2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns

Test-the-REST: An Approach to Testing RESTful Web-Services

Sujit Kumar Chakrabarti
Philips Healthcare Research
Bangalore, INDIA
sujit.chakrabarti@philips.com

Prashant Kumar
Philips Healthcare Research
Bangalore, INDIA
prashant.kumar@philips.com

***Abstract*—Representational state transfer (REST) is an architectural style that has received significant attention from software engineers for implementing web-services due to its simplicity and scalability. By definition, web-services are distributed, headless (lacking UI) and loosely coupled. This presents the implementers and testers of web-services with challenges, which are different from those in testing of traditional software. REST has unique characteristics like uniform interfaces, stateless communication, caching, etc. This also necessitates taking a fresh look at web-service testing specifically in the context of RESTful web-services. A large informatic infrastructure being developed within our organisation is largely based on service-oriented concepts wherein many of the services are RESTful. As a part of a research project named Test-the-REST (TTR), we have developed an approach for testing RESTful web-services of the above infrastructure. The project yielded in a number of novel technical innovations, e.g. a scalable plugin based architecture, an extensible XML based test specification format, a method for reusing and composing test cases for use-case testing etc. A prototype of TTR was used in testing a RESTful service of the above infrastructure early in the construction phase. Many bugs were uncovered resulting in significant value add. In this paper, we present our experience and insights in developing and using Test-the-REST.**

***Keywords*-Web-service; REST; SOA; ROA.**

I. Introduction

Web-services are becoming popular in architecting distributed software solutions due to the flexibility and simplicity of the World Wide Web (WWW). An inherent aspect of the WWW is its resource-orientedness. This means that the architecture of the Web and the recommendation of HTTP [7] – which is the ubiquitous application level communication protocol of the WWW – encourages that entities visible on the web are exposed as resources with their own universal resource identifiers (URI) to be accessed through the uniform interface of HTTP. Roy Fielding, in his PhD thesis [8], coined the term *representational state transfer* (REST) [9], which provides the general conceptual framework for networked applications which resemble the WWW in its essential features, i.e., stateless communication, uniform interface, etc. In the past few years, the idea of REST has been explored widely in the design of web-services. It has been found that since web-services use WWW and HTTP, REST should be the architectural style of choice for them. Web-services implemented following the RESTful architectural style, along with some additional engineering constraints, are known to have a *resource oriented architecture* (ROA) [13].

Testing web applications (websites and web-services) presents with challenges emerging from distributedness, loose-coupling and lack of reliability of WWW as the communication framework. Distributed nature makes both controllability and observability an issue for web-services. This makes fault localisation addedly difficult through testing. The fact that web-services are headless (i.e., they do not have UI) makes manual testing difficult as both inputs and outputs are not amenable to inspection by humans. The *publish-find-bind* paradigm followed by web-services results in loose-coupling between subsystems. Though, this aspect has its advantages, it also becomes the source of tricky failures due to runtime data-incompatibility. Finally, often, WWW is the computing infrastructure used by web-services as their communication channel. WWW is simple and scalable. But it is not very reliable. It becomes an added task for testing to distinguish between failures arising due to system faults or the unreliability of WWW.

After evaluating several web-service tools and standards (Section II), we concluded that our testing needs were very specific to our development scenario. Though there seems to be ample resources available for conducting simple and general web-service testing, support for requirements specific to RESTful web-service testing was not direct, if at all available. None of the available tools and standards suffices our needs, particularly related to customisability and future extensibility. Consequently, we found it feasible to undertake the development of such a tool in-house.

Test-the-REST (TTR) is the name by which we call an HTTP testing tool we have designed for testing RESTful web-services. It was developed as a part of a research project by the same name (TTR). The primary objective of the project was to provide a groundwork for a test framework for RESTful web-services being developed within our organisation. As a part of this project, we gathered the requirements both specific to our scenario and for testing RESTful web-services in general, surveyed the state-of-the-art, prototyped TTR and piloted it for testing within a live development project. The findings were positive enough for the organisation to decide to continue the development of

TTR as a full-fledged engineering product by a separate development team.

