# JRuby Topic Maps

Arnim Bleier, Benjamin Bock, Uta Schulze, and Lutz Maicher

Topic Maps Lab Leipzig
University of Leipzig, Johannisgasse 26, 04103 Leipzig, Germany
{bleier,bock,uta.schulze,maicher}@informatik.uni-leipzig.de

**Abstract.** The Topic Maps Application Programming Interface (TMAPI) provides a standardized way for working with Topic Maps on the Java platform. TMAPI implementations in other programming languages not available for the JVM use a different stack - not harnessing the existing Java code. In this paper we describe a different approach, reusing Java TMAPI implementations to build a high level Topic Maps API for the Ruby programming language without a heavy wrapper layer around the Java Objects. The high level API greatly simplifies implementations on top of the new API compared to TMAPI with minimal overhead. The plain TMAPI methods can be invoked without any overhead at all. The implementation was thought to be straight forward in the beginning but – as usual – some pitfalls were encountered. The according solutions will be outlined in this paper. The library, documentation, and tutorials can be found on the project's website[1].

## 1 Introduction

The original ActiveRecord-based implementation of RTM[1] (RTM::AR) provided the Ruby community with a persistent Topic Maps engine to be used in Ruby web frameworks like Ruby on Rails[2]. RTM::AR introduced some new features on top of a Topic Maps Data Model-compatible programming interface inspired by the then de-facto standard TMAPI 1.0. The usage of the object-relational mapper ActiveRecord allowed an efficient initial implementation with a concentration on API design and usability. However this academically motivated implementation proved to be disadvantageous performance-wise.

With the increasing popularity of JRuby[3] and the open sourcing of Ontopia[4] another perspective opened up. A strong demand for a well scaling implementation of a Ruby-style Topic Maps API and the opportunities of a Java implementation of the Ruby interpreter paved the way for the JRuby Topic Maps library (JRTM) presented in this paper.

The next section fleshes out the new requirements in detail, motivating the design goals. In the third section the stack of JRTM is introduced to make

[1] http://rtm.topicmapslab.de
[2] http://www.rubyonrails.org
[3] http://jruby.org
[4] http://code.google.com/p/ontopia

Maicher, L.; Garshol, L. M. (Eds.): Linked Topic Maps. Fifth International Conference on Topic Maps Research and Applications, TMRA 2009 Leipzig, Germany, November 12–13, 2009 Revised Selected Papers. Leipziger Beiträge zur Informatik. ISBN 978-3-941608-06-1

the reader familiar with the architectural decisions made. After these normative parts, the technical details of the implementation, and the API are introduced. Consistently beginning at the bottom of the application stack, the TMAPI core extensions are introduced first. The next section addresses syntactic sugar provided by the library. In the following section an API for the navigation between Topic Map constructs according to the TMQL axes is reviewed. The last engineering part of this paper introduces a JRTM I/O layer, providing a common interface across different Topic Map serialization formats and engines. The conclusion of this paper reviews whether all previously formulated requirements have been met and a short outlook on future work is given.

## 2 Requirements

The Topic Maps Lab is in need of a high level API allowing to access various Topic Map engines and building the fundament for a rapid Topic Maps application development. While there is TMAPI 2.0[2], its interface is neither adopted by all engines nor is it available in the Ruby programming language. Even more, the TMAPI provides no standard for the serialization and deserialization of Topic Maps.

For this and other reasons, the following design goals have been taken into account:

- TMAPI should be available to the user;
- Usage independent of the underlying TMAPI implementations;
- Support for multiple instances of different Topic Map engines in parallel;
- Implementation of all TMQL axes as single method calls on Topic Map Constructs;
- A single Ruby Interface for all the I/O API's of the underlying engines;
- Support for Ruby language style;
- Maintaining the syntax known from RTM;
- Possibility of passing Topic References as arguments of methods wherever Topics are required;
- Support of rapid adoption to a potentially changing standard;
- Reasonable performance;
- Extensibility to easily support further Topic Maps engines.

## 3 The Stack

The novel integration framework JRTM is designed upon the principle of modular design. This approach subdivides a system into smaller parts that can be independently created and used in different systems. Figure 1 depicts the stack JRTM has implemented. The following paragraphs discuss each layer of this stack, beginning from the bottom. The RTM::IO library is discussed later separately.

