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

ABSTRACT

Developing APIs as Web Services over HTTP implies adding an extra layer to software, compared to the ones that we would need to develop an API distributed as, for example, a library.

This additional layer must be included in testing too, but this implies that the software under test has an additional complexity due both to the need to use an intermediate protocol in tests and to the need to test compliance with the constraints imposed by that protocol: in this case the constraints defined by the REST architectural style.

On the other hand, these requirements are common to all the Web Services, and because of that, we should be able to abstract this aspect of the testing model so that we can reuse it in testing any Web Service.

In this paper, as a first step towards automating the testing of Web Services over HTTP, we describe a practical mechanism and model for testing RESTful Web Services without side effects and give an example of how we successfully adapted that mechanism to test two different existing Web Services: Storage Room by Thriventures and Google Tasks by Google. For this task we have used Erlang together with state machine models in the property-based testing tool Quviq QuickCheck, implemented using the `statem` module.

1. INTRODUCTION

In order for software to be testable, it needs to have clearly defined interfaces. When testing Web Services we get two advantages inherently:

- Web Services have a clear interface by definition.
- The interfaces for Web Services are language-independent. Because of this we do not need to know in which language they are written, under which operating system they are running, or where in the world they are deployed.

But because we have a protocol in the middle (usually HTTP), the number and complexity of tests needed are higher than would be in the case that we just had to call the functions of a traditional library because:

- We must test the constraints of the protocol supporting access to the services; in our case this is the REST architecture implemented over HTTP.
- We have to use sockets or other intermediate libraries which add complexity to the tests.
- Web Services are usually exposed to a more hostile environment – the internet – so they need to be tested for safety against malformed or maliciously constructed requests.

Nevertheless, since Web Services – and in particular REST Web Services [8] – usually follow a series of conventions, it should be possible to create a framework that abstracts the parts of the testing model which are applicable to all Web 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 and in an attempt to find those commonalities, we have developed a simple property-based test model that represents an idealised model of a REST Web Service (Section 3), and we have adapted this model to test two existing Web Services claimed to follow the REST principles: Storage Room [22] and Google Tasks [9]. (Section 4).

Despite the fact that our approach has only been applied on a small scale, and that we did not find any actual bugs, the exercise has allowed us to learn behaviours of the target systems that were different to those that we were expecting beforehand, thus giving us a broader understanding of their functionality. Furthermore, we achieved this in a non-intrusive way, by issuing a total of less than 5000 requests to each of the services and spreading them out in time so that they would not perceptibly diminish the availability of the Web Services. The contributions of this paper are:

- A base model to test side-effect-free REST Web Services.

- A technique to adapt the model to specific scenarios.
- Two examples that illustrate the model and technique applied to two existing Web Services.

2. RELATED WORK

The Web Services tested in this paper return and expect resources in JavaScript Object Notation (JSON) [11]. JSON is becoming popular and is often used where XML was used before, or in conjunction with XML. JSON is a lightweight format used for data exchange that is characterised by being a subset of the JavaScript language. This property makes it easy for web browsers to parse. In Section 6 we will talk a bit more about JSON and we will address the issue of randomly generating valid JSON input data for the use in testing.

QuickCheck was originally a tool created for Haskell [6] that uses random generated data as input for testing functions. Functions are fed with random inputs and the tool checks that the outputs comply with the appropriate post-conditions. Because at the time of initial development Haskell had a small industrial community, a new version of QuickCheck was developed for Erlang and it is currently being maintained by Quviq AB [10]. In turn, the latter QuickCheck inspired the creation of a similar but open-source tool called PropEr, which further explores the integration with Erlang type specifications [18]. Versions of QuickCheck have also been developed for other languages, notably Java [20].

Quviq's QuickCheck provides some specific tools that simplify the generation of input data for certain scenarios. It is the case of the modules `statem` and `fsm`. The module `fsm` allows the interface under test to be described by a finite state machine, where transitions represent calls to the interface. QuickCheck can use this descriptions to generate sequences of commands that comply with the given state machine. The module `statem` works in a similar way but it does not require a state machine to be defined, relying instead on a single "control state" with a mutable "data" state value, preconditions, and postconditions.

The work described in this paper is the result of using a QuickCheck `statem` – in the style described by Castro and Arts to test database intensive applications [3] – to the particular case of REST Web Services. We also show briefly how we can abstract the approach in order to use the module `fsm` for the same purpose, (Section 5).

In the past, other approaches have been used to test REST Web Services, like is the case of the tool Test-the-rest [4]. These approaches usually rely on unit testing in contrast to ours, which consists in property-based testing. There is also some work on testing and verification of the RESTfulness of Web Services that focuses on the general REST principles such as connectedness [5] and temporal constraints [13].

On the other hand, property-based testing has been used to test SOAP (non-restful) Web Services in an automatic way before. In [14] Lampropoulos and Sagonas show a way to create properties to test SOAP services by using WSDL descriptors. Their method generates a set of simple properties without the need of human intervention, and then they use this properties as a template for writing more complex ones. Nevertheless, despite the fact that WSDL descriptors are a *de facto* standard for SOAP Web Services, they are rarely present in REST Web Services. For this reason we use a different approach for the elaboration of generators.

