Model-driven Testing of RESTful APIs

Tobias Fertig
tobias.fertig@t-online.de

Peter Braun
peter.braun@fhws.de

Faculty of Computer Science, University of Applied Science
Würzburg-Schweinfurt, Sanderheinrichsleitenweg 20, Würzburg, Germany

ABSTRACT

In contrast to the increasing popularity of REpresentational State Transfer (REST), systematic testing of RESTful Application Programming Interfaces (API) has not attracted much attention so far. This paper describes different aspects of automated testing of RESTful APIs. Later, we focus on functional and security tests, for which we apply a technique called model-based software development. Based on an abstract model of the RESTful API that comprises resources, states and transitions a software generator not only creates the source code of the RESTful API but also creates a large number of test cases that can be immediately used to test the implementation. This paper describes the process of developing a software generator for test cases using state-of-the-art tools and provides an example to show the feasibility of our approach.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Languages; Measurement; Verification

Keywords

REST; API; Model-driven Development; Model-driven Testing

1. INTRODUCTION

Nowadays, the amount of web services is continuously increasing. Many of these web services are using the architectural style for distributed systems called REpresentational State Transfer (REST), which was suggested by Roy Fielding in [6]. The proliferation of RESTful web services is evidence for the efficiency of his architectural style. REST complies with the modern requirements for web-based and mobile applications. Therefore, Application Programming Interfaces (APIs) following Fielding's approach are becoming more and more popular for developers.

In contrast to developing RESTful APIs and following Fielding's constraints, there is a lack of information about quality assurance for RESTful services. This can mislead developers, e.g. reducing the level of quality assurance once they are running out of time, which can then cause side effects for all clients using the API.

There are two ways to improve the quality of RESTful APIs. Firstly, quality assurance has to be present in literature about development of RESTful services, i.e. it has to be extended by how to test those. Secondly, quality assurance needs to take advantage of automated test case generation, which will lead to higher test coverage. Furthermore, developers will no longer have to write test cases manually.

This paper aims at the latter way and introduces an approach for automated test case generation via Model Driven Testing (MDT). There are two possibilities to perform MDT: If the API under test was developed by Model Driven Software Development (MDSD), there exists a model that can be used for automated test case generation. If that is not the case, the MDT approach still needs a model. The testers will then have to produce their own model or Domain Specific Language (DSL) in order to describe the existing API for the generators. Both ways are applicable with MDT.

We will prove that MDT is a reasonable approach for testing REST and that it is worthwhile to move further in this direction. We defined the following requirements our approach should fulfill in order to pass the evaluation: Firstly, the generated amount of test cases has to be very high, in order to justify the effort compared to developing test cases manually. A high number of test cases is of course not a goal on its own, but must result in a high level of code coverage. Furthermore, the model should only contain information about the design of the API under test and should not contain any explicit information about testing. Thus, even developers without any knowledge about quality assurance are able to generate all possible test case. Moreover, it has to be possible to test several combinations of arguments, headers and login data to benefit even more compared to manual testing.

The paper is structured as follows: In Chapter 2 we will discuss other approaches dealing with testing RESTful APIs. Chapter 3 will explain the important aspects of testing REST. In Chapter 4 the scope of our whole research project will be described. Furthermore, our generators and their development will be introduced. In order to prove the success of

MDT we will give an example of evaluation in Chapter 5. Chapter 6 will contain a short summary followed by an outlook on future works.

2. RELATED WORK

This section will summarize some related works about testing RESTful APIs, using either a classic approach or automated generation of test cases.

Several approaches to simplify testing RESTful APIs exist. Commonly used tools like JUnit[1], NUnit and other xUnit frameworks aim at unit-testing. Their tight-coupling with the implementation language of the subject under test makes it difficult to use them for testing web-services. Therefore, many teams are searching for techniques to improve testing such services. One approach is SoapUI[2], a tool which can configure test cases for web-services by using a Service-Oriented-Architecture. The fact that it can mainly be configured via its graphical interface makes it difficult to enable an automated test case generation. Thus, it did not fulfill our requirements.