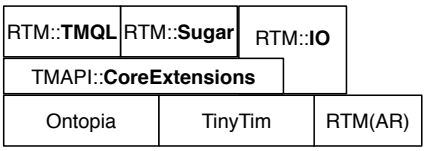


**Fig. 1.** The JRTM Stack

### 3.1 A Ruby Perspective on the TMAPI

Ruby is a modern programming language, promoting multiple programming paradigms such as the object-oriented, the functional, and the imperative. Ruby supports the programmer with dynamic typing, reflections, and garbage collection. The language was initially developed and designed by Yukihiro Matsumoto due to his perception of other scripting languages as being insufficient[3]. While Ruby is not yet standardized, a de-facto standard is given by Matsumoto's implementation of Ruby as an interpreter (MRI), written in the C programming language. The MRI branch was discontinued after the still popular 1.8.x releases in favor of a new virtual machine-based re-implementation introduced with version 1.9, called YARV. The newest version available at the time of writing is 1.9.2 preview 1. The technical specification is formalized mainly by the teams of the alternative implementations Rubinius and JRuby. A formal Standardization with ISO was initially discussed in 2008[5], affirmed in 2009 and is planned for the coming years.

The above-mentioned alternative JRuby[4] is a pure Java implementation. JRuby is alternatively compatible to Ruby 1.8 or 1.9, configured using command line switches. The benefit of using JRuby for our purpose is that the TMAPI is written in Java and by leveraging its implementations in combination with Ruby, programmers get the best of both worlds.

The access of Java from JRuby is straight forward; it is exemplary demonstrated with the creation of a new instance of a TMAPI TopicMapSystem:

```
tms = TopicMapSystemFactory.newInstance.newTopicMapSystem
```

The object assigned to the variable `tms` is a fully-fleshed instance of `org.tmapi.core.TopicMapSystem` providing all the methods the Java interface mandates.

One of our requirements is to support multiple instances of different Topic Maps engines in parallel. Thus, instead of employing standard TMAPI classes and interfaces, classes specific to the implementations have been used; such as:

- `org.tinytim.core.TopicMapSystemImpl` for tinyTiM and
- `net.ontopia.topicmaps.impl.tmapi2.TopicMapSystemImpl` for Ontopia.

[5] http://isotc.iso.org/livelink/livelink/JTC001-N-9289.pdf?func=doc.Fetch&nodeId=7 647385&docTitle=JTC001-N-9289

To allow the programmer connecting to these different implementations in a syntactically consistent way, `RTM.connect` is introduced. The `connect`-method takes two optional arguments:

- a hash specifying the used engine and
- a hash which specifies the parameters for the TopicMapFactory deployed.

The following line exemplifies this by the creation of a new instance of an Ontopia TopicMapSystem with a relational database backend:

```
tms = RTM.connect(:implementation => :ONTOPIA,
                  :backend => :RDBMS,
                  :properties => "path/to/jdbc.props")
```

### 3.2 The TMAPI Core Extensions

The core extensions are providing Ruby style methods for Topic Map Constructs. One of our prominent design goals facilitates the passing of Topic References as arguments of methods wherever Topics are required. These Topic References follow the syntax introduced by the Compact Topic Maps (CTM) notation[5] and as suggested by R. Cerny as part of JSON Topic Maps (JTM)[6], respectively. Thus, when a Locator gets prefixed with

- nothing or "`si:`" the Locator is specified as subject identifier,
- "`=`" or "`sl:`" the Locator is specified as subject locator,
- "`~`" or "`ii:`" the Locator is specified as item identifier.

To accommodate this while leaving the TMAPI methods unchanged, the different method naming conventions of Java and Ruby turned out to be useful. While the TMAPI specifies method names according to the Java naming convention (e.g. `topic.addType`), the Ruby convention suggests every method name to be lowercase only and words to be separated by underscores (e.g. `topic.add_type`)[7].

Even more, given the case that the method is a getter or setter, "`get`" or "`set`" is usually omitted (e.g. `topic.getName` equals `topic.name`). We took the approach to adapt the Ruby style methods, only. However, JRuby automatically creates such Ruby style aliases for Java method names[8]. Although this automatic aliasing guarantees a smooth integration of Java into Ruby, it raises some questions about where in the method table chain to insert our patched methods.

Ruby provides several ways to add methods to an object: methods can be added to

- a so-called meta class of each instance;
- the class of the object or a superclass thereof;
- a module included by the class or a superclass;

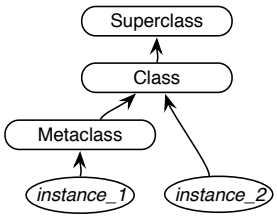


**Fig. 2.** Ruby's meta class, class, and super class inheritance

– a module which is named as the Java interface which is implemented by the class.

