Study on REST API Test Model Supporting Web Service Integration

Hu Wenhui
National Engineering Research Center for Software Engineering, Peking University
Beijing 100871, China
huiwenhui@pku.edu.cn

Huang Yu*
National Engineering Research Center for Software Engineering, Peking University
Beijing 100871, China
hy@pku.edu.cn

Liu Xueyang
National Engineering Research Center for Software Engineering, Peking University
Beijing 100871, China
liuxueyang@pku.edu.cn

Xu Chen
National Engineering Research Center for Software Engineering, Peking University
Beijing 100871, China
xuchen.@pku.edu.cn

***Abstract*—Representational state transfer (REST), which is a style of software architecture with simplicity and flexity, has been widely adopted in web services. To solve challenges like multidemensional API validation requirements, call sequencing and organization of test cases, data dependency between test cases, this thesis proposes a REST API testing model with an expressive description language based on JSON.**

Keywords—REST API, test model, Web Service intergration

I. INTRODUCTION

Visible entities on Web are exposed as resources, which are identified by a universal Uniform Resource Identifier (URI) and accessed by Hypertext Transfer Protocol. As an architecture style, REST[1] is obviously simpler than SOAP[2] and XML-RPC which are complicated. In the past few years, REST was applied in a wide range in Web service. More and more Web services adopted REST style in their designs and realization, and this trend is being accelerated by the rapid development of mobile business. With the development of agile software and popularization of DevOps, it is difficult to maintain GUI testing under such faster iteration speed and change in demands. In addition, API is used as a stable complicated application logic interface exposed outside, so API testing is more suitable for automated testing and integration testing. Because REST service has features [3] of distributability, loose coupling and uncertainty, the testing of REST API will confront some challenges. The paper will do some relevant research on the testing of REST API.

The verification requirements of REST API are complicated, not only including various correctness tests, such as response content and response header[4], but also including performance tests; the paper will focus on how to organize execution sequence of test cases, test data dependence between cases, and lower the difficulty of manual testing for Web service.

II. DESCRIPTION OF TEST CASE

A test model firstly needs to give a complete description of the test case. A test case that has been analyzed and sorted out and complies with standards is required to consist of the following:

Table 1: testing standard language for describing test cases

Property	Function
id	Test case with unique identification
URL	Uniform resource locator of web service, and each tested API corresponds to one URL
method	Request method, including GET (sending a display request to a specified resource), POST (submitting data to a made resource to request for server processing, such as submitting forms or uploading files), PUT (uploading updates to a specified resource), DELETE (requesting the server to delete resources identified by Request url), HEAD (same as GET, but the server will not return the text of the resource), with URL to uniquely determine the tested API
form-data (optional)	Submit form information (used in POST or PUT request)
Header (optional)	Header, for example, Accept a response format specified acceptable, Content-Type specifies the MIME type of a request body (used in POST or PUT request). Some websites that detect User-Agent and referrer are required to complete this item to avoid being rejected by HTTP request.
basic authentication (optional)	Basic authentication, provide user name and password as identification while requesting
assertion	Assertion, perform multi-dimensional

978-1-5090-6296-6/17 $31.00 © 2017 IEEE
DOI 10.1109/BigDataSecurity.2017.35

	general verification on HTTP response
scripts	Scripts, perform custom validation on HTTP response

According to Table 1, we can observe that besides assertion and scripts, many properties of test cases are associated with HTTP, so each test case is essentially a package for one HTTP request, and the execution of each test case needs a correct complete HTTP request.

Research and analysis indicated that a general REST API test needs the following:

- Function tests: tests of version compatibility, buffering and connectivity, part Get request, condition Get request, HTTP status code, correctness of response content, correctness of response header, and custom validation of support.
- Non-functional tests: the demand for testing concurrent volume is not much. Simple performance tests need to be supported, and system bottleneck should be discovered by determining response time and response size. Regression testing is supported by automatically triggering function tests, adjusting the set of test cases, and adopting a certain notification mechanism. There are three trigger modes: timing trigger, manual trigger and code update trigger. The notification mechanism includes mails, short messages, messages in collaboration tools, etc.