Below, we present a list of features that we find desirable for a framework for testing RESTful web-services to have.

Functional testing: Version compatibility, caching, partial Get, conditional Get, HTTP status codes, connectedness testing, etc.

Non-Functional testing: Performance testing, load-/stress/scalability testing, availability and reliability testing, etc. In non-functional testing, an overarching requirement is to localise the fault. For example, in performance testing it is important that the testing helps reveal the bottlenecks within the system.

An extended version of the above list was used to decide the basic design approach both of the tool and the test specification language.

In this paper, we present the best practices and lessons learned during the TTR project. The specific technical contributions of this work are the following:

1) An extensible pluggable architecture for a framework for testing RESTful web-services
2) An extensible XML based test specification language
3) A method of use-case testing through composable test cases
4) A list of requirements for TTR and design approaches for the same

Rest of this paper is arranged as follows: In Section II, we present some related work. Section III presents the overall architecture of TTR. Here, we also provide the design approaches we employed in implementing some of them. Section IV presents the elements and design rationales of the XML-based test specification language which we developed for TTR. Section V takes the discussion forward to talk about an advanced feature that allows reuse of test cases and use-case testing with TTR. In Section VI, we share our experiences in a pilot study of TTR. We conclude with pointers to future work in Section VII.

II. Related Work

Fiddler [3] is a free web debugging tool developed in Microsoft. It provides convenient features, e.g. HTTP traffic monitoring, UI based composition of HTTP request. We did not find its support of persistent test cases to be extensible for automatic test generation and for expressing wide variety of test requirements. WebInject [5] is a Perl-based HTTP testing tool with support for extensible test case format. SoapUI [2] is a commercially available tool with a freely downloadable version available. Version 2.5.1 has some support for testing RESTful web-services. It did not have some features which was considered important for testing in our organisation. JMeter [1] is a free tool from Apache. It is very stable and has good documentation. Though it has some support for functional testing, it is primarily designed for performance testing.

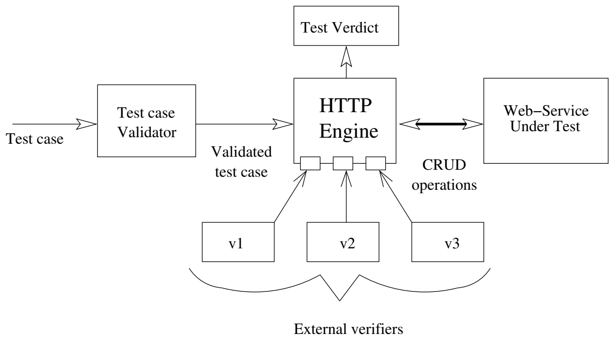

Figure 1. Schematic of TTR

JUnit [10], NUnit and other xUnit family tools are well-known and widely used in industry. But the fact they aim unit-testing and their tight-coupling with implementation language of the SUT makes it difficult to use them for testing web-services. [12] provides a BPEL [14] based framework for testing web-services which shares certains features with xUnit family tools. But it is primarily a white-box testing tool suitable for unit-testing.

Architecturally, TTR resembles WebInject the most according to us. Moreover, we have used the idea of test cases written as external XML files as done in WebInject. However, beyond this, our approach is distinct from WebInject's. The idea of automatic test case generation is an important consideration for us. Because of this, we have taken care that TTR test specification language aligns with WADL [11], which we perceive as a rapidly emerging favourite for specifying RESTful web-services. WSDL 2.0 [6] also provides support for REST, but is not exclusively designed for it.

III. TTR Architecture

The general architectural scheme of TTR is shown in Figure 1.

The input test case is written in a test specification language based on XML. This language includes constructs required to express the test details needed to execute the test. We present more details about this test specification language in Section IV.

The test case validator module does some sanity check on the test case file and passes it on the HTTP engine. The HTTP engine sends the corresponding inputs to the web-service under test (WUT) using HTTP methods (viz. POST (for Create), GET (for Read), PUT (for Update), and DELETE (for Delete), collectively known as CRUD operations). The HTTP response is validated as per the test case and test verdict is given.