Another approach improving the development of test cases for RESTful services is Haleby's REST-assured[3]. He developed a framework for rapidly writing test cases for any RESTful API. The syntax of his tool follows the so-called when-then-rule. Moreover, every test case can be configured by using a fluent interface[4]. As this speeds up only manual testing, we decided not to use it since we had already developed test cases using the JUnit framework, which were used as templates during software generation.

Furthermore, we discovered an approach using automated test case generation. Test-the-REST (TTR) is an HTTP testing tool especially designed for testing RESTful web-services. Chakrabarti et al. explain their tool in [3]. It is based on XML-files configuring the required test cases. In their sixth section they mention that they could generate over 40,000 test cases with TTR. The development team using their pilot decided to continue using it as they could eliminate plenty of bugs. Chakrabarti's validation was one of the reasons for us to focus on MDT for testing RESTful APIs. His work proved that it is possible to develop an automated test case generation for RESTful APIs. However, their approach still requires too much knowledge about testing to configure all test cases via XML. This additionally knowledge is, however, contradictory to our predefined requirements for the evaluation of our approach.

3. TESTING RESTFUL APIS

After the outline of related work the different kinds of test cases required for testing REST will be discussed in this chapter.

3.1 Functional Testing

Although testing the functional parts of RESTful APIs is one of the most basic steps, we will nevertheless summarize its most important aspects. A correct response body should contain the representation of the requested resource, using the requested media type or at least a valid HTTP Error Code Message. On the one hand, response headers have to be valid HTTP headers and on the other hand, they also have to match the given status code of the response. Furthermore, we determined every header to be tested in its own test case, thereby fulfilling the convention that a test case should focus on one single aspect only. Due to the similarity of each of those test cases, MDT is very useful. It can be exhausting for a human developer to write hundreds of similar test cases differing in only one line. The developer would try to speed up via copy and paste, which then often causes bugs. MDT fixes this problem as it is even easier to write generators when test cases are very similar.

Apart from checking headers and bodies of responses there has to be a validation of returned URLs. Every URL should point to an existing location since even a single broken link can cause the hypermedia mechanism to fail. Following Fielding's constraint of self-descriptive messages, at least one useful message has always to be returned by the server. We decided against crawling through returned URLs and are instead only checking the first level. Any URL found at deeper levels will usually be checked by the test cases for another endpoint.

3.2 Security Testing

Dealing with security often leads to topics like cross-site-scripting (XSS) or SQL-injection, but there are even simpler aspects we need to check. If a developer tests their web service they should at least write some test cases for authorization. Thus, there will be a few test cases covering only some combinations of states for username and password. This is due to the huge amount of time needed for manually checking all possible combinations. Username and password can each have the states VALID, INVALID or MISSING, so there are nine different combinations testable. Here, we recognized the same facts as in Section 3.1: MDT is again very useful regarding the similarity between the test cases. We can generate one test case for every combination just by using one additional for-loop to iterate over all possible combinations. Furthermore, we can test the authorization functionality of every single endpoint provided by the API. We can, therefore, detect whether an authorization is missing on any endpoint.

On our first attempt, we reduced the security testing to checking all possible combinations of states. However, we strongly recommend to also do some penetration testing on your web services. For further information about penetration testing, see Antunes' and Vieira's work in [1].

3.3 Performance Testing

There are several tools aiming at performance testing for example Apache JMeter[5]. In order to use MDT for performance testing additional information in the model is needed. The distribution of different HTTP requests is therefore required to describe different user behavior. The tester should at least receive information about the shortest and longest duration supplementary to the average duration as result of the performance tests. Moreover, the duration of the whole performance test has to be configured. Since it is only possible to make a performance statement about the API if the test has been running long enough, this value should at least be set to 60 minutes.

[1] `www.junit.org/`
[2] `www.soapui.org/`
[3] `code.google.com/p/rest-assured/`
[4] `www.martinfowler.com/bliki/FluentInterface.html`
[5] `jmeter.apache.org`

3.4 Behavior Testing

Testing the behavior of RESTful APIs often is a subjective task. According to [7] there are different ways of responding to a request. Furthermore, some developers prefer conditional requests defined in [8] to assure that resources can only be updated or deleted if the client has knowledge about the latest state of the resource. All different kinds of behavior of a given RESTful API have to be tested so that clients using this API can be sure that the API acts like promised. It is no big deal to provide all these similar test cases for each endpoint with MDT.