Haskell QuickCheck has also been used for SOAP Web Service testing [23], and automatic approaches for testing non-restful Web Services using both FSMs [1] and EFSMs [12] have been described in several places.

Finally, Lastres in [15] has also used QuickCheck `statem` to test a REST Web Service (Wriaki) in a property-based way. The method we use in this paper is quite similar to this approach although here we focus on generalising the common aspects of REST Web Services.

3. THE REST MODEL

The principles of the REST architecture [8] reflect the original principles behind the HTTP protocol. Because REST is a radical new perspective, which has been highly influential, the terms 'REST' and 'RESTful' have become over-used buzzwords, and despite that fact that the original principles identified by Fielding [8] are clear and unambiguous, there is controversy about what is and isn't a RESTful system in practice; see Fielding's blog post *REST APIs must be hypertext-driven* [21] for a particular example of controversy.

Introducing the model

According to the REST philosophy, 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 [19]. 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
http ... book_collection/entries/Hansel_and_Gretel
http ... book_collection/entries/Snow_White
...
```

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.

Note that `list()` uses the same HTTP action as `get()` , namely `GET`. The reason we do not use the same method for both is that `list()` is used for resources of the type "collection", and `get()` is used for the rest of resources. `list()` will extract the list of entries in the collection while `get()` will return the resource as a black box. The facade must be adapted for each particular implementation and by separating both behaviours we can keep a more general `get()` while `list()` is more specific.

The functions of the model

We now show the proposed Erlang functions of the model and how they would map with their equivalents in HTTP. They are presented in the form:

```
function(Parameter1, ..., ParameterN) -> Result
```

In this example, the titles of the books are used as keys, but in real systems keys would usually be just hashes. The values are strings containing the JSON representation of the information of the book but they could also be any string:

`get(Key) -> Entry` - Retrieves the entry with key `Key`. For example:

```
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
```

`delete(Key) -> ok` - Removes the entry with key `Key`. For example:

```
delete("Cinderella") -> ok
```

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

```
DELETE .../book_collection/entries/Cinderella
```

`post(Entry) -> Key` - Adds the entry `Entry` to the collection, and returns the `Key` used to store it. For example:

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

would be implemented as a `POST` call to the URL of the entries of the collection:

```
POST .../book_collection/entries
{
  ...
  title: 'Cinderella';
  author: 'Charles Perrault'
  ...
}
```

`put(Key, PartialEntry) -> ok` - Modifies the entry with key `Key` to include the fields in `PartialEntry`. For example:

```
put("Cinderella",
    " {
        author: 'Brothers Grimm'
      } ") -> ok
```

It is implemented as a `PUT` call to the URL where the entry with key `Key` is stored:

```
PUT .../book_collection/entries/Cinderella
{
  author: 'Brothers Grimm'
}
```

`list() -> [{Key, Value}]` - Returns a list with all the pairs of keys and entries in the collection, (in our implementation, a list of tuples). For example:

```
list() ->
  [ {"Cinderella", "{ title: ... }"},
    {"Hansel_and_Gretel", "{ title: ... }"},
      ...
    {"Snow_White", "{ title: ... }"} ]
```

It is implemented as a `GET` call to the URL of the entries of the collection:

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

Using the model

In order to guide the design of our initial test model before trying it in the real services, we implemented the previous model as an Erlang `gen_server` with a dictionary (module `dict`) as storage. In this implementation, the `post` method returns a long hexadecimal hash, and the entries were assumed to be arbitrary Erlang terms. Whenever an entry is deleted, it is no longer accessible via `get`, `list` or `put`. Also, whenever any method is used with a `Key` that is not in the dictionary, the tuple `{error, not_found}` is returned. This is illustrated in the example execution below:

```
1> model:start_link().
{ok,<0.33.0>}
2> HashCinderella = model:post("Cinderella").
"fe3d73ebe0045732f200d90b"
3> model:get(HashCinderella).
{ok,"Cinderella"}
4> model:list().
[{"fe3d73ebe0045732f200d90b", "Cinderella"}]
5> model:delete(HashCinderella).
ok
6> model:delete(HashCinderella).
{error,not_found}
7> model:get(HashCinderella).
{error,not_found}
8> model:list().
[]
9> model:stop().
ok
10> q().
ok
```

In order to test our model we used QuickCheck `statem` as described by Laura Castro and Thomas Arts in [3], but this time, we used the new grouped version of `statem` instead of the ungrouped one.

The ungrouped version uses the same function for each aspect of the test model, because of that, all the preconditions, all the postconditions, etc. have to be together in the code. By contrast, in the new grouped version, we can put together all the functions corresponding to the same method, for example, the precondition of the `GET` method can go together with the postcondition of the `GET` method, which makes the test model more readable.

The code of the test model module is given in Appendix A. The function `model:comment_gen()` returns a QuickCheck generator of a valid entry, (details are explained in Section 6). The user-defined type `listset`, is equivalent to a normal `set` but also provides a function to access its elements by index, because of this, by using `listsets`, it is much easier and efficient to retrieve a random element, (the function `modelutil:rnd_from_listset_gen/1`, returns a generator that produces a list with a random element of the specified `listset`).

`invariant/1` is a function that is called before and after any command is executed. Our invariant function calls the method `list()` and checks that the hashes of the resources match the hashes we have stored in our model. This way we try to ensure that the model and the Web Service are always on the same page.

4. TESTING REAL WEB SERVICES

Storage Room [22] is an engine that provides you with an online database. It has an administration web site that allows you to define the structure of the database and it provides you with a RESTful API with the typical queries that databases have.

Google Tasks [9] is a simple web application that allows you to keep several lists of tasks and subtasks, it integrates with GMail and Google Calendar, and it allows you to set due dates and to mark tasks as done. It provides a RESTful API with roughly the same functionality than the normal web version.

Both Google Tasks and Storage Room were services in production phase at the time of the experiment. In addition, the amount of requests we can make to the Web Services are limited in both cases. Because of this, we introduced a delay of half a second between requests and limited the amount of test cases produced by QuickCheck to 30.

A typical API call to insert an object into Storage Room would look similar to the one shown in Figure 1.

We can easily apply the test model of our model to Storage Room by making a facade as shown in Figure 2. But when running QuickCheck we get back the error below.

For purposes of clarity we have replaced the actual input data with the literal `ENTRY_1`.

