Automation of Integration Testing of RESTful Hypermedia Systems: A Model-driven Approach

Henry Vu, Tobias Fertig and Peter Braun
Faculty of Computer Science and Business Information Systems, University of Applied Sciences Würzburg-Schweinfurt, Sanderheinrichsleitenweg 20, 97074 Würzburg, Germany

Keywords: REST, Integration Testing, RESTful API, Hypermedia Testing, MDSD, MDE, MDT, Model-driven Testing.

Abstract: The proper design of Representational State Transfer (REST) APIs is not trivial because developers have to deal with a flood of recommendations and best practices, especially the proper application of the hypermedia constraint requires some decent experience. Furthermore, testing RESTful APIs is a missing topic within literature. Especially hypermedia testing is not mentioned at all. Manual hypermedia testing is time-consuming and hard to maintain. Testing a hypermedia API requires many test cases that have similar structure, especially when different user roles and error cases are considered. In order to tackle this problem, we proposed a Model-driven Testing (MDT) approach for hypermedia systems using the metamodel within our existing Model Driven Software Development (MDSD) approach. This work discusses challenges and results of hypermedia testing for RESTful APIs using MDT techniques that were discovered within our research. MDT allows white-box testing, hence covering complete program structure and behavior of the generated application. By doing this, we are able to achieve a high automated test coverage. Moreover, any runtime behavior deviated from the metamodel reveals bugs within the generators.

1 INTRODUCTION

The Web has become the deployment environment for software systems and applications. Office productivity applications and corporate tools such as invoicing, purchasing and expense reporting systems have migrated to the Web (Taivalsaari and Mikkonen, 2017). Web APIs have become the vital backbones for these applications and services. While the number of Web APIs is increasing, the need for good API design has become more crucial than ever before. Good APIs can be among a company's greatest assets, as customers invest heavily in buying, writing and learning them. However, bad APIs can also be among a company's greatest liabilities as they result in never-ending streams of maintenance and support (Bloch, 2014).

In 2000 Fielding (Fielding, 2000) presented an architectural design for building network-based applications that resemble the Web called REST. REST style requires that an application server must adhere to a set of constraints, such as client-server, stateless, cache, uniform interface, layered system and hypermedia. An API can be described as RESTful when making use of every Fielding's constraint including hypermedia (Richardson, 2009). Hypermedia constraint has tree jobs according to (Richardson et al., 2013): 1) It tells the client how to construct an HTTP request, what method to use, what URL to use, what HTTP headers and/or entity-body to send. 2) It makes promises about the HTTP response, suggesting the status code, the HTTP headers, and/or the data the server is likely to send. 3) It guides the client through the application workflow given by the server. Non-hypermedia servers have several problems: a) If a server does not dynamically generate hyperlinks, clients are forced to construct these hyperlinks piece by piece which would require prior knowledge about the implementation details of the server, b) since there are no hyperlinks for clients to follow, there is no application workflow, thus it is rather a static API and c) this client-server architecture is tightly coupled and is likely to break due to changes on either side: If the server changes the URIs, the clients will break and if any client is to be modified, the server must remain the same.

We decided to tackle the challenges of RESTful API development with a MDSD approach. In 2015 we proposed *Generating Mobile Applications with RESTful Architecture* (GeMARA) (Schreibmann and

Vu, H., Fertig, T. and Braun, P.
Automation of Integration Testing of RESTful Hypermedia Systems: A Model-driven Approach.
DOI: 10.5220/0006932004040411
In *Proceedings of the 14th International Conference on Web Information Systems and Technologies (WEBIST 2018)*, pages 404-411
ISBN: 978-989-758-324-7

Braun, 2015). Instead of a data-first approach which uses data models for creating an API, we went for an API-first approach which aims at a proper API design first which then automatically creates the underlaying database. The main idea of this project is to take RESTful API development to a higher level of abstraction by using a metamodel as input. We use our own Domain Specific Language (DSL) to describe a metamodel which is then to be translated into a RESTful API. This way, we can force consistency and achieve a higher standard of quality by encapsulating reliable and well-known libraries, frameworks and RESTful best practices behind our DSL.