3.5 REST Compliance

We have to mention that we did not focus on this topic in our first iteration of research as we wanted to prove the usability of MDT for testing REST first. In his blog [5] Fielding criticizes that many existing APIs are called RESTful although they do not adhere to his constraints. He mentions Hypermedia As The Engine Of Application State (HATEOAS) as one of the most violated constraints. An approach covering every aspect of testing RESTful APIs has to check the compliance to Fielding's constraints.

4. MODEL-BASED TESTING

In this section we describe the scope of our whole research project. It will also contain information about the framework we used to develop our generators.

4.1 Scope of research project

Our research group aims at the improvement of developing RESTful services. There are many APIs called RESTful although their developers violated Fielding's constraints. Using a model driven approach enables development teams with less experience about REST to comply with these constraints.

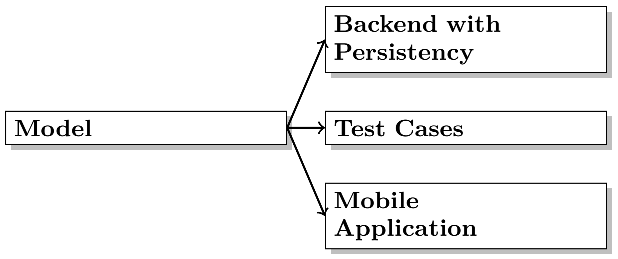

Figure 1: Provided scope of our research project.

Figure 1 shows the scope of our research project. The idea was to develop a DSL to describe a required RESTful API. The generators will then use this description to generate an entire backend including persistence. Moreover, to achieve a high test coverage, all test cases will be generated. Finally, in order to improve the development of mobile applications, the communication layer for communication with the backend will also be generated.

At the beginning of our project Schreibmann focused on the MDSD of the API including persistence [9]. After releasing his model we started to work on the MDT approach. However, the part for mobile applications is still listed on our to-do list.

A short example on how to use the model for describing an API can be seen in Listing 1. After a few basic configurations, e.g. the definition of the base path or choosing a security mechanism, the description of resources begins. We enabled an inheritance mechanism to extract common settings like media types or identifiers. The attributes of every resource can be listed in curly braces. Listing 1 contains only one resource description, however, the required API can contain as many resources as required.

Listing 1: Describing an API using our model.

```
BasePath "api"
Security {
    Authentication by HTTPBasic
}
AbstractResource Base {
   Long id as key
   MediaType "application/json"
}
Resource User extends Base  {
    String userName
    String password
    String firstName
}
```

4.2 Xtext and Xtend

There are different tools for MDSD and we decided to use the most common ones: Xtext[6] and Xtend[7]. For a detailed description of how to use both we recommend Bettini's book [2].

Xtext is a framework used for implementing programming languages or DSLs. By using Xtext, we can design a model for RESTful APIs and generate our test cases based on the given description. The framework provides all required tools like parsers, linkers or compilers for the designed DSL. Only a short Xtext file is required to define the grammar for describing RESTful APIs.

Listing 2: An Xtend syntax example.

```
     '''
public void «getMethodName()»() {
    «IF authorizationHeader.equals("")»
    generateAuthorizationHeader();
    «ELSE»
    setAuthorizationHeader(authorizationHeader);
    «ENDIF»
}
    '''
```

Xtend is a Java dialect that can be used for implementing code generators within Xtext. We used Xtend to provide templates for our test cases. Furthermore, we can generate Java code which looks like manually written code due to the fact that Xtend supports white spaces and intending. Listing 2 gives a short example for explaining the syntax of Xtend. The three single quotation marks are used to define the included code as a template. Later Xtend will generate Java code from it by keeping the defined intending and whitespace. Within the French quotation marks it is possible to use If-Else or For-Each statements what enables iterating over collections and generating Java code based on the content of the collections. The whole code in-between French

[6] www.eclipse.org/Xtext/documentation.html
[7] www.eclipse.org/xtend/documentation.html

quotation marks is Xtend code and will be executed before the Java code is generated. For a more detailed documentation of the whole Xtend syntax we recommend the official Xtend guide.