Validation of the response depends on a variety of factors, e.g. media-type of the response, HTTP response status code, etc. For various media-types [4], TTR uses relevant media-type validators, which can be developed externally and plugged in as needed (viz. v1, v2 and v3 shown in Figure 1).

```
<testcase>
  <id>TTR-1</id>
  <URI>http://localhost/cgi-bin/hello-sujit.cgi</URI>
  <atomicity>atomic</atomicity>
  <method>GET</method>
  <representation mediatype="XML"/>
</testcase>
```

Figure 2. A test case

For example, consider a case where we expect the response to be a valid XML document. This will be mentioned appropriately in the test case (see Section IV) in the test case. The HTTP engine will invoke an XML validator and pass the test only when the validator reports a successful validation. The above XML validator is to be provided to the HTTP engine as a plugin. In case the validator corresponding to the given media-type is unavailable, TTR will report this and will proceed or abort depending on the user direction. The idea of plugin validators took the validator development activity out of the critical path of the test framework development.

IV. Test Specification Language

The design principle of the test specification language was that it should be able to specify most of the test requirements mentioned in Section I. Extensibility was important as we expected that TTR development would be long-drawn and requirements would evolve. Another important requirement was that it should be machine-processable (in terms of generation and consumption). This led us to decide upon an XML-based format which drew from WebInject's [5] test case language and WADL [11].

Figure 2 shows a simple test case in our test specification language. The id of the test case is `TTR-1`. A HTTP GET request will be posted in this test case to a URI `http://localhost/cgi-bin/hello-sujit.cgi`. The mediatype of the response body, given by the `mediatype` element of the `representation` element is `XML` which means that the response to the HTTP request sent to the above URI is required to be a well formed XML document. Similarly, a test case may have `representation` of `mediatypes` like `PLAINTEXT`, `JSON`, `XML-SCHEMA`, etc. If the `mediatype` is `PLAINTEXT`, the response is required to match character by character with an expected string which also has to be provided along. Similarly, if the `mediatype` is `XML-SCHEMA`, then the response should conform to a XML schema which should be provided along.

A. Composite Validation

Consider a case where the media-type of the response representation is not known. For example, the tester knows that the response body will either be a well-formed XML document or a JSON object, but is not sure which of these.

```
<testcase>
  <id>TTR-COMPOSITE-VALIDATION</id>
  <URI>http://localhost/cgi-bin/hello-sujit.cgi</URI>
  <atomicity>atomic</atomicity>
  <method>GET</method>
<response>
  <representation mediaTypeExpression="xml | plaintext"/>
  <mediaType id="xml" value="XML"/>
  <mediaType id="plaintext" value="PLAINTEXT"
        src="D:\ResultValidationData.txt"/>
</response>
</testcase>
```

Figure 3. Composite validation

```
<testcase>
  <id>TTR-3</id>
  <atomicity>composite</atomicity>
  <sequence>
    <testcase>
      <src>TTR-1.xml</src>
    </testcase>
    <testcase>
      <src>TTR-2.xml</src>
    </testcase>
  </sequence>
</testcase>
```

Figure 4. A composite test case

In other words the tester would like the test case to pass if the response body is either of these. TTR specification language allows such a specification. An example is shown in Figure 3. The snippet says that the response representation should either be an XML or a JSON. The language allows arbitrary `mediaTypeExpression` expressions formed with atomic mediatypes and logical connectors (`"&"` and `"|"`).

V. Composite Test Cases

While the test case in Figure 2 corresponds to a single HTTP request-response with the WUT (indicated by the keyword `atomic`, we could also have test cases which are a sequence of such atomic test cases. An example of such a composite test case is shown in Figure 4. Note that for the test case `TTR-3`, `atomicity` is of type `composite`. It is a `sequence` of two test cases. The execution of the test case TTR-3 comprises the execution of the two test case TTR-1 and TTR-2 in that order. `TTR-NT-3` returns with a *pass* if both `TTR-1` and `TTR-2` pass.