Figure 2 depicts this class hierarchy.

The first alternative has the disadvantage of the meta class to be in between a single instance and the class. The methods ought to be added to each individual object separately. Thus, additional methods could only be called on the instance on which the process of adding methods had been carried out.

The second alternative is quite straight forward but requires the names of the classes to be known beforehand, making it hard to keep JRTM independent of the used TMAPI implementation at this step.

The third alternative of adding methods to a module also needs knowledge of the actual classes, even though it is more modular.

The fourth alternative is a special case of the second one and relevant only for JRuby. It allows keeping the additional methods independent of the used implementation and has been chosen as the way to go. However, the automatic alias generation mechanism of JRuby causes methods added this way to be potentially unreachable by the instances.

Ruby instances each have their own unique state but include no method table. Only a pointer from those instances to their meta classes and from there on to their classes and super classes allows calling methods on instances. In this serial manner the search for a called method stops at the first method object matching the call. If, as in the case of JRuby and TMAPI, an alias is created in a class and the method inserted to a super class; the method call gets allocated to the alias but not to the inserted method.

To overcome this issue, the Ruby-style methods which need to be augmented were overwritten in the actual Java implementations to just call `super`[6]. The augmented methods implemented onto the TMAPI interfaces then called the still available Java-style methods to avoid (endless) loops. As any TMAPI-based implementation should be usable out-of-the-box, JRTM needs to know the names of the Java classes which implement the TMAPI interfaces. We created a generic solution using the Clapper Java utility library[7] in connection with the ASM byte-

[6] Overwriting methods is a standard Ruby feature.
[7] http://www.clapper.org/software/java/util/

code manipulation library[8] to find all implementing classes for each extended TMAPI interface. Unfortunately, the searching of implementations drastically degraded the initial startup time. This is of less practical impact on long-running server processes but generally undesirable. A pragmatic solution was to hard-code all classnames of the best-known TMAPI implementations, Ontopia and TinyTiM. Using the implementing classes, in the next step, the Ruby style entries (auto-generated by JRuby) in the method tables of the classes are overwritten with pointers to their super classes when needed.

By doing so, JRTM allows its users to call the patched methods (those which follow the Ruby naming convention) as well as the unaltered TMAPI methods, if needed.

### 3.3 The Syntactic Sugar

According to the creator of Ruby, his language is designed to make programming more productive and fun[3]. Following this general design goal JRTM does not only provide Ruby style method calls to original TMAPI methods, but adds additional functionality that enables programmers to rapidly access, create, and modify Topic Map Constructs. "Hash access" and the `counterparts` method are the most prominent examples of the syntactic sugar which will be discussed in this section.

The Characteristics of a Topic are its Names and its Occurrences and can be accessed using the TMAPI methods `getNames` and `getOccurrences`, respectively. Optionally, both methods can be parametrized by a Topic constraining the type of the Characteristic. While the core extensions already give the programmer the freedom to use a string Topic Reference as parameter, hash access goes beyond that:

Instances of `RTM::Topic` can be used as a hash in which a key specifies the type and optionally the scope of the Names or Occurrences that should be returned. If the key is a Topic, only Occurrences of this type are given back and a scope may not be defined. To differentiate between Names and Occurrences in the case of a key being a Topic Reference, JRTM again makes use of the CTM Syntax convention, i.e. a key for a Name is prefixed with "`-`" while a key for an Occurrences is not[5]. This convention allows writing the type in square brackets right after the Topic:

```
topic["-birthname"] # -> Set of Names
topic["age"]        # -> Set of Occurrences
```

Since a Topic can, according to the TMDM, have multiple Names or Occurrences of the same type, the result is a set similar to the original methods of the TMAPI. If no type (i.e. key) is given, all Names and Occurrences of that Topic are returned.

Beside this functionality as a getter, hash access can equally be used as a setter if the square brackets are followed by an "`=`" as well as the Characteristic that should be added:

[8] http://asm.ow2.org/

```
topic["-birthname"] = "Smith"
topic["age"] = 30
```

A key may also specify the scope a Name or Occurrence should have. This is realized by appending " `@`" as well as all themes of the scope, separated by comma and/or space:

```
topic["-name @deu, at"] # -> Set of Names
topic["-name @fi"] = "Suomi"
```

A syntactic similar but semantic different kind of hash access is provided for Associations. The Association hash access takes as key a Role type and gives back the set of Role players the Association has enacting the given role type. If no type (i.e. key) is given, all Role players of that Association are returned. Next to the getter functionality, the hash access for Associations provides a setter functionality, too. In this case the value assigned to the hash item ought to be a Role player provided as Topic or Topic Reference:

```
association["employee"] # -> Set of Topics (the Role players)
association["institute"] = topic_uni_leipzig
```

Further syntactic sugar provided by JRTM are the `counterparts` and `counterplayers` methods for Topics and Roles.

