

Testing a custom object database using a RESTful interface: A case study

Kiran Kumar Kanchibhotla¹ and JONNALAGADDA Srinivas²

^{1,2} Ojus Software Labs Pvt. Ltd.,
16-11-310/12/A/1/1,
Saleem Nagar Colony – II, Malakpet,
Hyderabad – 500 036, AP
{js, kiran}@ojussoftware.com

Abstract:

In the world of relational database management systems (RDBMSs), the user data model has long been standardised based on Codd's relational algebra. Programmatic access to data within such systems was, in early stages, provided through vendor-specific database, and optionally client, libraries. Much later, such access to relational database management systems was itself standardized into ODBC/JDBC APIs. This latter standardisation was in no minor way enabled by the former. In the world of object-oriented databases, similar, though not as widely accepted or implemented, standards exist for user data model, in the form of ODMG standards. In contrast to the relational scenario, however, access to data within object-oriented databases has been specified in the form of programming language-specific bindings for Smalltalk, C++ and Java, while ODBC/JDBC are language independent APIs. It is, therefore, reasonably clear that testing an object-oriented database, even when it is strictly ODMG-compliant, through its language bindings is more complex than testing relational ones through ODBC/JDBC APIs.

The current system discussed in this paper is a custom object database with a data model that is neither relational nor strictly ODMG-compliant. While being loosely based on the ODMG model, as regards atomic attributes, composite attributes and collections, it does not fully implement the same. In addition, it features certain interesting and powerful extensions to the ODMG model, particularly with respect to collections and weakly referenced objects. It provides extensive facilities for declaring and enforcing both attribute-level and object-level constraints. Orthogonally, the system maintains all historical versions of each object, as updates are made to the same. Also orthogonally, the system features a multi-level, chained privilege management that works together with a "user" and "role" system of impersonation. All of these increase the number of independent variables in the system, making testing highly complex.

Owing to its current implementation being in the programming language Ruby, the data access APIs do not follow from those of ODMG language bindings. Further, the rich feature set of the database management system resulted in a rather large, and highly parameterised, API. A straight forward approach to testing such a system results in high effort and long cycles. The application layer that was developed in conjunction with the current system employed RubyOnRails as the framework for the presentation layer.

The RESTful approach encouraged by RubyOnRails was utilized in standardising component interfaces and inter-component interactions. Consequently, in order to ease testing the database itself, it was suggested that the direct API available for programmatic operations on the objects within the database be wrapped into a similar RESTful interface. The proposed interface would allow testing create, read, update and delete (CRUD) operations on objects inside the database, using the seven methods in RubyOnRails REST interface for resources, viz. index, new, create, show, edit, update and destroy. Supplementing them were to be a small set of methods for performing certain object operations - such as saving an object to a user-private area - that are specific to the current system. Such an interface was implemented, and was used as the primary API for testing the majority of the database management system

operations. The current paper describes the testing approach that was taken towards designing, implementing, executing and maintaining a “good-enough” regression test suite for the said testing.

Keywords:

RESTful interfaces, agile testing, noun-verb technique, feature matrix, gray-box testing, good-enough testing.

1. Introduction

In this paper we describe the approach employed for testing an object database which was developed using Ruby programming language employing an agile development methodology. The testing was performed on the available database features using only a small RESTful programmatic interface exposed by it. We also describe how we have used the noun-verb technique to prepare a feature matrix in order to identify the dependencies in various requirements and components of the system. This feature matrix was then used for defining the test cases. In addition, gray-box testing technique was used to automate testing of the application.

2. Test Approach for testing object database

In this section, we describe the major features of the object database, and the RESTful interface that exposed the same. The feature matrix, together with the methodology followed to arrive at it, are also presented.

3.1 Object Database Features

The following are the major features of the database system:

- it allows programmers to quickly define a complex data model using a domain-specific language (DSL) in Ruby
- it allows for definition of coarse business objects, called “resources”, each encapsulating one or more fine-grained objects
- a concept of “namespaces” allows for a logical separation of data into non-conflicting object graphs
- an associated application framework provides a registration unit for packaging various definitions, called “application”, which can be prepared using an “application specification”
- objects stored in the database are automatically revisioned; programmers can retrieve older revisions of the objects from the system on demand
- it allows the management of data using a REpresentation State Transfer (RESTful) interface (refer to RESTful interface in Appendix section)

The above features are briefly explained below.

