A framework for testing RESTful web services

Conference Paper · January 2010

DOI: 10.1109/ITNG.2010.175 · Source: DBLP

Hassan Reza
University of North Dakota
119 PUBLICATIONS **214** CITATIONS
SEE PROFILE

David Van Gilst
University of North Dakota
5 PUBLICATIONS **9** CITATIONS
SEE PROFILE

A FRAMEWORK FOR TESTING *RESTful* WEB SERVICES

Hassan Reza and David Van Gilst
University of North Dakota
School of Aerospace Sciences
Grand Forks, ND, USA
reza@aero.und.edu

Abstract- **While web services greatly reduce the cost and complexity of integrating systems; they also introduce a number of additional challenges for testing because web services require standards, *API*, and/or communication protocols and architecture that are not fully supported by traditional software testing methods and tools. In this paper, we discusses a software framework for providing a variety of test inputs for web service depended software, including predetermined test cases, replay of previous data, and generation of random test cases through data perturbation from a provided template.**

Key Words: Software Testing, Web Testing, WSDL (Web Service Description Language), Web Based Service Oriented Architecture

1. Introduction

With the ubiquity of HTTP and XML in modern computing, web services have become increasingly popular method of managing inter-system communications over networks. AWeb services-based architecture results in complex system that demands coordination, communications, and distributions among loosely coupled system with dependent users in order to provide scalability, flexibility and extensibility. Therefore, testing of web-services system demands constant availability and reliability; they are constructed from reusable components. This, in turn, poses serious issues for testers because Web services require standards, API, and/or communication protocols and architecture that are not fully supported by traditional software testing methods [3] and tools.

In the simplest form, web services refer to the software components that support interoperability among computer systems connected over communication network. The key elements of web service include, HTTP, XML, SOAP (Simple Object Access Protocol), UDDI (Universal Description, Discovery and Integration), WSDL (Web Services Description Language). Service Oriented Architectures (SOA) constructed atop the HTTP stack allows architecture agnostic intercommunication between systems with fewer weighty and expensive middleware systems.

By providing a common, easily accessible interface, web based service oriented architectures allow for the construction of large systems from reusable components, often much more quickly and with greater flexibility than a traditional system might be constructed. This feature, in turn, provides additional testing challenges, in that components of a complex system may be created by teams that have little communication with one another, and who have little idea about how their services will ultimately be used for.

Representational State Transfer (*REST*) [13] architectural style for distributed hypermedia systems provides a set of constraints with emphasizes on the scalability of component interactions, generic interfaces, reusable components in order to improve security and overall performance of a system. In particular, within the airborne science community, there has been a push for greater integration of live data from instruments, aircraft, satellites and ground based data sources to improve situational awareness and more effectively make use of very limited flight time. In the past when there was little communication of real-time data, custom solutions were used and integration testing was done aboard the aircraft at installation time. As the requirements for increased inter-systems communication as grown, there has been a move towards standard networking solutions, many of them built atop RESTful web services.

This paper proposes testing challenges, as these reusable components are developed in parallel leading up to any given mission or deployment by widely disparate groups at universities, research institutions and/or government agencies. This paper will therefore attempt to explore a framework for providing a generalized test harness to simulate RESTful web services that are otherwise unavailable to a developer using either previously generated data, provided test cases or random test cases generated through data perturbation.

2. Background

Service Oriented Architectures are designed to wrap up the contents of the multitude of existing business processes and software packages that exist across an enterprise and expose their data and functionality in a in a highly standardized manner. This architecture is designed to allow the more rapid development of new software capabilities based on existing capabilities and assets, regardless of the platform that these capabilities exist on. To do this, it addresses the problem of creating a loosely-coupled, standards-based and protocol-independent distributed computing environment [7]. The most popular SOA services in use today utilize the Web Services Description Language (WSDL), and Simple Object Access Protocol (SOAP).

