ADJUSTING WSMO API REFERENCE IMPLEMENTATION TO SUPPORT MORE RELIABLE ENTITY PERSISTENCE[1]

Ivo Marinchev

Abstract: In the presented paper we scrutinize the persistence facilities provided by the WSMO API reference implementation. It is shown that its current file data-store persistence is not very reliable by design. Our ultimate goal is to explore the possibilities of extending the current persistence implementation (as an easy short-run solution) and implementing a different persistent package from scratch (possible long-run solution) that is more reliable and useful. In order to avoid "reinventing the wheel", we decided to use relational database management system to store WSMO API internal object model. It is shown later that the first task can be easily achieved although in not very elegant way, but we think that the later one requires some changes in the WSMO API to smooth out some inconsistencies in the WSMO API specification in respect to other widely used Java technologies and frameworks.

Keywords: Semantic Web Services, Web Service Modelling Ontology (WSMO), WSMO API, WSMO4J.

ACM Classification Keywords: H.3.2 Information Storage: File organization; I.2.4 Knowledge Representation Formalisms and Methods: Representation languages

Introduction

Web services are defining a new paradigm for the Web in which a network of computer programs becomes the consumer of information. However, Web service technologies only describe the syntactical aspects of a Web service and, therefore, only provide a set of rigid services that cannot be adapted to a changing environment without human intervention. Realization of the full potential of the Web services and associated service oriented architecture requires further technological advances in the areas of service interoperation, service discovery, service composition and orchestration. A possible solution to all these problems is likely to be provided by converting Web services to *Semantic Web* services. *Semantic Web* services are "self-contained, self-describing, semantically marked-up software resources that can be published, discovered, composed and executed across the Web in a task driven semi-automatic way" [Arroyo et al, 2004].

There are two major initiatives aiming at developing world-wide standard for the semantic description of Web services. The first one is OWL-S [OWL-S, 2004], a collaborative effort by BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, SRI International and Yale University. The second one is Web Service Modelling Ontology (WSMO) [Roman et al, 2004], a European initiative intending to create an ontology for describing various aspects related to Semantic Web Services and to solve the integration problem.

As part of the later initiative the WSMO API specification and reference implementation [WSMO4J] has been developed. WSMO4J is an API and a reference implementation for building Semantic Web Services applications compliant with the Web Service Modeling Ontology. WSMO4J is compliant with the WSMO v1.0 specification [WSMO v1.0] (20 Sep 2004). At the time of this writing the WSMO API reference implementation is version 0.3 (alpha), that means that it is far from completed product and is subject to changes without prior notice. Nevertheless the API is incomplete as part of our work on the INFRAWEBS project [Nern et al, 2004] we have to utilize the WSMO4J package as it is the only available working implementation of the WSMO specifications.

[1] The research has been partially supported by INFRAWEBS - IST FP62003/IST/2.3.2.3 Research Project No. 511723 and "Technologies of the Information Society for Knowledge Processing and Management" - IIT-BAS Research Project No. 010061.

Current State of the Art

At the time of this writing the reference implementation of the WSMO API [WSMO4J] can export its internal data model to a set of binary files organized in a bundle of directories that correspond to the major entity (entities that are identifiable according to WSMO API terms) types. Every identifiable entity is saved as a separate file that contains the serialized entity identifier, then the Java object that represents the entity and at the end several lists of identifiers corresponding to the different groups of entities that are subordinates of the current entity. The order in which these lists are serialized to the output file (stream) is implementation specific and is implemented by several internal classes named entity-type processors. Every entity type has separated processor class that serializes/deserializes the corresponding objects to/from their persistent state. The subordinate entities are stored in the same way in separate files and so on. Fig. 1 shows the directories created by the file data-store. In practice, this simple solution appears to be very unreliable and a relatively small problem may incur enormous data-losses. The reader also has to keep in mind that this storage mechanism is intended to be used for processing ontologies. And all of the ontologies that are applicable to real-world problem tend to be extremely large.