4.3 Test Case Templates

We recognized the similarities between the different kinds of test cases in our existing unit test cases. Thanks to the template mechanism provided by Xtend we had the possibility to use those templates for the generation of test cases. One main structure can be extracted from the tests: If required there will always be a preparation of target URLs and used resources first. Hence, the preparation in the template has to be configurable. Afterwards the request will be built and executed. In doing so, there can be different headers for every request, e.g. different authorization information. As before, the template needs another placeholder for those headers. After the execution of the request each test case has to evaluate one single aspect of the given response. The assert statement at the end of the test case has therefore to be flexible, too.

We decided to use different templates for each kind of HTTP request. This is due to the fact that some request types like GET and DELETE do not need a resource attached to the request body while others like PATCH, POST or PUT do. Furthermore, we also decided to provide different templates for the same kind of HTTP request to achieve a separation for different purposes. For example there is a template for default DELETE requests and one for conditional DELETE requests. In general, we decided to provide an extra template for conditional requests.

Listing 3 defines our default template for GET requests. All similar parts of the test cases are hard coded into the templates. All varying parts are adjusted with Xtend using French quotation marks. In Listing 3 it can be seen that the name of the test case is also written in French quotation marks. This is to provide meaningful names even within the generated test cases what simplifies evaluating the test reports.

Listing 3: Default template for GET requests.

```
   '''
@Test
public void «spec.methodName»() {
  CloseableHttpResponse response = getHttpClient()
            .setTargetUrl(«spec.targetUrl»)
            «IF spec.authHeader.equals("")»
            .setNoAuthHeader()
            «ELSE»
            .setAuthHeader("«spec.authHeader»"))
            «ENDIF»
            .executeGetRequest();

  «FOR command: spec.preAssert»
  «command»;
  «ENDFOR»
  «spec.assertion»;
 }
  '''
```

4.4 Test Case Specifications

In Listing 3 the name of the variable *spec* can be seen. This variable represents a specification builder we designed to improve extensibility and changeability. We added a field in our specification variable for every placeholder in the templates. We can build different collections of specifications with a class implemented as a fluent interface. Thus, it is easy to add multiple specification objects and finalize the collection with a call to the *build()* method. We developed methods to build specifications for every possible test case scenario.

Due to the possibility of iterating over collections within Xtend templates we created methods for generating a collection containing all necessary test case specifications. This is used to check whether a response to a given request contains the status code, the headers and the body defined in the RESTful API. Our generators will use these collections to generate all required test cases.

If the requirements for a specific HTTP request change, only the settings in the *SpecificationsBuilder* class have to be adjusted. Every test case used for checking the behavior of the concerned endpoints can be regenerated and will fit into the new requirements.

4.5 Test Case Generators

After introducing the test case templates and the specifications for configuring them, the generators are still missing. There are different generator classes, one for every topic under test. There are generators for authorization test cases and classes for equivalence class and boundary testing. We also implemented generators to provoke internal server errors while testing. Moreover, we added some to cover the *405 - Method Not Allowed* error. There are many possibilities and topics a generator can be written for. We also generated test cases for semantic purposes, e.g. attributes limited to numbers greater than zero. Another advantage of automated test case generation is the possibility to test all combinations of attributes, e.g. while testing query methods or endpoints for partial updates via PATCH request [4]. The generation of all possible combinations is achieved by an algorithm based on binomial coefficients.

Listing 4: An example for generating test cases.

```
'''
«FOR authHeader: unauthorizedAuthorizations»
  «FOR spec: specBuilder.buildUnauthorized()»
      «spec.setAuthHeader(authHeader)»
      «testCases.generateGET(specification)»
  «ENDFOR»
«ENDFOR»
'''
```

Listing 4 shows the code for generating about 30test cases. The outer collection *unauthorizedAuthorizations* contains combinations of username and password which should result in a *401 - Unauthorized* response. The inner collection contains all specifications required to check whether the headers, the status code and the body of the response match an unauthorized response. If one takes into account that this happens for every endpoint of the API a rather small API with ten endpoints would then result in 300 test cases.