SOAP is a standard protocol for exchanging structured information in web services. All SOAP messages are based on XML documents and fragments. A SOAP message is a one way transmission between SOAP nodes, from a sender to a receiver. These messages are combined to build up request/response interactions and more complex, longer conversational exchanges [9].

SOAP has a number of advantages and disadvantages when compared with competing protocols such as CORBA - the use of HTTP and XML makes for a protocol that operates easily through existing firewalls and proxies, and which is largely platform and language independent. However, the verbosity of XML often makes for inefficient data transfer and decoding, necessitating the use of additional standards such as the SOAP Message Transmission Optimization Mechanism (MTOM) to efficiently transmit binary data [10].

WSDL is the W3C's Web Services Description Language. It is an XML-based language that provides a framework for describing Web Services. WSDL defines services as being a collection of ports, or network endpoints. A WSDL definition generally includes the following objects [11]: Services , Ports, Bindings, Port Type, Operations, Message, Elements, and XSD Files. Old versions of WSDL did not support the full operation set of HTTP and were unsuitable for the modeling of RESTful web services - version 2.0, currently in development will be more suitable for RESTful web services.

REST is an architectural style for building distributed systems based on four principles: Resources are identified through URIs, which provide a a unique global address for resources and service discovery. For RESTful web services, this means that each service and resource is accessed through a URL[1]. Resources are created, read, updated and deleted through a set of four actions: PUT, GET, POST and DELETE [1]. Messages are self descriptive. Some Metadata about resources is generally available, but messages are not constrained to a set format as with WSDL and SOAP. Messages from a RESTful web service can be in pretty much any format, including JSON, XML, plain text or image formats [1]. Stateful interactions through hyperlinks. Each interaction with a resource is stateless[1].

RESTful services are less constrained both in their inputs and outputs than are SOAP based web services, which have a more complex XML based message format. This makes integration testing difficult becuase there is a much wider array of input and output formats to be considered. Where a SOAP response will be contained in an XML package, a response from an RESTful web service might be anything - an XML document or fragment, an HTML document, streaming video, or a JPEG to use a few examples [4]. Thi is to some degree self-described by the http content-type: header.

Key limitations of the *REST* architecture include [12]: Messages are one-shot, Rest architecture contains no method for transferring data to a group of components, and REST is inherently client-server and proper testing of C/S can be very complex and costly because of complex and dynamic interactions among clients and servers. This complexity requires proper synchronization among multiple distributed clients and servers connected by the network.

3. Related Works

Mayer and Lubke [2] provide a good start in terms of planning an test framework for RESTful [13] web services by discussing a similar framework for services created with the more restrictive BPEL(Business Process Execution Language). The paper is not directly applicable as BPEL services are based on SOAP rather than being RESTful, but the paper still provides much of interest.

Mayer and Lubke [2] also highlight the need to bring formal testing and validation into the area of web applications, pointing out that many of the existing tools for BPEL Testing concentrate only on black box testing, to the exclusion of any kind of white box testing. The general layered testing architecture in [2] provides is a starting point for the creation of a testing architecture for RESTful services. In [2], the internal functionality of a web service can be unit tested through traditional means, but the web application infrastructure creates a very transparent communication between processes where you can easily examine program state. This is a significant advantage over programming in an environment where such introspection often requires a debugger.

In [8], they outline a data perturbation system that generates new test cases using previous test cases as a template. In this case they created a Regular Tree Grammar to represent the formal XML Schema. From this they generated new test cases for web services. This paper also provides a useful discussion of data types and their boundary values within such a data perturbation scheme [8].

In [5], they discussed a number of areas in which web services create additional testing problems. They conclude that web services need to confront additional problems of scale, such as work flow complexity, volume of data, number of nodes, complexity of operations, and differences in usage patterns.

4. The Approach

The REST architecture as a means of implementing service oriented architecture, facilitates the development of loosely coupled environments in which a data and computation are frequently widely spread across separate systems deployed and controlled my multiple different stakeholders. This presents a challenge when it comes to the development of unit and integration tests because a developer may not have control, or access to other parts of the system. Moreover, in a service oriented architecture, the other portions of the system may be considered completed, with other stakeholders uninterested in playing around with exhaustive integration testing or alternately, the other portions of the system that may not yet be developed.