We also explored the possibility of MDT (Fertig and Braun, 2015) and realized the lack of information about MDT. In general, MDSD processes are very sensitive to the introduction of defects. Any defect in a model or a model transformation can be easily propagated to the subsequent stages, thus causing the production of faulty software (González and Cabot, 2014). However, MDT is a possible way to achieve correctness within the generators. Test cases can be generated from the underlying model. Any deviating behavior of the application on a runtime environment could reveal possible bugs within the generators. Moreover, even third-party frameworks and libraries can be updated or replaced over time, these generated test cases can also be used to verify our platform code. Facing the need for better RESTful API design and deficits in its quality assurance, we are focusing on answering the following research questions (RQ):

RQ 1) How to generate appropriate test cases from an existing metamodel to test role-based behavior for every application state within a RESTful API?

RQ 2) How can test cases derived from RQ 1 be verified automatically on runtime?

RQ 3) Can the proposed approach completely relieve API developers from integration testing?

2 RELATED WORK

According to Fielding (Fielding, 2008) RESTful systems must be hypertext-driven, in other words these systems are to be designed as ε-NFAs. In (Zuzak et al., 2011) and (Hernández and García, 2010) the authors also present their ε-NFA formal models for specifying RESTful APIs based on their understanding. It is important to present a formal model in oder to assign formal semantics because of the benefits for formal validation and testing.

The established literature concerning REST such as (Richardson et al., 2013), (Amundsen, 2017) and (Webber et al., 2010) reveal little to no information about its quality assurance. Moreover, hypermedia testing is not mentioned at all. They explain the purposes of hypermedia, but do not present any approach to test it. REST API integration testing by sending HTTP-requests and verifying the received responses was mentioned in (Webber et al., 2010).

In (Frankel, 2015), the author suggests three different "entry levels" for integration testing: 1) At HTML level using a WebDriver tool such as Selenium (Project, 2018) to interact with HTML/CSS elements such as filling a form or clicking on a button. 2) At HTTP level by sending HTTP requests and checking HTTP responses. 3) At controller level by directly testing the methods. The main idea of this book is testing must be efficient and economic, thus recommending against a high test coverage and complex logic. However, our MDT approach enables automation of integration testing with a high test coverage with minimum manual effort, hence generating great economic return. Nevertheless, the author also neglects hypermedia testing by suggesting to manually craft each HTTP request with a fixed URL.

In (Chakrabarti and Kumar, 2009), the authors present their own framework Test-the-Rest to test HTTP based web services. The test cases are written in a test specification language based on XML to give the tester a structured approach. Other than that, response validation only depends on checking media type and status code. This approach does not fully address the challenges of testing RESTful APIs.

To the best of our knowledge, there is a lack of information about generating and testing RESTful systems, particular hypermedia testing.

3 CHALLENGES

The most important component in our metamodel to describe a RESTful API is the application state. An application state is a pair of a HTTP method and a resource. It represents a valid REST request to access a resource. Another central key element in a RESTful system is the resource. It is important to note that resource is not a storage object, but it is instead a conceptual entity. A resource represents a single object or a collection of objects. The intention of HTTP methods should be fixed, and developers should not be free to choose wrong methods. The four basic operations to create, retrieve, update and delete (CRUD) resources are mapped to the four HTTP methods POST, GET, PUT and DELETE. A

transition is our formal way of modeling relationships between application states. A client can navigate from one application state to another via transitions.

In 2017 we proposed several approaches to deal with MDT for RESTful systems which include server-side testing and client-side testing (Vu et al., 2017). MDSD is a software engineering paradigm that promotes the utilization of models as primary artifacts in all software engineering activities (González and Cabot, 2014), therefore, we have to ensure that there are no defects in our model in the first place. Our MDSD approach has two distinct artifacts that need to be tested on the server-side: the model and the generated code from the model. These artifacts are to be tested in a bottom up manner because they are built on top of each other. Errors within the model can propagate subsequent bugs in the generated source code which lead to undesired system behavior at runtime or production. Therefore, our model-driven hypermedia testing on the server-side is divided into two sequential steps: static and dynamic analysis. Our findings within the static analysis were presented in (Vu et al., 2018) covering up an automatic verification process for the input-model, thus ensuring its hypermedia characteristic as a ε-NFA before triggering code transformation. This paper is a follow-up contribution within our research regarding testing RESTful APIs using MDT techniques.