The main objective of sequential execution of many test cases is to execute a complete use case which comprises of multiple individual interactions with the WUT. Often, there is data dependency between these test cases. For example, consider a web-service which allows the user to maintain a list of books. The book list can be viewed by doing an HTTP GET on the following URI: `http://booklisthost.com/books`. An HTTP POST on this URI with necessary request content (containing the data pertaining to the book) creates a new entry of a book with the details as provided by the user.

A new resource is created corresponding to this newly added book whose id is, say, book1. This resource can be read on the URI:
http://booklisthost.com/book/book1.

The test cases corresponding to the above two operations are shown in Figure 5(a) and (b) respectively. Figure 5(c) shows the test case which invokes these two test cases to exercise the create and read use case. TTR-BL-1 (shown in Figure 5(a)) does an HTTP POST on the given URI. The content of the HTTP request contains the text "War and Peace|Leo Tolstoy". The absense of representation element means that we do no validation on the response content: any response will pass (as long as the HTTP status code is '201(Created)'. TTR-BL-2 does an HTTP GET on the given URI and checks for nothing (no representation element).

Test case TTR-BL-3 should pass since both its constituent test cases check for nothing and hence pass trivially. However, the URI of TTR-BL-2 contains book1, the id of the newly created book. TTR-BL-3 will pass once (by chance) if the id of newly created book is indeed book1.The intent is to do a read on the URI of the book resource created during the preceding execution of TTR-BL-1. The id book1 is hard-coded in TTR-BL-2. The id of the book created during the execution of TTR-BL-1, which happens just before its own execution, is different everytime TTR-BL-1 executes. It does not adapt to this fact.

TTR-BL-2 should be parameterised with the value of the book id which should be received from the preceding execution of TTR-BL-1. In this article we present constructs in the TTR language that allow sewing together of test cases so that data can be passed around between them – both as arguments passed to test cases and values returned from them.

```
<testcase>
  <id>TTR-BL-1</id>
  <URI>http://booklisthost.com/books</URI>
  <atomicity>atomic</atomicity>
  <method>POST</method>
  <input>War and Peace|Leo Tolstoy</input>
</testcase>
```

(a)

```
<testcase>
  <id>TTR-BL-2</id>
  <URI>http://booklisthost.com/book/book1</URI>
  <atomicity>atomic</atomicity>
  <method>GET</method>
</testcase>
```

(b)

```
<testcase>
  <id>TTR-BL-3</id>
  <atomicity>composite</atomicity>
  <sequence>
    <testcase>
      <src>TTR-BL-1.xml</src>
    </testcase>
    <testcase>
      <src>TTR-BL-2.xml</src>
    </testcase>
  </sequence>
</testcase>
```

(c)

Figure 5. Test cases for a book list web-service: (a) Create a book; (b) Read a book; (c) Create and read a book

A. Named Entities

There are two types of named entities in TTR language:

1) **Input parameter:** An input parameter gets its value from the calling test case and maintains the value through its lifetime. An input parameter is declared using the 'param' tag. For example, an input parameter named x is declared as:
<param>x</param>.
2) **Variable:** A variable is initially uninitialised. It gets its value from the HTTP response of the WUT or from return values of a child test case. A variable is declared using the 'var' tag. For example, a variable named x is declared as:
<var>x</var>.

There are no global variables or constants allowed. The scope of all input parameters and variables is the containing test case and their lifetime is during the execution of the same.

Usage: Named entities are used within strings by enclosing them within braces. In TTR-5-1 (Figure 6), we have an example of the use of input parameters x and y in the URI field:
<URI>http://localhost/{x}/{y}</URI>.

B. Data Passing between Test Cases

We solve the problem of data passing between test cases using the ideas from traditional programming languages: data gets passed into test cases as parameters, and out of test cases as returned values. Both features can be understood exactly as in the context of data-passing to and from functions in procedural programming languages like C.