The development of stubs to simulate other web services is a long and laborious process that can consume significant amounts of developer time, and can be an error prone process. By creating a framework for building stubs and specifying test data, the process of unit testing applications which rely on these web services can be much more easily facilitated.

Moreover, by providing a system with which a web service provider can specify a test interface and test data, the process of integration testing can be eased by allowing developers of software that interfaces with the web service to do extensive testing, such as boundary value and equivalence class testing, as well as testing against randomly generated and prerecorded data before the actually testing the system as a whole.

In the Airborne Science community, there is a constant push towards ever greater integration of real-time data sources to improve *situational awareness* and make more effective use of extremely limited and expensive flight time. In the past, communication between the platform and the instrument was relatively simple - the platform provided a small amount of "housekeeping" data, and perhaps a timing signal via NTP or a dedicated system such as IRIG.

As satellite communications systems have become more ubiquitous and instrument control systems have become more advanced, the potential for greater system integration has exploded. Consequently modern deployments are rapidly advancing towards the creation of a single advanced sensor web spanning multiple aircraft, satellites and ground stations. A service oriented architecture based on RESTful web services is an ideal system for handling much of this inter-system and inter-architecture communication.

Integration takes place over a very short period of time, often a period of only a few days or a couple of weeks, and the system as a whole exists for only a few weeks before everyone packs up and goes home. Consequently it is fairly important that each contributor shows up with software that is basically functional and ready to go.

All of the above mentioned issues present a significant testing problem becuse each component of this system is developed by a different stakeholder, with little communication between teams. It is therefore necessary to provide each of these developers something that they can test against. A framework that can be used to build test stubs for each of the anticipated web services in use in the larger system allows each developer begin testing each of their interfaces so that before integration testing begins, they're at least close to the mark.

4.1 Validation of Service Calls

The first requirement of such a test framework is the need to be able to specify what service calls are available and what their legitimate inputs are. For RESTful services, this means defining which URI's are available to be used, what parameters those URI's take, and which of the basic HTTP commands are used to access those URI's.

4.2 Specifying available interfaces

Specification of available URIs can be done via XML, with a single script responding to all requests and emulating the appropriate response based on the file path requested. At the most basic level, the framework needs to specify which files are available, and which HTTP commands can be used on them.

```
<service specification>
 <service name="housekeeping" GET="yes"
POST="yes">
     <description>
          Returns stored housekeeping data based on
positional or temporal limits
      </description>
      <filepath>
        /dc8/housekeeping.cgi
      </filepath>
    </service>
  <service name="NadirVideo" GET="yes"
POST="yes">
     <description>
          Returns a chunk of video from the downward
facing camera based on a lat/lon bounding box or
start/stop time.
      </description>
      <filepath>
        /dc8/nadircam.cgi
      </filepath>
    </service>
</service>
```

Figure 1: Input Specification

Figure 1 provides a very basic level of input specification. A pair of services are specified - one which returns basic housekeeping data and one which returns video from a camera. There is no information for verification of parameters, or for the specification of a return parameter, but even at this level a minimal amount of verification can be provided - if there is an attempt to access a URL that is invalid, a 404 can be returned. Access denied can be returned for an HTTP method which is not allowed. In either event, invalid requests can be rejected and logged.