```
Shrinking..........(11 times)
[{set,{var,5},{call,dbtest,post,
              [{struct, [ENTRY_1]}]}},
 {set,{var,7},{call,dbtest,delete,[{var,5}]}},
 {set,{var,11},{call,dbtest,get,[{var,5}]}}]

dbtest:post({struct, [ENTRY_1]}) ->
    "5190fa9b0f66027a4900046e"
dbtest:delete("5190fa9b0f66027a4900046e") -> ok
dbtest:get("5190fa9b0f66027a4900046e") ->
    {ok, {struct, [ENTRY_1]}}

Reason: {postcondition, false}
```

Indeed, if in Storage Room we `POST` an object, then we `DELETE` it and finally we `GET` it, we still obtain the object back.

The behaviour discovered goes against our first intuitions about the behaviour of REST Web Services, and it could be seen as a bug. But having set this behaviour on purpose would make sense in some cases. This result could as well have been caused by an unknown intermediate server acting as a cache between the server and the client: we checked this by appending a random and innocuous extra parameter to the end of the URL and it was not the case, the server was still returning the object.

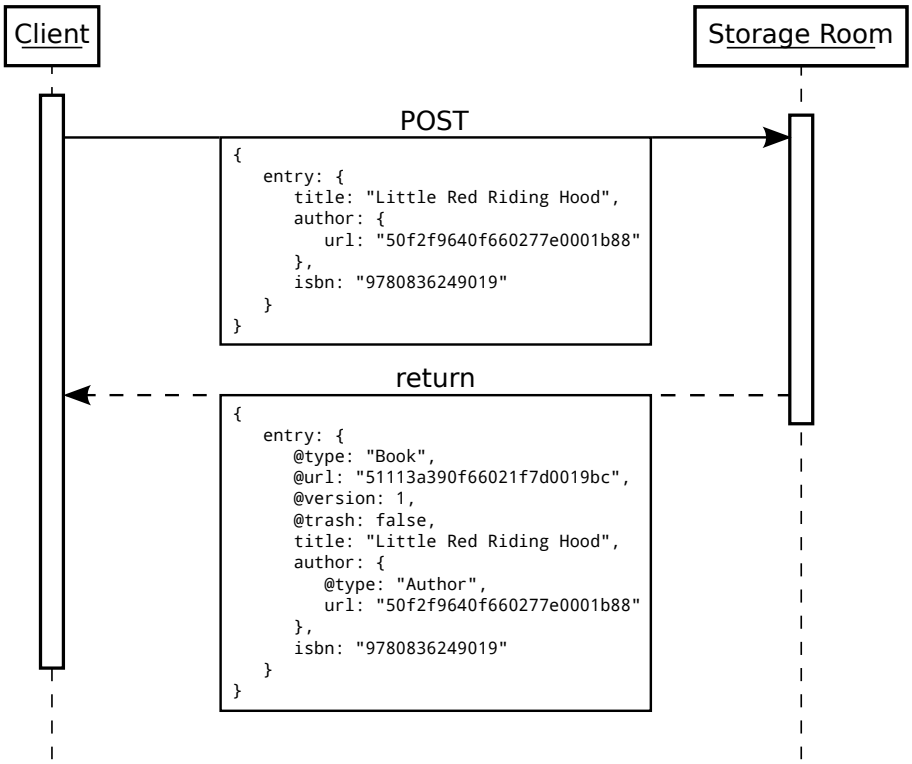

Figure 1: Representation of a normal POST call to Storage Room

The set of principles on which the web relies also says that queries do not necessarily have to be answered in the order in which they are made. It could be the case that we issued a `GET` query before a `DELETE` query but the server sees the `DELETE` before; and we may not want the `GET` query to fail because of that. This reasons could justify the behaviour.

Moreover, if disk space is not a problem, for data security reasons it may not be desirable to allow users to really delete data. Also, the usual intuition promoted by most GUIs is that when you delete something it goes to the recycle bin.

Klein and Namjoshi, in their formalization of RESTful behavior [13], also contemplate this delay in the execution of the DELETE communication as a reasonable variation on RESTful HTTP properties.

Storage Room has in fact a meta-attribute called `trash`, which tells us whether the object has already been deleted or not. In addition, we were able to check that the parent container object was also updated and it did not return any of the hashes corresponding to the children that were already deleted. Later we found the same behaviour in Google Tasks, the latter has an attribute called `deleted`.

We updated the test module to reflect the behaviour discovered in last section, (see Appendix B). And this time the 30 test cases passed without problems. The fixed tests worked with Google Tasks without any change to the `dbtest` module.

5. 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, see Figure 3.

We can implement a test model for this model by using the `fsm` module from QuickCheck which has some advantages over `statem`. For example, if we use the `fsm` module, then QuickCheck will try to ensure that the visits to each state are balanced so that the random tests generated reach all the functionalities evenly.

In the FSM model, we consider the following states, where 'normal' entries are those that are not 'trashed'.

- `EC` - Empty & Canonical - No entries are stored, (nor trashed, nor normal).
- `NC` - Non-empty & Canonical - Entries are stored, but none is trashed.
- `NN` - Non-empty & Non-canonical - There are both trashed and normal entries stored.
- `EN` - Empty & Non-canonical - There are entries but they are all trashed.