The challenges of the dynamic analysis are addressed withing the scope of this work. Dynamic analysis, as opposed to static analysis, always requires the execution of the software. The simplest form of dynamic testing is the execution of the software by a test person, thereby the tester can enter any input to operate the software. This is an unsystematic ad-hoc approach and thus inaccurate and usually not reproducible (Liggesmeyer, 2009). However, using MDT techniques allows us to follow a more novel approach. Once the model is verified by the static analysis, our tool will generate a functional server. This generated server should work correctly and include all the features described by its model. But since the generators are manually implemented, we cannot guarantee this. The aim of this procedure is to test the correct functionality of the transformation process by testing the generated code, thus detecting bugs within the generators. The dynamic analysis can be used to test several aspects of a RESTful API: 1) It checks whether a ε-NFA-compliant model correctly produces a ε-NFA-compliant RESTful API 2) We also have to consider the authorization concept of the application. 3) Additionally, we can also provoke negative tests to see how the server handles error cases on runtime.

The dynamic analysis is divided into three phases: First, model crawling phase, second, building an HTTP crawler, and third, generation of test classes. To simplify understanding, it is necessary to present an application example which will be the basis to demonstrate our further approaches. Our application example represents an online shop. There are two user roles: customer and admin. A customer can view items whereas a shop admin can also create, update or delete items. For the sake of clarity, we omitted transitions back to the dispatcher state and self-pointing transitions. The entire application workflow with all application states and possible transitions is described as an ε-NFA in the Figure 1.

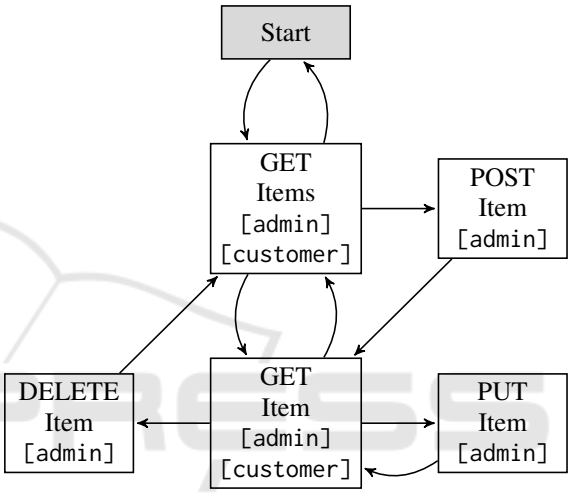

Figure 1: Workflow of the application example.

The concept of ε-NFAs revolves around a finite set of states with an initial state, a set of possible inputs and rules to map each state to another state, or to itself (Wright, 2005). Speaking in REST terms, a client can enter the application workflow through the initial state and only be in exactly one state at any given time and it can only change its current state by navigating through a directed transition with a valid input. ε-NFA compliance means that every state within an application is accessible. The client should be able to start from the entry point of the RESTful API, and it should be able to visit every application state and go back to the start state without getting stuck in a dead-end. We argue that within a RESTful application there are no accepting states. An accepting state assumes that a client has finished a task within the application workflow after a sequence of inputs. The question arises "When is a task finished?". Taking Twitter for example: Entering the Twitter app gives us our newsfeed. This could be argued as a task done or an accepting state, but navigate to a friend's page could also be a task and so is creating, modifying or deleting a post. It does not matter if the client wants to

stay in the loop of the application workflow or leave it at any given state. Due to these argumentations, we can perform this simplification that our ε-NFA model of a RESTful API does not have accepting states.

Another challenge of hypermedia testing is to check whether the generated API delivers appropriate hyperlinks based on the client's user role. Each user role sees a different representation of the API depending on its access rights. In other words, the API must guide the client through the workflow based on its user role, otherwise the authorization concept would be corrupt.