4.3 Verification of input parameters

Verification of input parameters is the next step after verifying that a valid resource has been accessed. This is a bit more complex than handling the incoming URLs, as there is a higher degree of freedom than in your input domain than with the URLs. Each GET or POST parameter needs to be considered for whether it's a valid parameter, whether parameters are present in an appropriate combination, and whether the values are in fact valid. A possible specification for parameter values:

```
<service name="housekeeping" GET="yes">
  <description>
    Returns houskeeping data
  </description>
  <filepath>
    /dc8/housekeeping/cgi
  </filepath>
  <parameter name="north_lat" type="latitude"/>
  <parameter name="south_lat" type="latitude"/>
  <parameter name="east_lon" type="longitude"/>
 <parameter name="west_lon" type="longitude"/>
</service>
<parameter_type name="latitude">
  <regex>^(\d{1,2}\.\d{0,5})$</regex>
  <match num='1' type='float' max='90.0' min='90.0'/>
   </regex>
</parameter_type>

<parameter_type name="longitude">
  <regex>^(\d{1,3}\.\d{0,5})$</regex>
  <match num='1' type='float' max='180.0' min='-
180.0'/>
   </regex>
</parameter_type>
```

Figure 2: Example of Specification for Parameter Values

With this specification we can verify the input to a simple service that accepts only a pair of latitudes and longitudes. Each parameter is defined with a type definition. The type definition has a regular expression for separating any expected values for the string, providing a first level of validation - if the input provided does not match the regex, something is wrong.

Once the regex has been used to extract values from an arbitrary text string, those values can be validated - in this case by checking that the floating point value provided is within the valid range for a latitude and longitude.

Other data types need to be validated based on appropriate parameters for the data type. A string for instance, can be validated based on length, or matched to another regular expression. Enumeration or boolean types can be checked to ensure that they have one of the prescribed values.

For XML parameters, we have an entirely different approach available. Provided with a DTD, the framework can first validate an input parameter to ensure that it matches the document type definition, then validate the content of each tag using the same type of rules used to evaluate the parameters extracted using regular expressions in the previous example.

Finally, input values are saved for later use in generating appropriate test output to return to the web service consumer.

4.4 Generation of test output

As discussed in [2], the authors describe two basic approaches to the automatic generation of test data:

- Data-centered approach where fixed data is used, input data is compared against predefined inputs and outputs are returned from pre-recorded data. This approach has significant utility, particularly when you have data from past activities that you know to be representative of the "Real World".
- Logic centered approach – The use of a fully fledged programming language to express test logic. A program is executed for each input to generate some output. This approach also has merit and will see usage in this framework, but a completely logic centered almost amounts to re-implementing the web service you would be simulating.
- In [6], the authors provide a third option in exploring data perturbation. A little of each option is necessary to provide for the entire test scenarios to be covered by this framework. This example uses the MochiKit AJAX tools for the sake of simplicity (See Figure 3).

```
function update_aims_temp() {

deferred = doSimpleXMLHttpRequest
("http://web/aims_temp.cgi");
deferred.add_callback(callback)
}
function callback(Result) {

  value =   Result.responseText;
  temp_display.innerHTML = value;

  if(value > 50) {
          temp_display.background = red;
  } else {
temp_display.background = blue;
  }
}
```

Figure 3: Using MochiKit AJAX tools

In order to facilitate white-box testing of this pair of functions, the framework needs to take a data centered approach and replay a set of pre-chosen values. In this case, a good set of boundary value test cases might be 0, 50, and 51.

First a template (see Figure 4) needs to be specified:

```
<output_template>
    <template>
  $temperature
    </template>
    <data_source>
 temperature_values.txt
    </data_source>
</output_template>
```

Figure 4: Simple Template

The data values themselves are stored in a comma separated value (CSV) text file with the variable name as a column heading: Temperature: 0, 50, 51.

For each subsequent call to the web service, the next line in the file is returned, thus facilitating unit testing of the callback function, and the testing of the update_aims_temp function and it's callback as a single unit. This is a simple example, but a prerecorded data sample could easily be several hundred test cases, or even thousands of cases recorded from previous usage of the system.

