Towards Property-Based Testing of RESTful Web Services

Pablo Lamela Seijas
University of Kent
Canterbury, UK
P.Lamela-Seijas@kent.ac.uk

Huiqing Li
University of Kent
Canterbury, UK
H.Li@kent.ac.uk

Simon Thompson
University of Kent
Canterbury, UK
S.J.Thompson@kent.ac.uk

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools (e.g., data generators, coverage testing)

Keywords REST, FSM, QuickCheck, statem, Google Tasks, Storage Room, Erlang, model, generator, JSON

Web Services, and in particular REST Web Services [5], are expected to follow a series of conventions, and so it should be possible to create a framework that abstracts the parts of the testing model which are applicable to all such services. This way we would not have to develop tests from scratch for each project but, instead, it would be sufficient to adapt the framework to each set of particular requirements.

As a first step, we have developed a simple property-based test model in Quviq QuickCheck [7] giving an idealised model of a REST Web Service. We have adapted this to test two existing Web Services that broadly follow REST principles: Storage Room [15] and Google Tasks [6]. This has shown us how the systems deviate from pure REST, and how to adapt the model accordingly.

Quviq's QuickCheck provides some specific tools that simplify the generation of input data for certain scenarios. In the style of [3] we use the module `statem` that allows the interface under test to be described by a one state machine, where transitions represent calls to the interface with a mutable "data" state value, preconditions, and postconditions. We also show briefly how we can abstract the approach in order to use the module `fsm`, which uses a multi-state FSM, for the same purpose.

Lastres has also used QuickCheck `statem` to test a REST Web Service (Wriaki) [12]; his work is specific, while our work focusses building a general model. Property-based testing has been used to test SOAP (non-restful) Web Services in an automatic way using PropEr [11] and Haskell Quick-Check [16]; and FSMs [1] and EFSMs [8] have also been for testing non-RESTful Web Services.

The REST model

According to the REST architecture [5], each resource must have a permanent Uniform Resource Identifier (URI) that identifies a resource globally [2] and which must be unique and persistent. In conformance with the HTTP protocol, whenever we want to retrieve a resource we must use the method `GET`; when we want to add a new resource we must use the method `POST`; when we want to remove a resource we must use the method `DELETE`; and when we want to modify a resource we must use the method `PUT`.

No assumptions must be made about the location of resources since it should be possible to find them by following hyperlinks from a base URI [14]. With this assumption in mind we can model a collection of resources in a similar way to how we would model a database table: each entry of the collection would correspond to a row in the database table and `GET` would correspond to `SELECT`, `INSERT` would correspond to `POST`, `UPDATE` would correspond to `PUT`, and `DELETE` would correspond to `DELETE`.

Our collection will have a defined URI. For example (where we use '`...`' to elide part of the initial segment of the URIs):

```
http://restsrv...collections/book_collection
```

Based on the URI for the collection, there will be a URI where we can `POST` new entries and `GET` the list of existing entries. For example:

```
http://restsr...collections/book_collection/entries
```

And all the entries in the collection will follow a common pattern. For example:

```
http ... book_collection/entries/Cinderella
```

In our model we can then define five operations, four based on the HTTP actions, and a fifth to list an entire set of resources. We give these as Erlang functions and indicate how they map to their equivalents in HTTP. For example

```
get(Key) -> Entry
```

retrieves the entry with key `Key`: e.g.

```
get("Cinderella") ->
  " {
         ...
         title: 'Cinderella';
         author: 'Charles Perrault'
         ...
    } "
```

would be implemented as a `GET` call to the URL where the entry with key `Key` is stored:

```
GET .../book_collection/entries/Cinderella
```

and the other operations are similar.

In our implementation of the `statem` [7] model, we define the state as a record containing two data structures: One set with all the keys that have been returned by the `post/1` command, and one dictionary with the pairs of keys and resources that are stored and have not yet been deleted.

In our statem module we generate sequences using the four basic commands `get/1`, `put/2`, `post/1`, and `delete/1`. Each

Erlang '13, September 28, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2385-7/13/09.
http://dx.doi.org/10.1145/2505305.2505317

of these commands expect at most two parameters: a key and a resource. Keys are generated either randomly or, when possible, they are chosen from the set of all previous keys. Resources are always generated by the function `comment_gen/0` which is specific to each Web Service.

The "next state" functions update the resources in the dictionaries and add the new keys to the set as expected. No preconditions are needed since all the commands can be performed at any time. Postconditions check that errors are received whenever the keys are wrong, and that the resources returned by the command `get/1` at least contain the information stored under the same key in the appropriate dictionary of the state record.

Finally, in order to ensure that the model and the Web Service are always on the same page, we implemented `invariant/1` to call the method `list()` and check that the list of hashes returned matches the hashes we have stored in the state of our model at anytime. The code of the test model is given in the full paper [10].