Due to our design we can easily change the specifications for unauthorized test cases and all created test cases will be updated. It is also possible to wrap our code from Listing 4 in an additional for-loop if multiple test cases for each single aspect are required.

5. EXPERIMENTAL VALIDATION

After describing our generators we will give an experimental validation in this section. We will determine the API under test and afterwards validate our approach.

5.1 The API under test

The approach for generating RESTful APIs developed in our research project was published by Schreibmann in [9]. Through testing his generated code we were able to determine whether MDT is a reasonable alternative to test systems generated by MDSD. Moreover, we could validate if it is possible to apply MDT to RESTful APIs.

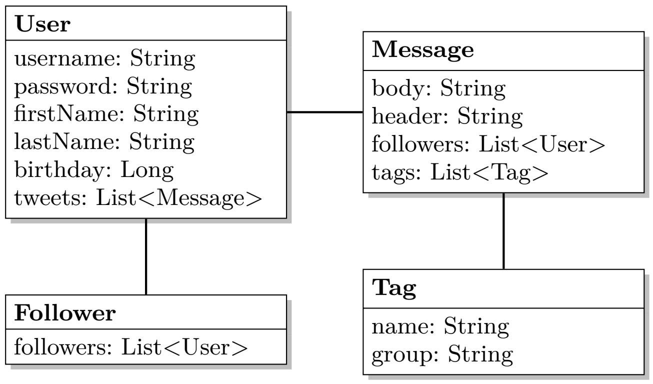

Figure 2: Class diagram of resources.

With his model we described the resources of a simple API shown in Figure 2. Associations in the class diagram easily show which class is contained in the collection attributes. We decided to use only four resources enabling us to determine a formula to approximate the minimum amount of test cases. Schreibmann's model provides the ability to define ranges for every attribute. Ranges for numbers define valid intervals while ranges for strings define its size. Regarding character strings, a semantic condition can be defined, too. Via semantic condition, the API ensures that the defined string will always represent for example an email address or URL.

5.2 Information in the model

Whether an existing model is used for MDT or not, the model should not contain any information about quality assurance. This was one of the requirements for our approach in order to pass the evaluation.

Listing 5: A resource with attribute ranges.

```
Resource resource extends Base  {
    String ("5" ... "15") email attribute1
    String url attribute2
    Long (100L ... 1000L) attribute3
}
```

The model implemented by Schreibmann in [9] provides enough information for MDT to generate test cases. The model contains information about ranges which can be used for equivalence or boundary tests. Listing 5 shows how to specify ranges and semantic information for attributes. The names of the resources and their attributes can also be extracted. Even semantic tests are possible due to the semantic conditions for attributes of type string. In case of unbounded attributes no ranges have to be defined. This applies to the semantic information, too.

However, we had to disregard our defined rule about additional information in order to enable performance testing. There has to be a possibility to define the distribution of the varying HTTP requests in order to simulate different user behavior. Moreover, the duration of the performance test has to be adjustable. However, for better compliance with our predefined requirements we defined these settings as optional. If performance tests are irrelevant or the tester has not enough experience with them they can just skip these settings. No performance test cases will then be generated.

5.3 Minimum amount of test cases

In Section 1 we defined that our approach has to ensure that the amount of generated test cases has to be at least high enough so that it cannot be done manually. Only then is the effort to implement the DSL and the generators justified. We generated a complete set of test cases for our reference API to validate MDT. Within our first iteration about 14,000 test cases could be generated for the user resource containing six attributes. About 20,000 test cases were generated for the whole API. After the generation we were able to count the amount of different test cases depending on the attributes of a resource. Moreover, we counted the default test cases which are generated for every endpoint. Based on these numbers we created Formula 1 to approximate the minimum amount of test cases depending on the amount of attributes n.

$$f(n) = \sum_{k=0}^{n} \binom{n}{k} * 160 + 740 \tag{1}$$

The first addend of Formula 1 represents the number of test cases depending on the amount of attributes a resource contains. We covered all possible combinations of attributes in order to test endpoints for queries or partial updates. We provided about 160 different test cases using all combinations. The second addend represents our default test cases. We provoked different errors and checked the authorization functionality on every endpoint what adds up to 740 test cases.