Many web services require more complex templates (see Figure 5) to adequately simulate the returned data. Another more complex example would return between an XML block with a header and multiple data elements. Such a template might look like this:

```
<datablock>
       <head length=[pass = 2, len(message],
datapoints=[pass =1, rand(1,10)]>
      <datapoints>
  [pass = 1, gen_datapoints(datapoints)]
      </datapoints>
</datablock>
```

Figure 5: Complex Template

Here we have three items that are substituted into a slightly more complicated XML structure - On the first pass we substitute in first a random number of data points (between 1 and 10), then give that value to the custom "gen_datapoints" function to crank out an appropriate number of datapoints, whether randomly generated or pulled from a prerecorded source.

On the second pass, the system fills in the message length - a parameter that can't be generated until later portions of the message are already generated. By recursively processing the message until no unprocessed commands remain, the system can construct complex XML message responses. By extracting and storing parameters from the input, these can even be made to be made fairly appropriate. Datapoints could be extracted from the function input to provide a requested number of test datapoints for instance. In addition to filling in templates using predefined cases, it's easy to generate test cases at random using a combination of ranges defined in the test template and input from the user. If a web application consumer calls a function that requires a longitude and latitude, it's easy to define a template rule that returns a floating point value in the appropriate range. This range could even be trimmed based on input parameters provided by the program. By generating random test cases, it is possible to perform long-running stress tests on an application without having to define hundreds or thousands of data points and without having cases repeat. This can be very important in testing AJAX applications where browser and memory management quirks occasionally make the behavior of long running applications somewhat unpredictable.

5. Conclusions and future work

The framework defined in this paper fills a niche that is largely untouched by other tools. By continuing to develop this tool, the integration if independently developed web services can be made to proceed more smoothly by providing an interface for developers to test against prior to beginning the integration testing process. By providing a system of recursively processed templates which can substitute either prerecorded data or randomly selected data constrained by either developer provided conditions or by input conditions from the calling program, even relatively complex web services can be simulated adequately for testing purposes.

Further work is needed to carefully define the templating system and the system for creating functions in Python to fill in more complex data types.

6. References

[1] C. Pautasso, O. Zimmerman, and F. Leymann, “RESTful Web Services vs 'Big' Web Services: Making the Right Architectural Decision”, *Proceeding of the 17th international conference on World Wide Web* , ACM, New York, 2008, pp. 805-814.

[2] P. Mayer and D Lubke, “Towards a BPEL unit testing framework”, Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications, *International Symposium on Software Testing and Analysis*, Portland, Maine, 2006.

[3] P. Jorgensen, “Software Testing: A Craftsman's Approach”, CRC Press, New York, 1995.

[4] A. Kesteren, “The XMLHttpRequest Object, W3C Working Draft 15 April 2008”, http://www.w3.org/TR/XMLHttpRequest/, 2008.

[5] S. Krishnamurthi and T. Bultan, “Discussion Summary: Characteristics of Web Sevices and Their Impact on Testing, Analysis and Verification”, ACM SIGSOFT Software Engineering Notes, ACM, New York, 2005.

[6] H. Ludwig, L. Pasquale and B. Wassermann, “REST-Based Management of Loosely Coupled Services”, Proceedings of the 18th international conference on World wide web, ACM, New York, 2009.

[7] M. Papazoglou and W. Jeuvel, "Service oriented architectures: approaches, technologies and research issues", The VLDB Journal - The International Journal on Very Large Data Bases, Springer-Verlag, New York, 2007.

[8] J. Offut, W. Xu, “Generating Test Cases for Web Services Using Data Perturbation”, ACM, New York, 2004.

[9] N. Mitra, Y Lafon, “SOAP Version 1.2 Part 0 Primer (Second Edition)”, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, W3C, 2007.

[10] M Gudgin, N Mendelsohn, M Nottingham, H Ruellan, “SOAP Message Transmission Optimization Mechanism”,http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, W3C, 2005.

[11] E Christensen, F Curbera, G Meredith, S Weerawarana, "Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl, W3C, 2001.

[12] R. Khare, and R. Taylor, "Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems", Proceedings of the 26th International Conference on Software Engineering, IEEE, 2004.

[13] R. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." *PhD. Dissertations*, University of California, Irvine, 2000.