3.1.1 Entity Types

An Entity Type represents a class definition that serves as a template for objects. The database system provides a root entity type meta class called “DataEntityType” that defines a large number of standardized services for defining new entity types. A root entity type called “DataEntity” similarly acts as the root entity type, and defines the guaranteed primary call interface of any object in the database. An entity type can be inherited from another entity type forming a parent child inheritance relationship. By default, DataEntity acts as the parent entity type for new entity types.

An entity type definition can consist of simple attributes, called “atoms”, having data types String, Fixnum, Float, Date, Time and DateTime. Atoms could also have complex types. For instance, an atom in an entity type A could have data type B, where B is another entity type. This facility can be used to define object composition. Continuing the above example, the object that is stored in the atom can be an instance of either B itself or any of its direct or indirect subclasses.

An entity type can also have computed attributes, called “derived atoms”. A derived atom calculates its value at run-time as per the functionality in a block used to define it. This could depend on the values of other attributes in the object, lookups into other objects, or whatever information is needed for such a computation. Derived atoms have a facility to optionally store a computed value in the database.

The system allows for the definition of attributes that store arbitrary binary objects, called “blobs”. Blobs are stored in a content-addressable manner, with a single copy of the object being stored across the system, even when stored multiple times.

Atoms of data types that are other entity types are strong references. The database also provides an attribute that is a weak reference. This is called an “association”. An association references an instance of the specified entity type. Since this is a weak reference, deleting the referenced object orphans the association. As in the case of an atom, the referenced object could be an instance of either the specified class itself or any of its direct or indirect subclasses.

In addition to atoms and associations that represent one-to-one relationships, the database provides attributes that hold one-to-many-related data. These are called “collections”. Collections can be of simple data types or complex ones, and of associations. Also, collections could be either ordered collections, acting as sequences, or could be dictionaries with keys and values.

3.1.2 Resources

As mentioned above, a “resource” represents a coarse business object, which encapsulates one or more entity types. Each resource has an identified primary entity type that is managed by it. An attribute of such a primary entity type can be designated to be shown in object listings. This atom is called “primary entity type listing atom”. Optionally, it can also identify an attribute in the primary entity type on which a uniqueness constraint can be set. Further, such a constraint can be made case insensitive for string attributes.

Resources are the primary holders of the business logic of an application.

3.1.3 Namespaces

“Namespaces” provide logically separated address spaces for objects to reside in. This allows for isolated object graphs to exist, without polluting a global logical storage area. While strong references are not allowed across namespaces, it is possible to setup associations that cross namespace boundaries.

3.1.4 Applications

An “application” is a unit of self-contained functionality. It comprises a set of resources and their entity types, together with a namespace in which its data has to reside. This namespace is owned by the application.

3.1.5 Application Specifications

An “application specification” is a packaging unit for deploying an application. It provides the instructions needed to register the application, its namespace, its resources and their entity types.

3.2 RESTful Interface

Once the registration process of the applications is complete, all object storage and retrieval operations can be performed using a RESTful interface provided on each resource. All resources subclass a system-provided base class called “ResourceBase”, which provides the base implementations for all the methods of the said interface. These methods comprising the RESTful interface are listed below, with a brief description of each.

Method	Description
index	this method returns a listing of objects of the current resource from the database
new	this method returns a new, and empty, instance of the current resource
create	this method stores the given new object in the database
show	this method retrieves one requested object from the database
update	this method updates the given object in the database, creating a new revision of the same
destroy	this method makes the current revision of the object inaccessible, effectively deleting it

Each of the above methods is richly parameterised to provide for specifying a number of filters. Privileges available to the current user are automatically taken into account by these methods.

Since these methods expose the database storage and retrieval operations, by exercising them in carefully crafted round-trip operations, it should be possible to test various

aspects of the database itself. This, precisely, was the approach taken to test the database system, as detailed in later sections.

3.3 Test Analysis

This phase is a precursor of identifying the testing effort, testing techniques that will be employed, testing tools that will be employed to test the application. In this phase we perform the following activities to determine a high level picture of the above and produce a test strategy.

3.3.1 Requirement Analysis

We took the functional and non-functional requirements of the system, and asked the following questions:

- What is the significance of this feature?
- Why is the feature required?
- Where is this feature used?
- When is it used?
- Who uses it?
- Is this feature testable?
- What is the effort involved in testing a feature?