Testing Real Web Services

Storage Room [15] is an engine that provides users with an online database with a RESTful API, with an administration website. Google Tasks [6] is a simple web application that allows users to keep several lists of tasks and subtasks, with due dates and the ability to mark tasks as done; again this provides a RESTful API (as well as the usual one).These are production systems, and so we were constrained in their use: we introduced a delay of half a second between requests and limited the number of test cases produced by QuickCheck to 30.

Testing unearthed no faults, but we did find a discrepancy between expected and actual behaviour: if in Storage Room we `POST` an object, then we `DELETE` it and finally we `GET` it, we still obtain the object back (but with a `trash` meta-attribute set). This 'delated' delete mimics the behaviour of 'trash' on the desktop, and indeed Klein and Namjoshi, in their formalisation of RESTful behaviour [9], also see this variation of the behaviour of `DELETE` as reasonable. We updated the test module to reflect this, and then the 30 test cases passed without a problem. These modified tests also worked with Google Tasks without any further changes.

Finite State Machine

As an alternative approach, we can abstract out some of the preconditions and postconditions by designing the model as a finite state machine with multiple states; in particular, we make the distinction between an empty and a non-empty store, and whether or not there are any elements marked as `trash`. We can implement this model using the `fsm` module from QuickCheck, implying that QuickCheck will try to ensure that the visits to each state are balanced so that the tests generated reach all the functionalities evenly.

Generators

Input to Web Services usually consists of XML or JSON structures. In the recent years JSON[4] has been gaining popularity; JSON is a format with similar functionality to XML, but JSON objects can also be interpreted as pure JavaScript. Its syntax basically consists of nested dictionaries (`{...}`), arrays (`[...]`), and the basic boolean, number, string, and null primitives . Both Storage Room and Google Tasks produce and expect JSON as resources.

In the early stages of this experiment, we used the same object as input for all the tests; in the final version, we used randomly generated objects to get better coverage of the target systems.

In order to make a QuickCheck generator out of a JSON structure, we had several options: we could use strings with pieces of JSON and use Erlang to put them together, we could use the Erlang representation of JSON and then encode the composition, we could use a similar approach to the one used on XML by Lampropoulos and Sagonas in [11], we could create a new language mixing both JSON and Erlang's symbolic representation, or similarly we could add annotations as template languages like PHP or JSP do with HTML. In our case it was enough to label attributes with tags as strings in the JSON itself. With these tags we can specify which kind of generator should be placed where, and which parameters could be omitted. We can then parse the JSON structures with libraries like `mochijson2` from MochiWeb project [13], and find and replace the tags programmatically.

For example, wherever we find the string "`bool()`" we would replace it with the `bool()` generator from QuickCheck. In the same way we can choose to remove or not elements that are tagged as "`optional()`", or insert generators for other types of data.

Conclusions and Future Work

This paper contributes a practical example using QuickCheck to test RESTful Web Services, applying the technique to two specific scenarios. The approach involves substantial manual work that could be automated in the future. It is also not very extensive, since we only tested single collections and we did not cover dependencies between different collections or between parameters inside the same entry. As we tested 'external' web services we had to reduce the number of tests cases; that could be relaxed 'in house'.

The obvious next step would be to automate this process further. However, there are also a number of other improvements that would be useful, such as including tools to focus on common bugs that are already well known: buffer overruns, problems with encoding, etc.

We acknowledge the support of the European Commission for PROWESS, www.prowess-project.eu/, grant no. 317820.

References

[1] A. A. Andrews et al. Testing web applications by modeling with FSMs. *Software & Systems Modeling*, 4(3), 2005.

[2] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): generic syntax, 1998.

[3] L. M. Castro and T. Arts. Testing Data Consistency of Data-Intensive Applications Using QuickCheck. *Electr. Notes Theor. Comput. Sci.*, 271:41–62, 2011.

[4] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON), 2006.

[5] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California, 2000.

[6] Google Tasks. Google Tasks. https://mail.google.com/tasks.

[7] J. Hughes. Quickcheck testing for fun and profit. In *PADL'07*. Springer, 2007.

[8] C. Keum et al. Generating test cases for web services using extended finite state machine. In *Testing of Comm. Systems*. Springer, 2006.

[9] U. Klein and K. S. Namjoshi. Formalization and Automated Verification of RESTful Behavior. In *Computer Aided Verification*, 2011.

[10] P. Lamela Seijas, H. Li, and S. Thompson. Towards Property-Based Testing of RESTful Web Services. Tech. report, http://kar.kent.ac.uk/34865/, 2013.

[11] L. Lampropoulos and K. Sagonas. Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services. *arXiv:1210.6110*, 2012.

[12] R. Lastres Guerrero. Testing a distributed Wiki web application with QuickCheck. 2012.

[13] MochiWeb. MochiWeb. https://github.com/mochi/mochiweb.

[14] P. Prescod. REST and the Real World. *Published on XML.com*, 2002.

[15] Storage Room. Storage Room. http://storageroomapp.com.

[16] Y. Zhang et al. Automatic testing of web services in Haskell platform. *Journal of Computational Information Systems*, 6(9), 2010.