C. Parameterised Test Cases

Figure 6 shows an example of how values get passed into a test case. TTR-5-1 is an atomic test which receives two input parameters, namely, x and y. TTR-5-1 then uses these values in its URI which is:
http://localhost/{x}/{y}.
TTR-5 is the composite test case which makes a call to TTR-5-1. Two strings – "cgi-bin" and "hello-sujit.cgi" – are passed as parameters in

```
<testcase>
  <id>TTR-5-1</id>
  <param>x</param>
  <param>y</param>
  <atomicity>atomic</atomicity>
  <URI>
    <URI>http://localhost/{x}/{y}</URI>
  </URI>
  <method>GET</method>
  <representation mediatype="XML-WELLFORMED"/>
</testcase>
```

(a)

```
<testcase>
  <id>TTR-5</id>
  <atomicity>composite</atomicity>
  <sequence>
    <testcase>
      <src>TTR-5-1.xml</src>
      <param>cgi-bin</param>
      <param>hello-sujit.cgi</param>
    </testcase>
  </sequence>
</testcase>
```

(b)

Figure 6. Passing data into a test case: (a) Parameterised test case; (b) Invoking test case

this call. Thus, when `TTR-5-1` begins executing, its input parameter `x` gets the value "`cgi-bin`" and `y` gets the value "`hello-sujit.cgi`. Replacing these values in the URI of `TTR-5-1`, the concrete URI becomes: `http://localhost/cgi-bin/hello-sujit.cgi`.

D. Returning Values from Test Cases

Figure 7 shows how values are returned from a test case. `TTR-6-1` (Figure 7(a)) is the atomic test case which returns two strings: "`true`" and "`false`". These are received by `TTR-5` into two variables: `x` and `y` respectively.

E. Use Case Testing

The test cases for specifying this test are shown in Figure 8.

F. Design Notes

Here, we mention some important design points of composite test cases:

- The test cases can be composed to any level of nesting.
- TTR handles these test cases in a manner similar to program run-time, i.e. by maintaining a program stack. A stack frame is created corresponding to the currently invoked test case. The stack frame is popped when the test case finishes execution. If a test case fails, TTR raises an appropriate exception with the stack dump. The same proves very useful in debugging both the test cases and the WUT.

```
<testcase>
  <id>TTR-6-1</id>
  <atomicity>atomic</atomicity>
  <URI>
    <URI>http://localhost/cgi-bin/hello-sujit.cgi</URI>
  </URI>
  <method>GET</method>
  <represenation mediatype="XML-WELLFORMED"/>
  <return>true</return>
  <return>false</return>
</testcase>
```

(a)

```
<testcase>
  <id>TTR-6</id>
  <atomicity>composite</atomicity>
  <var>x</var>
  <var>y</var>
  <sequence>
    <testcase>
      <src>TTR-6-1.xml</src>
      <return>{x}</return>
      <return>{y}</return>
    </testcase>
  </sequence>
</testcase>
```

(b)

Figure 7. Returning data from a test case: (a) Test case returning value; (b) Invoking test case

- At present TTR does not allow recursive test cases, i.e. a test case which invokes itself. If there is is a cycle in the test case call graph, the same is detected before the tests start running, and TTR aborts testing.

G. Novel Features of TTR Test Specification Language

Here is a list of features of TTR test specification language which do not appear together in any other language or format.

1) It fully aligned for specifying tests for RESTful web-services.
2) Logical operators can be used to combine test criteria.
3) It allows test case reuse and use-case testing by composing test cases to arbitrary levels of nesting and using named-values for data-passing.

VI. Experimental Validation

The TTR prototype was done using C# language on .Net platform. It was piloted for testing an in-house RESTful service, named Job Service (JS), early during its construction phase.

The first round of pilot involved manual generation of a test suite which was created by our researchers in association with architects of JS. This suite initially contained about 200 test cases which grew to about 300 over the duration of the study. For about a month, this test suite was executed daily by all individual developers in JS locally on their machines

```
<testcase>
  <id>TTR-BL-1-UC</id>
  <var>uri</var>
  <URI>http://booklisthost.com/books</URI>
  <atomicity>atomic</atomicity>
  <method>POST</method>
  <input>War and Peace|Leo Tolstoy</input>
  <response>
    <header>
      <field name="location" value="{uri}"/>
    </header>
  <response>
  <return>{uri}</return>
</testcase>
```