The four original REST methods were divided according to their possible different behaviours and an extra one was added:

- `POST` - Creates a new entry.

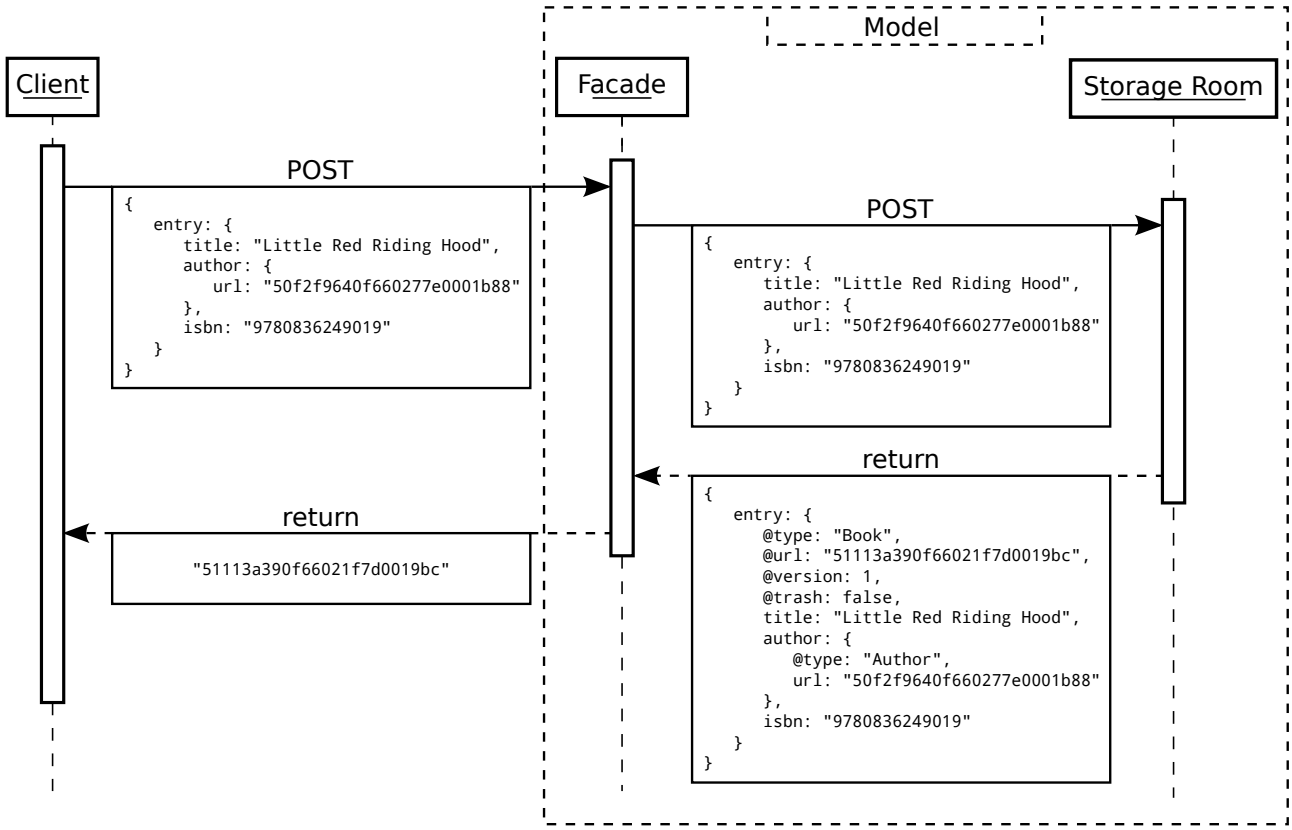

Figure 2: Representation of model and facade over Storage Room

  - `post_first` - Creates the first normal entry.
  - `post_norm` - Creates an additional normal entry.

- `GET` - Retrieves an existing entry.
  - `get_norm` - Retrieves a normal entry.
  - `get_del` - Retrieves a trashed entry.

- `DELETE` - Trashes an existing entry.
  - `del_last` - Trashes the last remaining normal entry.
  - `del_norm` - Trashes one of several remaining normal entries.
  - `del_del` - Tries to trash an already trashed entry, (does nothing).

- `PUT` - Modifies an existing entry.
  - `put_del` - Modifies a trashed entry.
  - `put_norm` - Modifies a normal entry.

- `empty_trash` - Removes definitely all the trashed entries. This command is not an API method neither in Storage Room neither in Google Tasks, but we decided to add it to our model for completion.

6. GENERATORS

Input to Web Services consists in most of the cases of XML or JSON structures. In recent years JSON has been gaining popularity and it seems to be slowly replacing XML in a growing number of scenarios. JSON is a format with similar functionality to XML, but JSON objects can also be interpreted as pure JavaScript. This property makes it very easy and efficient to parse by web browsers. Its syntax basically consists of nested dictionaries (represented with curly brackets), arrays (represented with square brackets), and the basic primitives of type boolean, number, string, and null [7].

During the early stages of this experiment, we used the same object as input for all the tests, which made things easier; but in the final version, we used several randomly generated objects to get a better coverage of the target systems.

In order to achieve this, we made generic generators out of our example objects. To illustrate this, we will use a more complex type of object than the books: `CommentOnABook`. Our original example object was similar to the one shown in Figure 4.

In order to make a QuickCheck generator out of a JSON structure we could use strings with pieces of JSON and use Erlang to put them together. We could also use the Erlang representation of JSON and then encode the composition. But perhaps the most readable way to mix QuickCheck generators and JSON is to add annotations as template languages like PHP or JSP do with HTML, so we decided to

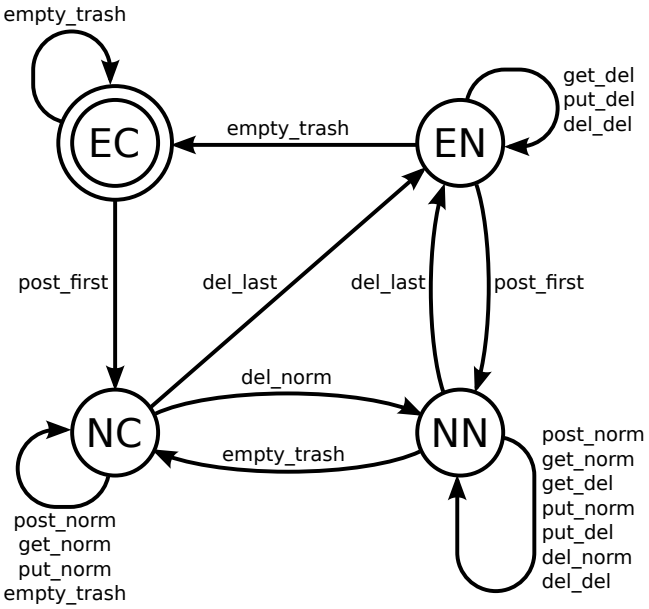

Figure 3: Representation of model and facade over Storage Room

label attributes with tags in the JSON itself. These tags specify which kind of generator should be placed where, and which parameters could be omitted as shown in Figure 5. We have also considered the possibility of creating a hybrid language combining JSON and Erlang or Erlang's symbolic representation, but creating a new language is always a delicate task and by using our simple solution we could reuse existing parsing libraries, which was enough for our approach. We leave the creation of an advanced JSON template language for future work.

Once we have defined the JSON template, we can programmatically parse it and replace its tags with QuickCheck generators. To parse the JSON structures we used `mochijson2` from MochiWeb project [17]. For example, we replaced `nonempty_string()` with the following generator:

```
nonempty_string_gen() ->
    ?LET(String,
         [choose(33, 126)|list(choose(32, 126))],
         list_to_binary(String));
```

This way we only generate visible standard ASCII characters. Some manual tests showed that Storage Room is implemented to consider that empty strings do not fulfil the mandatory parameter constraint. That is also the reason why we add a single character at the beginning, and why we do not allow the "white space" (ASCII code 32) for that character. For `bool()` we just used the `bool()` generator provided by QuickCheck.

Alternatively, in their paper about automatic WSDL-guided testing [14], Lampropoulos and Sagonas present a quite convenient and flexible method to build custom generators for XML. A similar approach could be used for JSON and it would probably be more appropriate for the ultimate goal

```
{
 "entry": {
   "name": "Stupid comment",
   "rating": 3,
   "was_read": false,
   "finished_reading": "2013-01-30",
   "time_started_page_37": "2013-02-01T10:35:00Z",
    "where_read": {
      "lat": -82.40859297752702,
      "lng": 65.6540299999998
   },
   "ideas": [
     "cold",
     "snow",
     "freezing",
     "ice",
     "wind"
   ],
   "comment": "I don't like the book at all.",
   "book": {
     "url": "http: ... /51113a390f66021f7d0019bc"
   }
 }
}
```

Figure 4: Example of `CommentOnABook` JSON

```
{
  "entry": {
    "name": "nonempty_string()",
    "rating": 3,
    "was_read": "bool()",
    "finished_reading": {
      "optional()": "2013-02-08"
    },
    "time_started_page_37": {
      "optional()": "2013-02-01T10:35:00Z"
    },
    "where_read": {
      "optional()": {
        "lat": -82.40859297752702,
        "lng": 65.6540299999998
      }
    },
    "ideas": [
      "nonempty_string()",
      "nonempty_string()",
      { "optional()": "nonempty_string()" },
      { "optional()": "nonempty_string()" },
      { "optional()": "nonempty_string()" }
    ],
    "comment": "nonempty_string()",
    "book": {
      "url": "http: ... /51113a390f66021f7d0019bc"
    }
  }
}
```

Figure 5: Example of tagged JSON structure