The `counterparts` method for Topics returns all Roles belonging to an Association in which tis Topic plays a Role, except for the Role the Topic plays itself. An optional filter hash takes values for the keys `:rtype` and `:atype`, whereby only those roles are given back which are of the type `:rtype` and play in an association of the type `:atype`. Equally a `counterparts` method is available for instances of `RTM::Role`.

The `counterplayers` method has a syntax and semantic similar to `counterparts`. The difference is that it does not return the Roles itself but the Role players. `counterplayers` can take the same optional parameters as the `counterparts` method.

The methods `internal_occurrences` and `external_occurrences` are solely available for Topics and return their internal and external occurrences[9], respectively. Furthermore, instances of `RTM::Topic` are equipped with the methods `locators` and `references`, handing back an array of Locators and an array of Topic References (as strings), respectively.

### 3.4 The TMQL Navigation

One of our initial requirements of JRTM has been that not only TMAPI 2.0 functionality is provided with syntactic sugar build on it, but also a mechanism to navigate between Topic Map Constructs is implemented. The Topic Map Query Language suggests twelve navigation axes; namely `types`, `supertypes`, `players`, `roles`, `traverse`, `characteristics`, `scope`, `locators`, `indicators`,

[9] The name external occurrence refers to occurrences having the datatype `XSDanyURI`.

`item`, `reifier`, and `atomify`[9]. We decided to implement all of theses axes in forward and backward direction as single method calls named according to the axes in the TMQL draft. The TMQL axes methods for the backward direction are prefixed with "`reverse_`"

However, the TMAPI 2.0 asserts the method `getRoles` for Associations and – according to the Ruby method naming convention – JRuby provides the methods `get_roles` and `roles`. Since the `roles` axis computes all Role typing Topics and the TMAPI `getRoles` method returns the Roles participating in the Association, a method naming conflict arises.

To overcome this conflict, proxy objects have been introduced, effectively allowing two different methods sharing the same name. A proxy object wraps a Topic Map Construct that acts as starting point of a TMQL navigation step (e.g. Topics, Associations, Names, Occurrences, and Strings) hiding the TMAPI and providing the TMQL methods.

To retrieve such a proxy object from a Topic Map Construct the method `tmql` has to be called. Since multiple successive TMQL navigation steps are supported the result of a navigation step is either again a proxy object or an array consisting of proxy objects and extended by the TMQL methods. To leave the TMQL mode, thus, to access the Constructs inside the proxy, the method `result` has to be called. Consider the following line as an example.

```
topic.tmql.reverse_types.characteristics("TelephoneNumber").
      atomify.result
```

The example code calls the TMQL mode on `topic`, gets its instances and collects the characteristics of the type "`TelephoneNumber`" of these instances. The result is either an empty array – if `topic` has no instances or if the instances do not have names or occurrences – or an array of atoms representing the values of the characteristics.

## 4 A Common I/O API

The serialization and deserialization of Topic Maps and its Constructs are a central aspect of many applications. While the TMAPI provides consistent and unified interfaces for creating, reading, updating, and deleting of Topic Map Constructs, it does not standardize an API for serializing and de-serializing them. This lack of generally agreed I/O interfaces lead to manifold implementations, making it hard to guarantee the modular substitutability between different engines.

To exemplify this; the optional tinyTiM MIO package takes the following line of code to de-serialize a XTM file, while the Ontopia I/O API takes the next line.

```
new XTM10TopicMapReader(tm, new File("path/file.xtm")).read();
tm = new XTMTopicMapReader(new File("path/file.xtm")).read();
```

**Table 1.** Supported (de-) serialization formats

| Format | tinyTim | Ontopia | RTM::AR |
|---|---|---|---|
| CTM | I | I | - |
| CXTM | O | O | - |
| XTM 1.0 | I/O | I/O | O |
| XTM 2.0 | I/O | O | I/O |
| JTM | I/O | - | O |
| YAML | - | - | O |
| LTM | I | I/O | - |
| Snello | I | - | - |
| TM/XML | I | I/O | - |
| RDF[10] | I | I/O | - |

Even more, the different implementations support different sets of serialization formats, making it almost impossible to create a Topic Map application with I/O capabilities but without knowing the used TMAPI implementation beforehand; thus, diminishing the benefits of having created a common application programming interface in the first place.