Assertion is the core section of a test case. It is used for judging whether the output of a test method conforms to what is expected, and used for comparing and judging whether API meets requirements. The verification of assertion should meet multi-dimensional demands which have been mentioned above. The output of REST API test is HTTP response, and the verification of HTTP response involves several aspects in Table 2.

Table 2: Validation content

Category	Object	Meaning
Functional, focus on the correctness of REST API	Status code	3-digit code which indicates HTTP response status of web servers, the following are general status codes: 200 – request is successful, 302 - temporary redirect, 404 – requested URL cannot be found
	header	Information referring to server transmission response status, for example, Content-Type indicates the MIME type of a file
	Response entity format	Format of message entity, including XML, JSON or TEXT
	Response entity content	Content of message entity
Nonfunctional, focus on performance of REST API	Response time	From the request sent by user to server, to the time spent on downloading target content to user end
	Response size	Number of bytes returned in response

Each validation object in the table above corresponds to one validator, so REST API testing tools include six validators which are status code, response header, response entity format, response entity content, response time and response size. Each validation of validators is one assertion which consists of components in the following table:

Table 3: Assertion description

Name	Function
source	Source for obtaining data of validation can be obtained from HTTP response headers, JSON, XML or text entities; and also could be HTTP response status code, response time and response size.
property	Search property of data source. For HTTP header, it is the name of the HTTP header. For the response entity in JSON format, it needs to be achieved in a path indicated by standard Javascript tab. For response entity in XML format, it needs to be achieved in a path indicated by Xpath expression. If the data source is HTTP response status code, response time or response size, this section is empty.
comparison	Manner for comparing extracted data and target data. According to data source and property, true value in response can be obtained; whether it is empty can be determined; whether it is equal to target value, whether it includes substring of target value, whether it is a number, whether it includes some property (only applicable for JSON response format), etc. can be determined
argetValue	Predicted value compared with true value

One or more assertions can be defined in one test case. Each assertion can be considered as a testing step. When the test case is running, the test case cannot pass until all assertions are successful.

Assertions can meet most general validation requirements. However, assertions re largely restricted in complicated scenarios for property extraction and validation. For example, in some cases, it is difficult to determine the return format of one REST API is XML or JSON, so that validation cannot be made by assertions. Through custom scripts, firstly, judge the response format; then, select a method according to the obtained response format to search for property for validation. The relationship between any two of assertions is “and”, so more complicated validation logic can be defined in scripts, and an assertion can be randomly created.

In addition, scripts can access all requests and response data, including request headers, request parameters, response headers, response entities, etc. Some standard libraries will help provide more flexible and powerful functions [5]. For example, they can provide more space for testers with

programming fundamental when they are judging whether some property conforms to regular expression, encoding or decoding properties, processing date, performing arithmetical operation, etc.

Scripting language is generally weak and selects interpretive execution, which can simplify the processing of response, such as javascript, groovy. Scripts should be written in compliance with the syntax of the selected test language, if not, the test case will not pass the validation of validators. Note, scripting is executed prior to assertion.

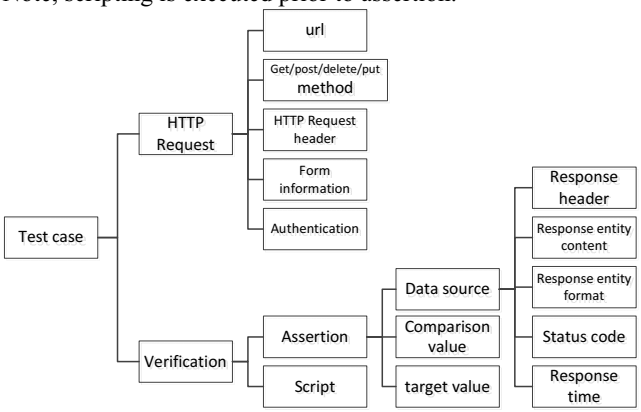

Figure 1: Description of testing case

The description of test case is introduced above. The test description language not only needs to meet demands described by the test case, but also needs to have high expansibility so as to be convenient to add more testing conditions in the future. The most important is to be easy to process by machines, so that it will be convenient to generate and execute test case. After making comparison, I decided to adopt JSON[6] which is lightweight and easy to read.