Until now, we have looked at several challenges to check correct hypermedia-driven behavior. This falls under the category of positive testing because our model allows white-box testing, hence revealing complete program structure and behavior. Therefore, we can perform a sequence of valid inputs and check for expected outcomes. But what if the client makes invalid requests? We must also test the robustness of the system by checking its ability of handling error cases. The correct response for an invalid request within a hypermedia-driven system should contain at least a self-descriptive message (Fielding, 2000) to tell a client what to do next. The dynamic analysis can also be used to provoke negative tests. Instead of verifying hypermedia behavior, we can test how the server reacts to a client's non-hypermedia behavior. Since dynamic testing allows runtime execution of the to-be-tested system, a client can intentionally send false requests to test the robustness of the server: Whether it remains functional or breaks, or whether an appropriate response is given in the event of an unauthorized request. Each state-to-state transition requires specific inputs in order to perform. For instance, if a client wants to enter the *POST Item* state its request header must include authorization with credentials of an admin and its request body must include a proper `Item` resource representation (e.g. JSON) as payload. The server has to validate two input types and handle several error cases:

1) Authorization header: If the client is not authenticated to the server, the server must response with a proper HTTP code such as `401 Unauthorized`, indicating that the request lacks valid authentication credentials for the target resource. Or if the client is authenticated to the server but does not have permission to access an application state, the HTTP response code has to be `403 Forbidden`, indicating that the server understood the request but refuses to authorize it due to the application logic.

2) Entity-body: If the request body is empty or not in the correct format, the server has to return a `400 Bad Request`, indicating the request could not be understood by the server due to malformed syntax. The client should not repeat the request without modifications. If the request body is in the correct format but contains invalid data e.g. price of an item should not be smaller than 0, the server has to respond with a `422 Unprocessable Entity`, meaning the server understands the content type of the request entity but was unable to process the contained instructions due to the application logic.

In addition to self-descriptive messages, we also have to ensure that the client will be redirected to its previous application state or at least to the dispatcher state.

4 APPROACH

In order to carry out a dynamic analysis, it is necessary to perform two crawling processes: first model crawling and then HTTP crawling. The model crawling process is intended to derive information from the underlying model to build appropriate test cases. Afterwards, an HTTP client will test against these test cases when the generated code is deployed on a runtime environment. It is also necessary to generate test data based on the model to prevent a client from performing tests on an empty server. At last, a generator will combine test cases constructed by the model crawler and functionalities of the HTTP client to produce runnable test classes.

4.1 Model Crawling

The model crawling process is divided into three parts to retrieve crucial information for our test case generation: 1) Verifying hypermedia response, 2) deriving test paths to visit every state based on the application workflow and 3) negative testing.

A client expects to only see its permitted hyperlinks at any given state. If a request is valid because the client is authorized to enter an application state, response of this application state must contain the same hyperlinks as given by the model to this role. If a request goes wrong, the client must be provided with an appropriate response code or redirected to the dispatcher state. To accomplish this task, we have to map every incoming transition of an application state with its outgoing transitions with respect to a specific user role. This is necessary, because the client only knows the URI to the entry state. Every other URI is dynamically generated by the server. So, in order to determine whether a hypermedia response is correct we have to map every relation type to permitted following relation types while considering the client's

user role. This way, an HTTP client is able to make a request to a relation type and expects to see the corresponding set of following relation types to its request by checking these mappings. In order to accomplish this, the model crawler loops through every state and check if a state can be accessed by a specific user role. From any given state: A user role is permitted to access a set of following states. So we can map a state's incoming transition as a key to a set of permitted outgoing transitions.

In the previous step we know what hypermedia response to expect after making a request to a particular application state through our mappings. Nevertheless, once the server is deployed on a runtime environment, the HTTP client cannot verify these responses by making direct requests to the URI endpoints but it rather must start from the entry state and navigate through the application workflow. This way, we can make sure that responses of every application state have been tested at least once. This step is to derive all possible test paths within a role-based workflow from the existing model to generate test cases for the crawler.