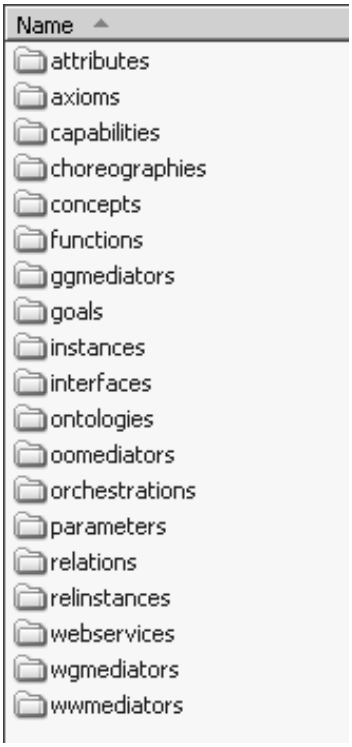

Fig. 1: Directories created by the "FileDataStore" class of the reference implementation.

This ad hoc solution is simple and does not require any third party programs/libraries but it has several major drawbacks that prevent it to be used in a production system. Some of them are implementation problems and can be easily circumvented but others require rather sophisticated solutions to general purpose problems. Below we enumerate some of the problems:

1. If a certain entity is loaded, all its dependent entities are loaded as well no matter whether they are needed or not.
2. If any object has to be changed all its subordinate objects are overwritten again no matter if they are changed or not. Moreover because the implementation of the ObjectOutputStream in Java serializes objects by reachability (when an object is serialized all objects that are referenced by it are serialized as well) a certain object is serialized as many times as the number of objects that refers to it in *direct and indirect way*! *Such persistence scheme brings enormous excessive overhead in the serialization of large object graphs and the worst is that the overhead increases exponentially with the size of the object graph*. A possible solution to the later problem is all objects to implement Externalizable interface in order to control their serialization process but even it will not remove all unneeded read/write operations.
3. In the current implementation, the file names correspond to the entity identifiers. These identifiers may become rather long. This is especially important as many identifiers are created from URLs and at the same time, many of the entities have the identifiers that extend the identifier of its parent entity (for example axioms that are part of the ontology). The implementation makes the file names additionally longer by encoding some special characters that may be prohibited by certain file systems (for example / is encoded with .fslash., : with .colon., * with .star.). At the same time most of the file systems do not allow file names with more than 255 characters.
4. The store operations are not atomic. Thus, there is no guarantee that the data will remain consistent and the original object graph can be recreated from its serialized state. For example if an exception occurs when the data is being saved, the operation is terminated and the on-disk structures remain in an unpredictable and

undeterminable state – the user neither can fix them, nor can turn them to their state before the last operation occur (roll back the last operation).

5. The store implementation is not thread-safe due to the usage of fields to transfer data between methods. But even that one can instantiate several different data-store objects they can not work on the same store simultaneously because of the lack of any locking or synchronizations.

In short, if we use database terminology, the current implementation is very far from being ACID[1] compliant. It is obvious that any of the above issues can be solved but the solutions are usually very sophisticated, and one ultimately will implement complete transactional database storage engine in order to solve all of them.

Using Relational Database as a Data Store

The first improvement that we implemented was to move the persistent data to the relational database by just replacing the file data-store directories with the database tables. The tables consist of two columns: the first one for the entity identifier, and the second column of type BLOB (Binary Large Object) that stores the serialized Java objects. This extension was relatively easy to be implemented. We changed several private methods that deal with the file names and entity types (getFileNameFor, getEntityType, etc.) to work with the database tables and records, and then we changed all of methods that serialize and deserialize data to store/load it to the corresponding BLOB fields instead of using file output and input streams. In order to be able to use the transaction facilities provided by all modern relational database management systems (RDBMS) we use a single database connection that is initiated at the beginning of the store process and get committed at its end (or rolled back in case of exceptional circumstances).