Figure 2 shows a test case described by test model language. The id of the test case is 123; it will send a post request to url (https://onboard.cn/account/signi); in the request header, Content-type is specified as application/x-www-form-urlencoded; the form parameters are "email"-"xuchen109@gmail.com" , "password"-"12345678"; http validation is not made. There are four assertions: the expected return code is 200; the value of Content-Type in the expected response header is text/html; the format of the expected response entity is text and includes the character string "ITEM"; the expected response time is shorter than 1000ms. The return code of inspection status of the test case is 200.

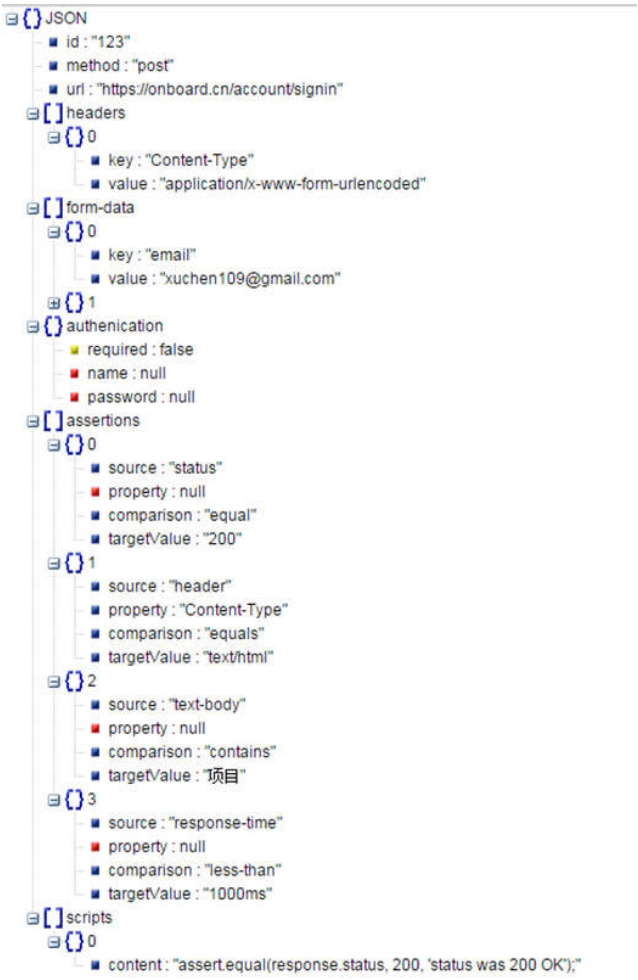

Figure 2: Examples of test language standards

After the test model language is defined, the case content of input can be tested by graphical interface to automatically generate test case, which solves the problem that there's no interface in REST API test and the problem that the readability of input and output is poor. In the meantime, the writing of a large amount of testing codes is omitted.

III. TEST CASE COMBINATION

A. Sequence of organizing and executing test case

Figure 1 shows single test cases. To comprehensively test Web service, improve test coverage and manage test cases, the test cases can be organized into a test suite with hierarchical structure in the test plan. A group of associated test cases are gathered to form a test suite. The test suite is a container for collecting one or more test cases and managing multiple test cases in groups. The test cases in each test suite will be executed in the sequence as stated in the test suite.