We decided to develop a depth-first search algorithm that derives all possible paths for every user role from the model. This algorithm spans a role-based specific workflow from a directed graph into a tree. The tree represents every possible path within a workflow. Each path represents a task. For instance, a task could be update an item. To manage this task, a client with admin role must sequentially visit these application states: *Start, GetItems, GetItem, PutItem*. The root element is the *Start* state, it represents the entry point of the API. A valid path ends when the client is directed back to one of the previous states in the current path. For example, after changing the price of an item from a *PutItem* state, the server will redirect the client to *GetItem* state which the client has visited before. When a valid path ends, it is marked as visited and the HTTP client starts at the *Start* state again to crawl through the next path.

A task is considered successfully tested when an HTTP client is able to visit all edges of the role-based application workflow. Any occurring error can be tracked down by looking at the path where the client fails to walk through. According to REST style, every task within the workflow must be reachable from the entry point, so this is the best way to approve this premise. Once the client manages to walk through every path of the tree and to verify every obtained response on its way, then we can make a definite statement about the correct hypermedia behavior of the RESTful API.

This step is to derive role-based positive test paths from the model for an HTTP client to test against later on runtime. To achieve this, we have to span the role-based representation of the application workflow into a tree with the start state as root. By definition, every hypermedia API represents a ε-NFA or it can also be defined as an complete unweighted directed graph *G = (V,E)* with *V* is the set of possible application states and *E* is the set of possible state-to-state transitions.

Assuming our ε-NFA is a directed graph $G = (V,E)$ with $V = \{A,B,C,D,E\}$ and $E = \{(A,B),(B,A),(B,C),(C,B),(B,D),(D,C),(C,E),(E,B)\}$ shown in Figure 2.

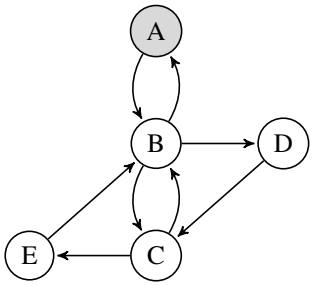

Figure 2: Graph representation of the application workflow.

At the beginning we have not walked down any path yet, hence the *Visited edges* set and the *List of paths* are empty and the *Unvisited edges* set contains all available edges. A path is defined as a sequential list of edges and an edge is equivalent to a transition in our RESTful context.

- *Unvisited edges* = $\{(A,B),(B,A),(B,C),(C,B),(B,D),(D,C),(C,E),(E,B)\}$
- *Visited edges* = $\{\}$
- *List of paths* = $\{\}$

The algorithm terminates when the last edge has been visited at *(B,A)*, it walks backwards and finds no further unvisited outgoing edge from every node along the path, subsequently it marks all edges left as visited.

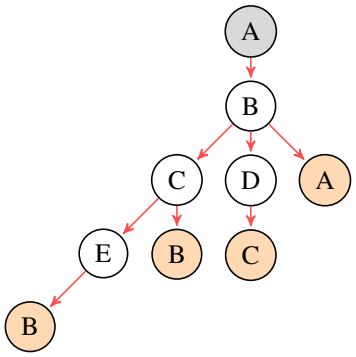

Figure 3: Deriving last path.

Our final list of paths now contains all possible paths from the start node *A* and there is no unvisited edge left.

- *Unvisited edges* = {}
- *Visited edges* = $\{(A,B),(B,A),(B,D),(D,C),(E,B),(C,E),(C,B),(B,C)\}$
- *List of paths* = {{(A,B), (B,C), (C,E), (E,B)}, {(A,B), (B,C), (C,B)}, {(A,B), (B,C), (C,B)}, {(A,B), (B,A)}}

To explain this process in further detail, we take a look at the pseudo code in Algorithm 4.1. Our algorithm converts a directed graph into a tree, similar to block-cut trees (Hopcroft and Tarjan, 1973). The difference to block-cut trees is that we use paths instead of biconnected components.