Even with these simple modifications, we get several important advantages:

1. We get atomic changes – all of the changes are written or all are discarded at once.
2. No data is lost in case the storing gets terminated - not only by checked exception but even if the whole process is terminated by the unchecked one.
3. The store may be physically located on the remote system.
4. Several different client processes can use the store simultaneously.
5. The store may use the back-up, replication, and clustering facilities provided by the underlying RDBMS.
6. The store uses the data caching provided by the database.
7. The database can be changed at will if one does not use proprietary database extensions.

Avoiding Identity-Lists Serialization

The next logical consequent step is to start removing object serializations. As it was discussed earlier when a certain entity is persisted the file data-store in the reference implementation serializes first the entity identifier, then the entity object and at the end several lists of the identifiers of the entities that depend on the current one in the entity hierarchy. This last step is entity type specific and is implemented by a specialized entity processors for different type of entities that take care of saving, loading the lists (Vectors in Java terms) of identifiers.

Using the information from these entity processor classes, we created a separate table columns that hold the lists of identities of a given type. For example, for the “capability processor”, we created columns for Assumptions list, Pre-Conditions list, Post-Conditions lists, and Effects list. Thus, the content of the database tables gets more human readable and it is easier to debug potential problems, but we have to emphasize that the database is still not even in the first normal form (1NF) - it requires all table columns to be atomic.

Utilizing Object-Relational Frameworks

The newly created columns in fact represent the relationships between the entities represented by the corresponding table rows and their dependent entities. That is why we can remove these columns and replace them with foreign key columns in the dependent tables for one-to-many relationships and with relationship tables

[1] ACID – Atomicity (states that database modifications must follow an “all or nothing” rule), Consistency (states that only valid data will be written to the database), Isolation (requires that multiple transactions occurring at the same time not impact each other’s execution), Durability (ensures that any transaction committed to the database will not be lost).

for the many-to-many relationships. Dealing with one-to-many relationships with hand-written code is boring but not a complicated task. But the many-to-many relationships can become really problematic to be manipulated as they use additional (relationship) tables and the WSMO object model even have many-to-many reflexive relationships (for example the one between concepts and sub-concepts of the ontology). These facts imply that the programming code needed to deal with all these "housekeeping" activities will be much more than the code that implements the actual business logic.

At this point one can realize that the required changes to the original reference implementation become rather complex and in fact we start "reinventing the wheel" that is already created by others. So, the wise approach to this problem is to use some object-relational mapping frameworks to do the work for us. There are a lot of such frameworks available (for example Java Data Objects [JDO] implementations, Hibernate [Hibernate], Oracle TopLink [TopLink], and others) and many of them are open-source and free even for commercial use. The common feature of all these frameworks is that they use XML configuration files to specify how the objects, fields and relationships (object model) are mapped to the corresponding database tables and columns (relational model). The basic idea behind these mapping files is to keep the object model and the relational model loosely coupled so that the two models can be changed independently. Utilizing such framework provides other useful features:

1. Automatic generation of database queries;
2. Loading/saving/updating the complete object graph with a single method call;
3. Lazy-loading (or on-demand loading)[1];
4. Tracking the user changes and updating just the changed fields;
5. Support for many different RDBMS;
6. Object caching - even distributed caching is possible;
7. Other specific features.

For our purpose the lazy-loading is extremely useful because if one wants to load and change a certain entity it does not need to load and save the complete sub-graph that originates from this entity.

Unfortunately, it appears that several significant issues arise in any attempt of integration between object-relational mapping framework and the current version of the WSMO API and its reference implementation. These issues are discussed in the next section. At the end of it, we represent one possible solution of the problem and why we think the proposed changes are appropriate.