Test suites support nest. Multiple test suites can be combined and then added into another test suite, just like multiple test cases are added into one test suite. This manner is very suited for the development and integration of program module in reality. Finally, one root test suite corresponds to

one test plan. The form of each test plan is a multi-way tree. When REST API tools run, test cases will be executed as ordered by the preorder traversal of the tree.

B. Solve dependency between test cases

Test cases in one test plan cannot be completely independent, and they may have some dependency relationship[7]. There are two kinds of dependency relationship: data dependency and condition dependency. Several common scenarios of data dependency and condition dependency in REST API test are listed below:

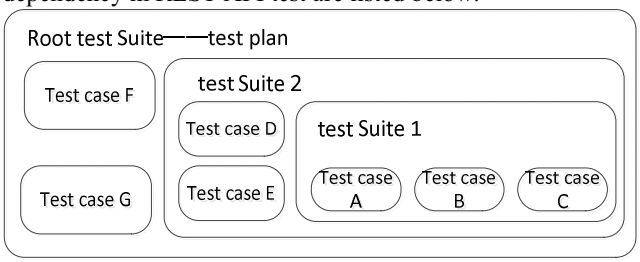

Figure3: Relationship of combined test cases

（1） Content dependency of response entity

For example, one Web service permits a user to manage to-do; a new to-do can be created by sending POST request to URI http://todohost.com/todos to submit the information referring to the to-do. The to-do list can be obtained by sending HTTP GET request to the same URI. After sending POST request, assume the id of the resource corresponding to the new added to-do is todo2, the to-do can be accessed through URI http://todohost.com/todos/todo2.

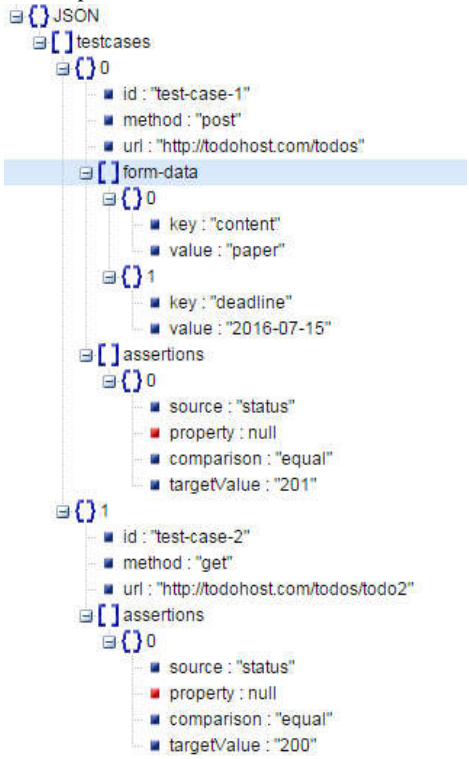

Figure 4: Data dependency of response entity

Figure 4 covers the language description of test suites of two test cases. The test-case-1 sends HTTP POST request to the specified URI; the form-data in the request includes the content, expiration date, etc. of the to-do; the HTTP response code returned by assertion is 201 (creation successful). The test-case-2 sends HTTP GET request to the specified URI, and the HTTP response code of assertion is 200.

We can see that the id of URI including the new created to-do in the test-case-2 is hard coded. If the id of the new resource is really todo2, the test plan will be passed. However, this pass is accidental. The id of to-do resources created will vary when the test-case-1 is executed at different time, so the next execution of test suite must be failed. In such case, the establishment and execution of test-case-2 will depend on the execution result of test-case-1, which needs to extract the id of to-do from the response entity of the request in test-case-1 to establish its own URL. Data transmission is required between both.

（2） Data dependency of response header

Http is a stateless protocol, so it cannot manage the status of request and response which occurred previously. Though the stateless protocol consumes fewer resources, it also brings the difficulty for servers in managing the status of clients because it cannot manage information like login status. To resolve the problem, Cookie technology[8] was introduced. Cookie information is written in request and response messages to control the status of client ends. In Web service referring to login and user conversation, the cookie information returned by the former test case needs to be added to the header of the subsequent test case, otherwise, the login or conversation will be invalid.