```
{
  "name": "nonempty_string()",
  "rating": 3,
  "was_read": "bool()",
  "optional()": {
    "finished_reading": "2013-02-08"
  }
  "optional()": {
    "time_started_page_37": "2013-02-01T10:35:00Z"
  }
}
```

Figure 6: Example of incorrectly tagged JSON

of developing a framework to automate the testing of REST Web Services, since it fits a broader range of scenarios. Nevertheless, for this scenario, our simpler *ad hoc* approach was enough.

Finally, lists or dictionaries that have parameters with the `optional()` tag, are replaced by generators that randomly remove some or all of the parameters tagged as `optional()`, (or none of them). Note that, in the case of dictionaries, parameters only have their value tagged. If we tagged both key and value we could have several parameters with the same key. See for example the incorrect JSON structure shown in Figure 6.

This would not make sense in JSON because dictionaries are made to be accessed by key.

7. KNOWN LIMITATIONS

The current approach still involves a lot of manual work that could probably 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. Dependencies should not be a problem for any particular project, but they are a more complex subject when speaking about a generalisation for any REST Web Service.

The self-imposed limitation to 30 tests was necessary in this scenario, due to the use of foreign Web Services. This amount of tests is much smaller than the 100 that QuickCheck uses by default. Even with such a small number of tests, we where able to use QuickCheck with our technique to find out that our preconceived initial model did not match the systems under test. Nevertheless, for cases where the tester has access to an appropriate testing environment, this number can be increased just by changing a constant in the test model.

8. FUTURE WORK

The obvious next step would be to automate this process further. However, there are also a number of lateral improvements that would be useful, as well as those mentioned in the previous section. For example, a common framework could include tools to focus on common bugs that are already well known: buffer overruns, problems with encoding, problems with character escaping, pages without authentication mechanism, code injection.

CRUD (Create, Read, Update and Delete) [16] is the acronym for database-like behaviour. REST is not CRUD. The `POST` method may have side effects or may not create a resource with an URL of its own. Future work should be focused in automating the modelling of this side effects. Nevertheless, most of the examples of REST Web Services that we have found in fact exhibit substantial CRUD-like behaviour. Because of this we envision a broader automated framework that may instantiate versions of this model adapted in some way to test different CRUD parts of Web Services and will fill the gaps by linking the CRUD parts and extending them by using a high-level model representing the side effects that go beyond the CRUD model.

9. CONCLUSION

In this paper we have contributed with a practical example of the application of QuickCheck to REST Web Services without side effects, with a technique to adapt this model to specific scenarios, and with an example of how we used the model to learn about the real implementation of the two Web Services we tested which differed from our preconceived mental model of them. Hopefully this approach will be useful as a first step towards the automation of testing of REST web services.

We are grateful to the European Commission for their support for of work via the collaborative project PROWESS, `http://www.prowess-project.eu/`, grant number 317820. We also wish to thank the anonymous reviewers of this work for the Erlang Workshop 2013 for their detailed and thought-provoking comments.

10. REFERENCES

[1] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with FSMs. *Software & Systems Modeling*, 4(3):326–345, 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] S. K. Chakrabarti and P. Kumar. Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pages 302–308. IEEE, 2009.

[5] S. K. Chakrabarti and R. Rodriquez. Connectedness testing of restful web-services. In *Proceedings of the 3rd India software engineering conference*, pages 143–152. ACM, 2010.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

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

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

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

[10] J. Hughes. Quickcheck testing for fun and profit. In *Practical Aspects of Declarative Languages*, pages

1–32. Springer, 2007.
[11] JSON, the JavaScript Object Notation. http://www.json.org.
[12] C. Keum, S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi. Generating test cases for web services using extended finite state machine. In *Testing of Communicating Systems*, pages 103–117. Springer, 2006.
[13] U. Klein and K. S. Namjoshi. Formalization and Automated Verification of RESTful Behavior. In *Computer Aided Verification*, pages 541–556. Springer, 2011.
[14] L. Lampropoulos and K. Sagonas. Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services. *arXiv preprint arXiv:1210.6110*, 2012.
[15] R. Lastres Guerrero. Testing a distributed Wiki web application with QuickCheck. 2012.
[16] J. Martin. *Managing the Data-base Environment*. Prentice-Hall, 1983.
[17] MochiWeb. https://github.com/mochi/mochiweb.
[18] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM, 2011.
[19] P. Prescod. REST and the Real World. *Published on XML.com*, 2002.
[20] QuickCheck for Java. http://java.net/projects/quickcheck.
[21] REST APIs must be hypertext-driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.
[22] Storage Room. http://storageroomapp.com.
[23] Y. Zhang, W. Fu, and J. Qian. Automatic testing of web services in haskell platform. *Journal of Computational Information Systems*, 6(9):2859–2867, 2010.

APPENDIX

A. TEST MODEL

```
-module(dbtest).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

-compile(export_all).