With this rough approximation the minimum amount of test cases for a resource containing only one attribute results in 1060 test cases. A real API will contain a lot of resources with more than one argument. Thus, our approach passes the first criteria mentioned in 1. However, we strongly recommend running the test cases at night since our systems could run about 200 test cases per minute. Therefore, even an API containing only one resource with one attribute will need five minutes to complete all test cases.

5.4 Result

Since of TTR generated about 42,000 test cases for a realistic API, we proved the success of MDT as it created 20,000 test cases for a very small API. Furthermore, there were some bugs which were only discovered because of testing all possible combinations of authorizations or attributes in queries.

Another advantage is that the model needs no additional information. A developer with knowledge about RESTful APIs can easily define their API and does not need any special knowledge about how to test REST. This is an important fact especially with the lack of information about testing REST in literature, we mentioned in Section 1.

There are many debates about how to test a system generated by MDSD. We recommend to consider MDT as a tool for quality assurance when using MDSD due to the success of testing our API with MDT. If there already is a complete model or DSL for developing a system no high efforts are needed to generate test cases with the given information. Nevertheless, we recommend an efficiency check with alternative approaches in case of a missing model.

6. SUMMARY

This paper described an approach for MDT of RESTful APIs. Actually, two reasons exist why testing RESTful web services can fail. Firstly, there is a lack of information about how to test REST both in literature and in the world wide web. Secondly, if projects are running out of time their managers often tend to reduce quality assurance instead of reducing the amount of functionality provided. The aim of this paper was to encourage the use of tools to automatically generate all required test cases. As this generation does not need any additional time the profit of reducing quality assurance will disappear. The time for gathering knowledge about testing REST will furthermore be reduced if the team can use an existing approach.

At the beginning we summarized related works to prove the need of automated test case generation. Afterwards we explained the different kinds of testing which have to be taken into account for quality assurance of RESTful APIs. After this consideration we introduced our model driven approach. We designed some test case templates using placeholders and specification classes to fill those placeholders during the generation process. Each generator class focuses on one kind of testing REST. Before filling the templates with concrete information it generates the required specifications. If requirements change, only the specification classes will be affected. The evaluation based on a RESTful API generated by MDSD was successful as we could generate over 20,000 test cases for an API containing only four resources. Furthermore, no additional information about testing was needed in the model, only about describing the API under test.

After we proved the usability of MDT for testing RESTful APIs there are multiple tasks to focus on in future work. In Section 3.5 we mentioned that we did not focus on REST compliance yet. In our next steps we have to focus on this topic in order to enable test cases on a more abstract level. Moreover, the importance of extended testing if an API complies with Fielding's constraints cannot be denied since Fielding published [5] on his blog. However, the other topics on testing REST, e.g. security testing or performance testing offer plenty of testable alternatives, too. Within our approach we did not provide any penetration testing which could also be extended.

Another possible direction is aiming at a framework which offers an independent model to describe any existing RESTful API. The process of generation should be divided into two steps, one for gathering information about returned error messages and one for the final generation of test cases using the given information of the model and the gathered information of the first step.

7. REFERENCES

[1] N. Antunes and M. Vieira. Penetration testing for web services. *Computer*, 47(2):30–36, Feb 2014.

[2] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. EBL-Schweitzer. Packt Publishing, Limited, 2013.

[3] S. 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, Nov 2009.

[4] L. Dusseault, L. Lab, and J. Snell. RFC 5789, PATCH Method for HTTP, 2010.

[5] R. Fielding. REST APIs must be hyper-text driven. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.

[6] R. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1, 1999.

[8] R. Fielding and J. Reschke. RFC 7232, Hypertext Transfer Protocol – HTTP/1.1: Conditional Requests, 2014.

[9] V. Schreibmann. Design and Implementation of a Model-Driven Approach for RESTful APIs. In *Proc. Fifth IEEE Germany Students Conference 2014 Passau*, 2014.