Figure 5: Data dependency of response header

Take *onboard* as example: one user reviews own team list after logging in. Figure 5 shows one test suite including two test cases. After the user logs in test-case-3 test, it will skip to “select team” page; the words “select team” can be found in the “assertion” page. The test-case-4 test selects team functions, and the HTTP status code of assertion is 200. However, if the cookie information of the response header returned by test-case-3 is not added in the http header of test-case-4, onboard will re-navigate the request to the login page, the status code is 302, and login failed.

（3） Condition dependency

Test case A only can decide whether to be executed based on the test result of test case B. In such case, test case B does not influence the establishment of test case A, but will influence the execution of test case A. Test case A should meet certain requirements: whether test case B is passed, whether the response entity of test case B meets specified conditions, whether the response header content of test case B meets specified conditions, etc. For example, for a service providing to-do management, test case B tests whether to-do (id is todo2) is completed, if not completed, test case A will be executed to complete to-do function; otherwise, test case A will not be executed. Condition dependency is a special data dependency.

To resolve the dependency relationship between test cases and solve share data between test cases, adopting thought of traditional programming languages, we consider the whole test plan as a program editing towards process, and consider the test case as a function, which is the basic unit of the whole test plan to be executed. The test suite is also considered as a function, which is the call of a function group.

A function has parameters and return value. Parameters indicate what the function depends on external environment; the return value indicates what influence the function makes on external environment. Similar, if one test case depends on data generated by previously executed test case, the data is input as parameters of the current test case. If the subsequent test case depends on the data generated by the current test case, the data generated by the current test case is input as return value and transmitted to the subsequent test case.

We expanded the previous test specification language, adding two parts:

1) Variable: use identification “var”, the form is key - - value. The key indicates the name of return value, and the value can be obtained from HTTP request header, JSON, XML or text entity; the retrieval mode is consistent with that of data in assertion. More flexible and powerful test scripts should be used for complicated retrieval variable. The life cycle of variable is from beginning of statement to the end of test plan, so only the test case executed after the current test case can use variable. Test cases are executed according to the sequence, so if two different test cases state variables with the same key name, the later variable will cover the front return.
2) Parameters: use identification “param”, and use the variable that is stated previously. The test case is used in the manner of “brace + parameter name”, for example, the URI of one test case is http://todohost.com/todos/{id}. When the test case is searching for variable, it should begin at the test suite with the current case, and search from “inside” to “outside”. If the internal test suite defines the variable having the same name of the external test suite, the variable of the internal test suite will shield the variable of the external test suite. If the root test suite is found, but the statement of variable is not found, the test case will not pass the test validator.

After being expanded, each test case corresponds to one function; “id” indicates the function name; “param” indicates the used variable assigned with value; “var” indicates to call the function and assign value to a variable; other parts form the function body. With such mechanism, the data are shared among test cases. The example below will show how to resolve data dependency through data transmission.