-record(state, {keys = listset:new(),
                dict = dict:new()}).

initial_state() -> #state{}.
invariant(S) ->
    Local = element(1, lists:unzip(
                         dict:to_list(S#state.dict)
                        )),
    Remote = element(1, lists:unzip(list())),
    lists:sort(Local) =:= lists:sort(Remote).

get_command(State) ->
    {call, ?MODULE, get,
     [oneof([modelutil:hash_gen()|
             modelutil:rnd_from_listset_gen(
               State#state.keys)])]
    }.
get_pre(_State) ->  true.
get_pre(_State, _Args) -> true.
get_next(State, _Value, _Args) -> State.
get_post(State, [Key], Res) ->
    is_substruct(
      case dict:is_key(Key, State#state.dict) of
          true -> {ok, dict:fetch(Key,
                                  State#state.dict)
                   };
          false -> {error, not_found}
      end, Res).

post_command(_State) ->
    {call, ?MODULE, post, [comment_gen()]}.
post_pre(_State) -> true.
post_pre(_State, _Args) -> true.
post_next(#state{keys = Keys, dict = Dict} = State,
          Var, [Value]) ->
    State#state{
      keys = listset:insert(Var, Keys),
      dict = dict:store(Var, Value, Dict)
     }.
post_post(_State, _Args, Res) when is_list(Res) ->
    true.

put_command(State) ->
    {call, ?MODULE, put,
     [oneof([modelutil:hash_gen()|
             modelutil:rnd_from_listset_gen(
               State#state.keys)]),
      comment_gen()]}.
put_pre(_State) -> true.
put_pre(_State, _Args) -> true.
put_next(#state{dict = Dict} = State, _Var,
         [Key, Value]) ->
    case dict:is_key(Key, Dict) of
        true -> State#state{
                  dict = dict:store(Key, Value,
                                    Dict)
                 };
        false -> State
    end.
put_post(State, [Key, _], Res) ->
    is_substruct(
      case dict:is_key(Key, State#state.dict) of
          true -> ok;
          false -> {error, not_found}
      end, Res).

delete_command(State) ->
    {call, ?MODULE, delete,
     [oneof([modelutil:hash_gen()|
             modelutil:rnd_from_listset_gen(
               State#state.keys)])]
    }.

delete_pre(_State) -> true.
delete_pre(_State, _Args) -> true.

delete_next(#state{dict = Dict} = State, _Var,
            [Key]) ->
```

```
    State#state{
      dict = dict:erase(Key, Dict)
     }.
delete_post(#state{dict = Dict}, [Key], Res) ->
    case dict:is_key(Key, Dict) of
        true -> ok;
        false -> {error, not_found}
    end =:= Res.

weight(_S, get) -> 2;
weight(_S, post) -> 1;
weight(_S, put) -> 1;
weight(_S, delete) -> 1;
weight(_S, _Cmd) -> 1.

prop_db() ->
    ?FORALL(Cmds, commands(?MODULE),
            begin
                {H, S, Res} = run_commands(?MODULE,
                                           Cmds),
                clean_up(S#state.dict),
                pretty_commands(?MODULE, Cmds,
                                {H, S, Res},
                                Res == ok)
            end).

%%% Returns true if the Left element is a
%%% substructure of the Right element
is_substruct(SubTuple, OriTuple)
  when is_tuple(SubTuple), is_tuple(OriTuple) ->
    SubList = tuple_to_list(SubTuple),
    OriList = tuple_to_list(OriTuple),
    case {length(SubList), length(OriList)} of
        {Same, Same} ->
            BL = [is_substruct(Sub,Ori) ||
                     {Sub, Ori}
                         <- lists:zip(SubList,
                                      OriList)]
                and_colapse();
        {_, _} -> false
    end;
is_substruct([], []) -> true;
is_substruct(List, []) when is_list(List) -> false;
is_substruct(List, [Head|Tail])
  when is_list(List) ->
    is_substruct(
      sub_if_substruct(List, Head), Tail);
is_substruct(Original, Original) -> true;
is_substruct(_, _) -> false.
and_colapse([true|Tail]) -> and_colapse(Tail);
and_colapse([false|_]) -> false;
and_colapse([]) -> true.
sub_if_substruct([], _) -> [];
sub_if_substruct([Head|Tail], Element) ->
    case (is_substruct(Head, Element)) of
        true -> Tail;
        false -> [Head|sub_if_substruct(Tail,
                                        Element)]
    end.

clean_up(Keys) ->
    clean_up_aux(
      element(1, lists:unzip(dict:to_list(Keys)))).

clean_up_aux([]) -> ok;
clean_up_aux([H|T]) -> ok = delete(eval(H)),
                       clean_up_aux(T),
                       [] = list().

%% Interface selection
%% ===================
check_prop() ->
    model:start_link(),
    eqc:quickcheck(eqc:numtests(1000, prop_db())),
    model:stop().

%% Commands
get(Key) -> model:get(Key).
post(Value) -> model:post(Value).
put(Key, Value) -> model:put(Key, Value).
delete(Key) -> model:delete(Key).
list() -> model:list().
comment_gen() -> model:comment_gen().
```

B. FIXES TO THE MODEL

```
-module(dbtest).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

-compile(export_all).