Based on the answers to the above questions, the ambiguity in the requirements was understood and resolved. Appropriate questions were asked and tracked through an issue tracker. This enabled us to determine high-level testing effort, and identify those features that could not be tested due to various constraints.

3.3.2 Feature Matrix

We employed an agile development methodology so that we could have short release cycles. Typically, when there are short release cycles, it is difficult for the testing team to ensure adequate coverage within the duration of each cycle. In order to overcome this problem, we have used the noun-verb technique to build a feature matrix to identify the dependent test cases, and accordingly plan the effort for building a “good enough” set of new and regression test cases for each release cycle.

Building a feature matrix involves identifying the nouns and verbs in the features that are available in the system, and placing them in rows and columns. Then we try to identify if there are any dependencies arising out of noun-noun, verb-verb, noun-verb or verb-noun combinations. Since the noun-verb and verb-noun combinations normally result in a state transition, they are further placed in the feature matrix and are re-analyzed.

We asked the following questions to determine if there are any dependencies.

- Does this feature depend on any other feature?
- Does this feature invalidate any other feature?
- Does this feature influence any other feature?

Keywords	Application Spec	Application	Entity	Resource	Index	New	Create	Read/Show	Update	Delete/Destroy	Version	Object	Re-create	Register	Install	Upgrade	Versioned Object	Versioned Resource	Versioned Entity		
Application Spec	x																				
Application	Y	x																			
Entity	Y	N	x																		
Resource	Y	Y	Y	x																	
Index	N	N	N	N	x																
New	N	N	Y	N	Y	x															
Create	Y	Y	Y	Y	Y	Y	x														
Read/Show	Y	Y	Y	Y	Y	N	Y	x													
Update	Y	Y	Y	Y	N	N	Y	Y	x												
Delete/Destroy	N	N	Y	Y	N	N	Y	Y	N	x											
Version	N	N	Y	Y	N	N	Y	Y	Y	Y	x										
Object	N	N	Y	Y	Y	N	Y	Y	Y	Y	Y	x									
Re-create	N	N	N	N	Y	Y	N	N	Y	Y	Y	Y	x								
Register	N	N	Y	Y	N	N	N	N	N	N	N	N	N	x							
Install	N	Y	Y	Y	N	N	N	N	N	N	Y	Y	N	Y	x						
Upgrade	N	Y	Y	Y	N	N	N	N	N	N	Y	Y	N	Y	Y	x					
Versioned Object	N	N	Y	Y	Y	N	Y	Y	Y	Y	x	x	Y	N	Y	Y			x		
Versioned Resource	N	N	Y	x	Y	N	N	Y	Y	Y	x	Y	Y	Y	Y	Y			Y	x	
Versioned Entity	N	N	Y	Y	N	N	Y	Y	Y	Y	x	Y	Y	Y	Y	Y			Y	Y	x

Refer to the above feature matrix that was prepared by taking a few such nouns, verbs and their combinations. Each cell in the above feature matrix will have the test cases that are related to noun-noun, verb-verb or noun-verb combination, the test case includes both positive and negative test cases.

To highlight the advantages of the above approach, let us consider two examples. Calling “new” (a verb) on Resource (a noun) yields a new object, and storing that using “create” (a verb) yields an object that is version-tracked.

Resource.new => anObject

Resource.create(anObject) => aVersionedObject

From the above matrix, we see, consequently, that the cells representing the intersections of Resource and “new”, and Resource and “create” are both “Y”.

Continuing the above example, the object returned by “new” should be of the type of the entity that is encapsulated by the current resource. Thus, we see that the cells representing



the intersections of Resource and Object (a noun), Resource and Entity (a noun), and Entity and Object are “Y” as well.

As a second example, let us consider the intersection of Entity and Upgrade (a verb). When an existing entity is upgraded, it results in a new version of the entity.

Entity.upgrade => aVersionedEntity

Resource.upgrade => aVersionedResource

By making the resultant elements part of the matrix again, we can now ask questions like: can an old object be accessed using a newer version of its resource? Or, can the newer version of an object be accessed using an older version of its entity, but a newer version of its resource?

This feature matrix was updated for every release, helping us in quickly identifying the effort involved in testing a given release. This also enabled us to identify the most critical functionality that should be tested for the release, together with the dependent features that were to be tested. For each release, the test cases that had to be executed were identified based on such an impact analysis performed on the feature matrix.