Problems with the Current Version of WSMO API and its Reference Implementation

The most serious problem concerning the applicability of the OR mapping framework with the WSMO API (and its reference implementation) is that the object-relational frameworks work with JavaBeans classes/objects. We do not know why WSMO API was specified and implemented in its current form, but the fact is that all of the entity classes in it deviate from the JavaBeans specification [JavaBeans]. Specifically they lack the properly named accessor and mutator methods for the non-primitive types. At the same time, all methods for accessing non-primitive types are named as listXXX. We do not know why such naming scheme has been selected but we think that it is even not very intuitive. The worst is that at the same time listXXX methods return value is of type java.util.Set. In fact, the following issues appear:

1. The names of the property accessor methods are misleading for the user and deviate from other well-known framework and the JavaBean specification.
2. The semantics of the Set and List data types are significantly different as the list is an ordered collection of elements. More over unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null

[1] The framework loads the expensive (in memory footprint and construction time) object fields and referenced objects just before they are accessed by the client program. This feature is usually very flexible and can be configured in the mapping files on a field level.

elements if they allow null elements at all. So the words list and set is not very appropriate to be used interchangingly.

3. Using “un-typed” return types in the listXXX methods in the otherwise very strongly typed specification is somewhat strange decision.

It is true that we can overcome the first problem by sub-classing all needed entity classes of the reference implementation and turn them to regular JavaBeans by adding the missing accessor and mutator methods and then create mappings for the newly introduced classes. But we do not want any consequent version of the reference implementation to break our “extension”, or to require conversions of the database schema. So, this solution does not seem appropriate in the long-run.

At the end we will express our inner conviction that the persistence package has to be as loosely coupled as possible to the rest of the implementation, and to be written as much as possible against the specification not against the implementation as it is now. In the confirmation of the later we propose the WSMO API specification to be changed in the following way:

1. Add the missing get/set methods to the entity interfaces to turn the implementation classes in correct JavaBeans.
2. Introduce type-safe sets for any entity type that is needed and return them in the corresponding property accessor methods instead of java.util.Set.

Conclusion

As a conclusion we want to point out that although it is in its early stage of development and the fact that it is or still may be immature in some of its parts, the WSMO4J is sound enough to be used as a development tool in the research projects, and facilitates researchers in the early adoption of the WSMO related technologies. We hope that the WSMO API working group will take into account our remarks and suggestions and even they are rejected they will contribute in some way in the future improvements of the specification and/or implementation.

Bibliography

[Arroyo et al, 2004] Arroyo, S., Lara, R., Gomez, J. M., Bereka, D., Ding, Y., Fensel, D. Semantic Aspects of Web Services. In: Practical Handbook of Internet Computing. Munindar P. (Editor). Chapman Hall and CRC Press, Baton Rouge, 2004

[JDO] http://java.sun.com/products/jdo/

[Hibernate] http://www.hibernate.org

[JavaBeans] http://java.sun.com/products/javabeans/

[Nern et al. 2004] H.-Joachim Nern, G. Agre, T. Atanansova, J. Saarela. System Framework for Generating Open Development Platforms for Web-Service Applications Using Semantic Web Technologies, Distributed Decision Support Units and Multi-Agent-Systems - INFRAWEBS II. *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS*, ISSN 1790-0832, Issue 1, Volume 1, July 2004, 286-291.

[OWL-S, 2004] The OWL Services Coalition: *OWL-S:* Semantic Markup for Web Services, *version 1.0*; available at http://www.daml.org/services/owl-s/1.0/owl-s.pdf

[Roman et al, 2004] D. Roman, U. Keller, H. Lausen (eds.): Web Service Modeling Ontology (WSMO), version 0.1; available at: http://nextwebgeneration.com/projects/wsmo/2004/d4/d4.1/v01/index.html

[TopLink] http://www.oracle.com/technology/products/ias/toplink/

[WSMO v1.0] http://www.wsmo.org/2004/d2/v1.0/20040920/

[WSMO4J] http://wsmo4j.sourceforge.net

Author's Information

Ivo Marinchev – Institute of Information Technologies, Bulgarian Academy of Sciences, Acad. G. Bonchev Str., Bl. 29A, Sofia-1113, Bulgaria; e-mail: ivo@iinf.bas.bg