Table 1 gives the reader an overview on the supported serialization formats of tinyTim, the Ontopia Topic Map engine, and the ActiveRecord-based RTM implementation.

To provide at least a minimum of independence of the used engine, JRTM provides a common syntax for serialization and deserialization.

This I/O library supplies for instances of `RTM::TopicMap` the read method with the location of the resource to be de-serialized as mandatory parameter and a hash of optional parameters. The hash of parameters takes values for the keys

- `:MappingSource`, if importing RDF, to specify the source to read the mapping vocabulary from,
- `:BaseLocator` to provide a base locator,
- `:Format` to specify the format.

It should be noted again that JRTM provides just a layer on top of the Topic Maps processor used. Thus, the way the parameters are interpreted can possibly vary form implementation to implementation. Exemplifying the syntax for the deserialization, the following line of code is given.

```
tm.read("/path/file.xtm")
```

Equally, JRTM provides a common syntax for the deserialization, providing the programmer "to_FORMAT" methods for Topic Maps for the supported formats of the actual engine. Equally to the deserialization method call for `RTM::TopicMap` is translated into calls of the appropriate syntax for the used

[10] Being more precise; RDF is a data model, the respective serialization formats are: N3, N-Triples, RDF/XML, TriG, TriX, and Turtle.

engine. Consequently, the JRTM syntax for the export of a Topic Map to the XTM2 format is as follows:

```
result_string = tm.to_xtm2
```

Furthermore, all other RTM Topic Map constructs are additionally equipped with to_xtm1, to_xtm2 to_jtm, and to_yaml methods, allowing the export of Topic Map fragments to these formats. Unlike in the case of the "to_FORMAT" of `RTM::TopicMap`, solely wrapping the API of a particular engine, this deserialization functionality builds on JRTM's own serializer methods and is present independently of the used implementation. Whereby the to_yaml methods hands back the valid YAML notation of the JSON Topic Maps format, as discussed by Cerny[6]. Likewise, the other export methods return the valid format, as expected.

## 5 Conclusion

In this article we suggested JRTM as a high level Ruby library build on top of Java TMAPI implementations. Going this way JRTM tries to strengthen the impact of already existing and well proven Topic Map engines by guaranteeing access to them from another language. While implementing JRTM we think we succeeded to meet the majority of our initially formulated requirements. However, we are only partly satisfied with our success in implementing a common Ruby API for serializing and deserializing of Topic Maps, reintroducing some dependency on the actual implementation used.

Even though the library is still very young, it is already used by applications such as Musica Migrans 2 and the generic Topic Maps browser Maiana, providing us with valuable feedback. To further guarantee a maximum of possible stability, the RSpec test framework has been applied[11] with more than 600 test cases. An interesting opportunity for the future is that RSpec test cases can be used to generate documentation by the Yard[12] tool, next to the already provided RDoc documentation.

## References

1. Bock, B.: Ruby Topic Maps. In: Lecture Notes in Computer Science. v. 4999/2008, p. 172–185 Springer, Berlin (2008)
2. Heuer, L., Schmidt, J.: TMAPI 2.0. In: Maicher, L.; Garshol, L. M.: Subject-centric Computing. p. 129–136 Springer, Berlin (2008).
3. Venners, B.: The Philosophy of Ruby, A Conversation with Yukihiro Matsumoto. http://www.artima.com/intv/ruby4.html
4. Ruby Community: Ruby, http://jruby.codehaus.org/
5. Bogachev, D.; Heuer, L.; Hopmans, G.: Topic Maps – Compact Syntax. Working draft, (2008-05-15). http://www.isotopicmaps.org/ctm/

[11] http://rspec.info
[12] http://yard.soen.ca

6. Cerny, R.: JSON Topic Maps. http://www.cerny-online.com/jtm/
7. Cohen, J.: Ruby 101: Naming Conventions.
http://www.softiesonrails.com/2007/10/18/ruby-101-naming-conventions
8. Sun microsystems: JRuby Basics.
http://www.javapassion.com/rubyonrails/ruby_jruby.pdf
9. ISO/IEC WD 18048: *Information Technology – Document Description and Processing Languages – Topic Maps – Query Language*. International Organization for Standardization, Geneva, Switzerland, 2007-07-13
http://www.isotopicmaps.org/tmql/