M. Davis, Tags for Identifying Languages, Internet Engineering Task Force, 2009.

[7] P. V. Biron, A. Malhotra, XML Schema Part 2: Datatypes Second Edition, World Wide Web Consortium, 2004.

[8] J. Grant and D. Beckett, RDF test cases, World Wide Web Consortium, 2004.

[9] D. Tomaszuk. Serializing RDF graphs in JSON. III Krajowa Konferencja Naukowa Technologie Przetwarzania Danych, 2010.

[10] J. Perez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, ACM Transactions on Database Systems, 2009.


## Authors


**Dominik Tomaszuk** — Master of Science, Institute of Computer Science, University of Bialystok, Bialystok, Poland; E-mail: *dtomaszuk@ii.uwb.edu.pl*