Algorithm 1: Transformation of a directed graph into a tree for deriving role-based test paths. The resulting tree is similar to the block-cut trees (Hopcroft and Tarjan, 1973).

```
Input: s as entry state
Output: allPaths as list of unique paths for a
        specific user role
initialize role as specific user role;
initialize allPaths as list;
initialize path as list;
initialize root ← s;
while s has unvisited transition for role do
    add s to path;
    initialize next ← next unvisited state;
    if path contains next or next has no
     unvisited transition for role then
        mark transition between s and next as
         visited;
        foreach state in reversed path do
            if state has no unvisited transition
             for role then
                mark transition between state
                 and state's parent node as
                 visited;
            end
        end
        add next to path;
        add path to allPaths;
        path ← empty;
        s ← root;
    end
    else
        s ← next;
    end
end
```

4.2 Generation of Test Data

Our generated server has to be populated it with test data. Otherwise a client would not be able to obtain or modify any resource while navigating through the application workflow. Our MDSD approach allows automated test data generation based on our existing metamodel. This means, once written, this process will start to generate adequate test data by looking for available resources within the input model.

4.3 Building HTTP Client

For our hypermedia testing purpose, we need to build our own hypermedia-driven HTTP client. First, the client represents a specific user role while testing, so it makes sense to save this authentication information. Then, it must be able to send request to relation types instead of sending request directly to URIs. By definition, a hyperlink can be returned within the response header or body or both. For the sake of simplicity, this approach only addresses hyperlinks within the response header. A header response link of our RESTful API consists of four parts: resource URI, `rel` as relation type, `type` for media type and `method` for HTTP verb as listed in Listing 1. These elements need to be extracted and parsed, so the HTTP client can understand and send request to it. This is necessary because technically speaking an HTTP client requires merely a URI to send request to, but it is not recommended to hard code these so called "REST endpoints" into clients (Fielding, 2013), because they can change, e.g, to resource renaming. So our hypermedia response is always provided with a fixed relation type serving as a method call to guide action and of which the client has knowledge.

```
<http://api/item>;rel="createItem";
type="application/json",method="post">
```

Listing 1: Hypermedia link in HTTP response header.

4.4 Generation of Test Classes

The actual model is encapsulated behind the generation process, and therefore in order to run these test cases out of the box, we have to combine the retrieved data from the model crawling process with the functionalities of the HTTP client and embed them in generated test classes. This can be done by a generator producing test class files as shown in Figure 4. There are three crucial elements that must be persisted in a test class for an HTTP client to be able to run the test cases: 1) List of paths that represents given positive test cases, 2) mappings between relation type

and correct outgoing relation types which allow the HTTP client to verify response after each request and 3) mappings between relation type and unpermitted relation types which allow the HTTP client to perform negative testing. This information will be generated and initialized as hard-coded objects in each test class.

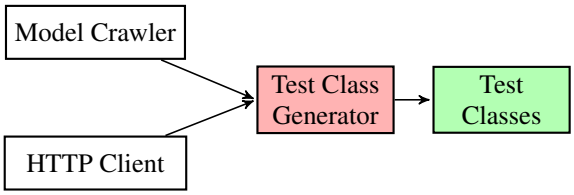

Figure 4: The model crawler and the HTTP client are required by the test class generator to create test classes.

5 CONCLUSION

The main goal of this paper is to automate integration testing with the focus on hypermedia testing using MDT.

To address RQ 1, we propose a model crawling process to extract information from the model to build appropriate test cases. These test cases must consider role-based access of application states, hypermedia response validation and negative testing. In order to retrieve information for role-based access of application states, we have developed an algorithm to derive all possible test paths for each user role. According to our assumption every task within a hypermedia system must be reachable from the entry state, and therefore, a path must represent a distinct task. Our algorithm is able to embody an user role to navigate through the application workflow, visiting every transition and application state. As a result, it delivers a set of distinct test paths for each user role. Validation of application state responses requires mapping information between relation type and role-based follow-up relation types. We have achieved these mappings by letting the model crawler loop through every application state and map all incoming transitions of an application state with permitted outgoing transitions for each user role. Our model crawler also managed to retrieve information for negative testing by mapping relation type to unpermitted relation types. Our model also allows a straightforward approach towards test data generation. This was achieved by generating test data based on meta information of the resources given by the input model.