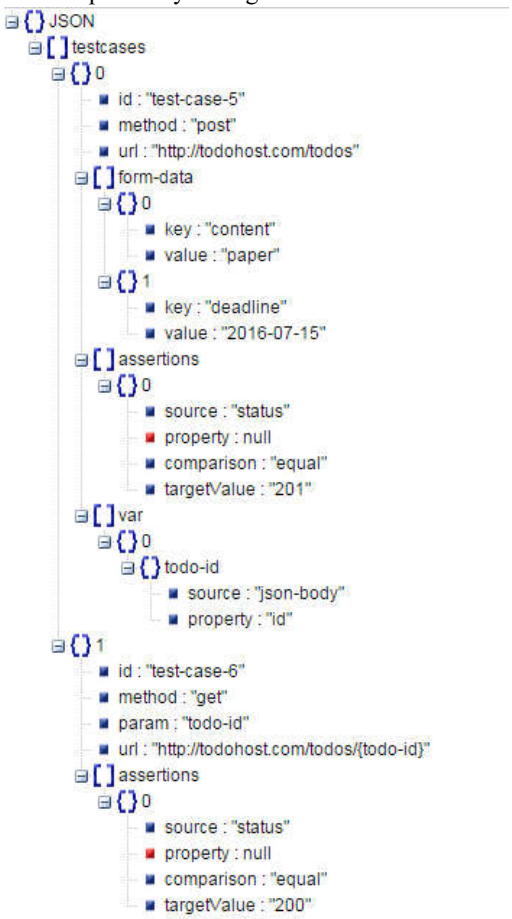

Figure 6: Solve data dependency through data transmission

Figure 6 shows a test suite including two test cases. Test-case-5 states variable from test case, the variable name is “todo-id”, and the value of variable is acquired from the JSON character string of the new to-do returned from HTTP

response. Test-case-6 and test-case-5 are in the same test suite, and test-case-6 will be executed after test-case-5. The request URL of test-case-6 is established by "todo-id" variable; during test execution, {todo-id} is replaced dynamically.

The problem regarding data dependency of response header will be resolved by the same way.

The above content shows that data are transmitted from one test case to the other through variable. In some cases, some public values, such as domain name and API instruction, are required in all test cases in the whole test plan. If these values are repeatedly used in each test case, the test cases may not be well maintained in the future. We consider the whole test plan as complete program programming towards process, so these values should state definition through initial variable to be present in the whole test plan where they can be accessed by all test cases in the test plan. The definition of test variable occurs before the first test case in the test plan is executed, and the initial variable is used in the same way of common variables.

Initial script can perform dynamic initialization on initial variable by some standard libraries, such as timestamp and UUID, which can be combined with initial variable to provide more powerful functions.

Data transmission didn't resolve the problem of condition dependency. Expand the test suite to add conditional judgment section. The conditional judgment section of the test suite is a ternary expression consisting of left operand, comparison operator and right operator. When defining conditions, operand could be a hard code or the variable of previous statement. The comparison operator is consistent with the comparison symbol in assertion. The test suite (including sub-test suites and test cases in the test suite) can be executed only when conditions are satisfied, otherwise will skip. The test suite which must be executed also has conditional judgment which always holds.

IV. CONCLUSION

Fully for REST web service, test model has the following features:

1) Support multi-dimension API validation by assertion and script.
2) Test cases support the suite of any level; test cases are executed in the manner of preorder traversal of the multi-way tree.
3) Test cases are processed in the manner when the program is in process; mapping the test tool model to program execution language helps to realize the data transmission among test cases and the conditional execution of test cases.

Propose test model description language which has good expressing capability and make it possible to automatically generate test cases through interface.

References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, Irvine - Irvine, CA 92697, USA, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=932295

[2] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Soap version 1.2," W3C recommendation, vol. 24, 2003.

[3] S. Azzam, M. N. Al-Kabi, and I. Alsmadi, "Web services testing challenges and approaches", in Proceedings of the 1st Taibah University International Conference on Computing and Information Technology (ICCIT 2012), 2012, pp. 291- 296.

[4] A. Bucchiarone, H. Melgratti, and F. Severoni, "Testing service composition," in Proc. 8th Argentine Symp. Softw. Eng., Mar del Plata, Argentina, 2007, pp. 1–16.

[5] Hanna, Milad, Nahla El-Haggar, and Mostafa Sami. "A Review of Scripting Techniques Used in Automated Software Testing." International Journal of Advanced Computer Science and Applications (IJACSA) 5.1 (2014).

[6] Crockford D. The application/json media type for javascript object notation (json)[J]. 2006.

[7] Chakrabarti, S. K., & Kumar, P. (2009, November). Test-the-rest: An approach to testing restful web-services. In Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World: (pp. 302-308). IEEE

[8] Nakajima S. Model-checking verification for the reliable web service. In: Proc. OOPSLA'02 Workshop on Web Services, 2002.