3. Testing Techniques

We have employed a gray-box testing strategy for testing the application. This involves a combination of both black-box and white-box testing techniques. Employing this technique enabled us to design test cases and test scripts that allowed us to concentrate more on the low-level components by performing exhaustive testing, and spending less effort on the high-level components by designing a good enough test suite that utilised the published RESTful interface.

The database was developed using Ruby as the programming language, and hence RUnit was a natural choice for writing the automated regression suite. We have used RUnit's in built `assert_*` functions to check the output of various method interfaces (of the entry points). Where the output was written to the secondary storage, a binary diff was used to compare the outputs to check the same for consistency.

Those components (classes and utility functions) which were used by other components in the system were tested individually with the help of certain supporting components and functions, as needed. All other functionality was tested using the RESTful entry points only.

We have also used Rcov, a Ruby code coverage tool, to determine if the test scripts are covering the complete code. Looking at the coverage in the code coverage tools, we have updated the feature matrix and test cases after every iteration cycle.

4. Appendix

RESTful Interface

REST: REpresentational State Transfer; is a way of thinking about the architecture of distributed hypermedia systems. This is relevant to us because many web applications can be categorized this way.

The ideas behind REST were formalized in Chapter 5 of Roy Fielding's 2000 PhD. dissertation. 4 In a REST approach, servers communicate with clients using stateless connections: all information about the state of the interaction between the two is encoded into the requests and responses between them. Long-term state is kept on the server as a set of identifiable resources. Clients access these resources using a well-defined (and severely constrained) set of resource identifiers (URLs in our context). REST distinguishes the content of resources from the presentation of that content. REST is designed to support highly scalable computing while constraining application architectures to be decoupled by nature.

There's a lot of abstract stuff in this description. What does REST mean in practice?

In REST, we use a simple set of verbs to operate on a rich set of nouns. If we're using HTTP, the verbs correspond to HTTP methods (GET, PUT, POST, and DELETE, typically). The nouns are the resources in our application. We name those resources using URLs.

In case of the database implementation the following calls in the RESTful interface were implemented in a common class called ResourceBase which is sub-classed by all other resources in the system.

5. Citations

The surnames of authors and year of publication should be given to the corresponding text. The references should be left aligned in 6-point spacing (after) and 0-point spacing (before) should be there for each reference.

- **Journal articles:**
Pure Testing (Testing Thought Leadership) Creating Real-World Test Cases using Extension to Noun and Verb technique. Wednesday, September 26 CONQUEST 2006.
- **Book**
Cooper, Victor: 2008. *Noun & Verb Technique*. Delhi: Pure Publications.
- **Book Chapter**
Cooper, Victor: 2009. Understanding exploratory method of software testing. In Cooper V, Borah J (eds) *The Paradigms of Testing: a guide to effective testing methodologies* (pp 27-55). Delhi: Pure Publications.

References

- [1] Cooper, Victor: 2008. *Noun & Verb Technique*. New York: Pure Publications.
- [2] Cem Kaner: Feature Matrix. Training Provided at Cigital Inc (formerly RST Corporation)

Biography



KANCHIBHOTLA Kiran Kumar is currently in-charge of testing a large life sciences product. He has over 14 years of experience in software industry, which includes two years of training experience. He has successfully migrated very large applications related to EDA and cellular network planning, from Windows to UNIX and vice versa. He has successfully executed projects in both client-server and web-based paradigms in life sciences domain. He has designed both exhaustive and "good enough" testing strategy for algorithms, multi-user and multi-threaded client server applications, and web-based applications. He has also provided cost-effective testing solutions for large projects and products. He is also experienced in managing projects and product development. He holds a B.Sc. in Electronics.



JONNALAGADDA Srinivas is currently Chief Technical Officer of Siri Technologies Pvt Ltd, Bangalore, India. He has over 15 years of experience in the Information Technology industry. He has provided software solutions to a broad spectrum of businesses ranging from banking and insurance, to ERP and life sciences. At Siri, he has handled software services delivery, knowledge management initiative and SiriBio, Siri's life sciences division. He has published and presented papers in several journals and at various international conferences. Prior to Siri, he had worked for Satyam Computer Services at Hyderabad, and prior to that, for Dun & Bradstreet Software at Chennai. He holds a Masters in Mathematics and a Ph.D. in Computer Science.