(a)

```
<testcase>
  <id>TTR-BL-2-UC</id>
  <par>y</par>
  <URI>{y}</URI>
  <atomicity>atomic</atomicity>
  <method>GET</method>
</testcase>
```

(b)

```
<testcase>
  <id>TTR-BL-3-UC</id>
  <atomicity>composite</atomicity>
  <var>x</var>
  <sequence>
    <testcase>
      <src>TTR-BL-1.xml</src>
      <return>x</return>
    </testcase>
    <testcase>
      <src>TTR-BL-2.xml</src>
      <par>x</par>
    </testcase>
  </sequence>
</testcase>
```

(c)

Figure 8. Use case testing of Book-List service: (a) Create a book; (b) Read a book; (c) Create and read a book

using TTR. Every week, code used to get checked into the archive. The same test suite used to be run on the merged archive. The test execution used to take less than 5 minutes to run using our prototype. Between 5 to 10 test cases would routinely fail every day during the complete cycle all of which corresponded to real system bugs.

The second round of pilot involved automatic generation test cases corresponding to an exhaustive list of all valid combinations of query parameter values. This resulted in a test suite with 42,767 test cases. The first round of testing caused 38,016 test cases to fail. This took less than half an hour to run. Most of the failures reported corresponded to two major bugs in JS. On fixing them and re-executing the tests, the failure count came down to 1781. Hereafter, the development team agreed upon using the prototype tool for further development time testing of their product without our assistance. We deemed the pilot study complete and successful at this point, and the testing activity was tranferred to the JS development team.

VII. Conclusions

A. Constraints and Limitations

For full utilisation of all TTR capabilities, we require the service under test to be hosted in a controlled environment. TTR causes persistant state change in the web-service under test which it might be difficult to revert in the live deployment. Also, TTR approach is black-box approach. Hence, directly, it is not possible to use TTR to localise faults to internal components or subsystems.

B. Future Work

Our current research focuses on defining a formal web-service specification language and designing algorithms to automatically generate TTR test cases from web-service specifications written in it.

C. Closing words

In this paper, we have shared our design approach and main results of our requirement analysis step in TTR. We hope that this will prove to be a good starting point for those who wish to implement their own testing tool.

We decided to invest our efforts in defining a a test specification language. As opposed to a GUI based approach of testing this resulted better automation in test execution and opened ways for further research in automated test case generation. This facilitated early adoption of the tool by development teams. We present this design decision as an important best practice.

Though web-service testing has matured significantly, support for REST specific testing seems limited. In the current scenario, we think that developing in-house REST testing tool should work out for medium to large organisations due to benefits of flexibility, customisability and maintainability without dependence on external support.

Acknowledgment

We cordially thank our business-unit partners from Philips Healthcare – Sukanta Bhatt, Vijayananda and Shivakumar – for the numerous discussions during requirement analysis and demo sessions during prototyping.

References

[1] Apache JMeter, http://jakarta.apache.org/jmeter/ [accessed, September 11, 2009].

[2] eviware soapUI, http://www.soapui.org/ [accessed, September 11, 2009].

[3] Fiddler http debugger, http://www.fiddlertool.com/ [accessed, September 11, 2009].

[4] Mime media types, http://www.iana.org/assignments/media-types/ [accessed, September 11, 2009].

[5] Webinject - web/http test tool, http://www.webinject.org/ [accessed, September 11, 2009].

[6] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web service description language 2.0, http://www.w3.org/tr/wsdl20/ [accessed, September 11, 2009], 2006.

[7] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical report, 1998.

[8] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[9] R. T. Fielding, D. Software, and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2:115–150, 2002.

[10] E. Gamma and K. Beck. Junit. http://www.junit.org/ [accessed, September 11, 2009].

[11] M. Hadley. Web application description language, http://wadl.dev.java.net/ [accessed, September 11, 2009], 2006.

[12] P. Mayer and D. Lübke. Towards a bpel unit testing framework. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 33–42, New York, NY, USA, 2006. ACM.

[13] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.

[14] A. T. and C. F. et al. Business process execution language for web services 1.1. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ [accessed, September 11, 2009], July 2002.