-record(state, {keys = listset:new(),
                dict = dict:new(),
                nodeldict = dict:new()}).


initial_state() -> #state{}.
invariant(S) ->
    Local = element(1, lists:unzip(
                         dict:to_list(S#state.dict)
                        )),
    Remote = element(1, lists:unzip(list())),
    lists:sort(Local) =:= lists:sort(Remote).

get_command(State) ->
    {call, ?MODULE, get,
     [oneof([modelutil:hash_gen()|
             modelutil:rnd_from_listset_gen(
               State#state.keys)])]
    }.
get_pre(_State) ->  true.
get_pre(_State, _Args) -> true.
get_next(State, _Value, _Args) -> State.

get_post(State, [Key], Res) ->
    is_substruct(
      case dict:is_key(Key,
                       State#state.nodeldict) of
          true ->
              {ok, dict:fetch(Key,
                              State#state.nodeldict
                             )};
          false -> {error, not_found}
      end, Res).
```

```
post_command(_State) ->
    {call, ?MODULE, post, [comment_gen()]}.
post_pre(_State) -> true.
post_pre(_State, _Args) -> true.
post_next(#state{keys = Keys, dict = Dict} = State,
          Var, [Value]) ->
    State#state{
      keys = listset:insert(Var, Keys),
      dict = dict:store(Var, Value, Dict)
     }.
post_post(_State, _Args, Res) when is_list(Res) ->
    true.

put_command(State) ->
    {call, ?MODULE, put,
     [oneof([modelutil:hash_gen()|
             modelutil:rnd_from_listset_gen(
               State#state.keys)]),
      comment_gen()]}.
put_pre(_State) -> true.
put_pre(_State, _Args) -> true.

put_next(#state{dict = Dict,
                nodeldict = NoDelDict} = State,
         _Var, [Key, Value]) ->
    case dict:is_key(Key, Dict) of
        true ->
            State#state{
              dict = dict:store(Key, Value, Dict),
              nodeldict = dict:store(Key, Value,
                                     NoDelDict)
             };
        false ->
            case dict:is_key(Key, NoDelDict) of
                true -> State#state{
                          nodeldict =
                              dict:store(Key,
                                         Value,
                                         NoDelDict)
                         };
                false -> State
            end
    end.
put_post(State, [Key, _], Res) ->
    is_substruct(case dict:is_key(
                        Key, State#state.nodeldict)
                  of
                   true -> ok;
                   false -> {error, not_found}
              end, Res).



delete_command(State) ->
    {call, ?MODULE, delete,
     [oneof([modelutil:hash_gen()|
             modelutil:rnd_from_listset_gen(
               State#state.keys)])]
    }.

delete_pre(_State) -> true.
delete_pre(_State, _Args) -> true.

delete_next(#state{dict = Dict} = State, _Var,
            [Key]) ->
    State#state{
      dict = dict:erase(Key, Dict)
     }.

delete_post(#state{keys = Listset}, [Key], Res) ->
    case listset:exists(Key, Listset) of
        true -> ok;
        false -> {error, not_found}
    end =:= Res.



weight(_S, get) -> 2;
weight(_S, post) -> 1;
weight(_S, put) -> 1;
weight(_S, delete) -> 1;
weight(_S, _Cmd) -> 1.

prop_db() ->
    ?FORALL(Cmds, commands(?MODULE),
            begin
                {H, S, Res} = run_commands(?MODULE,
                                           Cmds),
                clean_up(S#state.dict),
                pretty_commands(?MODULE, Cmds,
                                {H, S, Res},
                                Res == ok)
            end).

%%% Returns true if the Left element is a
%%% substructure of the Right element
is_substruct(SubTuple, OriTuple)
  when is_tuple(SubTuple), is_tuple(OriTuple) ->
    SubList = tuple_to_list(SubTuple),
    OriList = tuple_to_list(OriTuple),
    case {length(SubList), length(OriList)} of
        {Same, Same} ->
            BL = [is_substruct(Sub,Ori) ||
                     {Sub, Ori}
                         <- lists:zip(SubList,
                                      OriList)]
                and_colapse();
        {_, _} -> false
    end;
is_substruct([], []) -> true;
is_substruct(List, []) when is_list(List) -> false;
is_substruct(List, [Head|Tail])
  when is_list(List) ->
    is_substruct(
      sub_if_substruct(List, Head), Tail);
is_substruct(Original, Original) -> true;
is_substruct(_, _) -> false.
and_colapse([true|Tail]) -> and_colapse(Tail);
and_colapse([false|_]) -> false;
and_colapse([]) -> true.
sub_if_substruct([], _) -> [];
sub_if_substruct([Head|Tail], Element) ->
    case (is_substruct(Head, Element)) of
```

```
        true -> Tail;
        false -> [Head|sub_if_substruct(Tail,
                                        Element)]
    end.

clean_up(Keys) ->
    clean_up_aux(
      element(1, lists:unzip(dict:to_list(Keys)))).
clean_up_aux([]) -> ok;
clean_up_aux([H|T]) -> ok = delete(eval(H)),
                       clean_up_aux(T),
                       [] = list().

%% Interface selection
%% ===================
check_prop() ->
    model:start_link(),
    eqc:quickcheck(eqc:numtests(1000, prop_db())),
    model:stop().

%% Commands
get(Key) -> model:get(Key).
post(Value) -> model:post(Value).
put(Key, Value) -> model:put(Key, Value).
delete(Key) -> model:delete(Key).
list() -> model:list().
comment_gen() -> model:comment_gen().
```