Approaching RQ 2, we first build an HTTP client to verify hypermedia test cases. This HTTP client must be able to authenticate as a pre-defined user role, send request to relation types and understand hypermedia responses. We made use of an open source Java HTTP client named OkHttp (Square, 2017b) and added necessary features. These objectives were accomplished with no further complication. Afterwards, we combine both information of role-based test cases retrieved by the model crawler and functionalities of the HTTP client to generate ready-to-run test classes. For this purpose, we used an open source library named JavaPoet (Square, 2017a) to handle source code generation. As a result, we successfully generated role-based test classes, including test paths, validation mappings, negative testing, HTTP crawler and all required imports.

In order to answer RQ 3, we must take several aspects of integration testing into consideration: By using the underlying model, we were able to generate test classes to cover all possible tasks within the example application workflow, assuring white-box hypermedia testing of the overall system. This would be a time consuming task if implemented manually because a) deriving all distinct role-based test paths without an algorithm would be inconceivable, b) role-based hypermedia testing is a repetitive task because a developer has to write many similar test cases and c) negative testing also requires enormous amount of time when unpermitted actions have to be verified for every application state. So far, we managed to generate positive test cases, by making valid requests with valid data as inputs and checking whether the application response as expected. These results are quite satisfactory as our generated test cases could cover all possible tasks within a application and verify role-based responses of every application state. Nevertheless, the type of negative testing we were able to accomplish within the scope of this work was sending unauthorized requests by the HTTP client to the server. There are many other possibilities to generate different types of negative testing, such as applying not allowed methods, trying to access a (sub-)resource after deleting it or forcing the HTTP client to send wrong resource representations. We can extend the model crawler to extract more possible test cases to achieve larger test coverage. These features only need to be implemented once and test cases will be generated for free. We can strive to reveal more bugs or as Dijkstra states in his article (Dijkstra, 1972): Testing can only reveal the presence of bugs, but not their absence.

Additionally, we should discuss more about how the application should behave in case of invalid requests. As for the scope of this work, we only expect to see an appropriate response code. We have also discussed a bit about whether the server should

send a client back to its previous state. However, this approach would violate the stateless constraint. Always sending a client back to the start state would, on the other hand, reduce the usability of the application, forcing the user to repeat many unnecessary steps again, especially when the intended task is nested deep within the application workflow. Further research would be needed to clarify this matter.

REFERENCES

Amundsen, M. (2017). *RESTful Web Clients - Enabling Reuse Through Hypermedia*. O'Reilly Media, Sebastopol.

Bloch, J. (2014). How to design a good API and why it matters. http://static.googleusercontent.com/media/ research.google.com/en//pubs/archive/32713.pdf. Last accessed on May 23, 2018.

Chakrabarti, S. and Kumar, P. (2009). 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.

Dijkstra, E. W. (1972). The humble programmer. *Commun. ACM*, 15(10):859–866.

Fertig, T. and Braun, P. (2015). Model-driven Testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web Companion*, WWW '15 Companion, pages 1497–1502, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

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

Fielding, R. (2008). REST APIs must be hyper-text driven. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. Last accessed on May 23, 2018.

Fielding, R. T. (2013). WTF is a "REST endpoint". https://twitter.com/fielding/status/324448353180061696. Last accessed on May 22, 2018.

Frankel, N. (2015). *Integration Testing from the Trenches*. Leanpub.

González, C. A. and Cabot, J. (2014). Test data generation for model transformations combining partition and constraint analysis. In Di Ruscio, D. and Varró, D., editors, *Theory and Practice of Model Transformations*, pages 25–41, Cham. Springer International Publishing.

Hernández, A. G. and García, M. N. M. (2010). A Formal Definition of RESTful Semantic Web Services. In *Proceedings of the First International Workshop on RESTful Design*, WS-REST '10, pages 39–45, New York, NY, USA. ACM.

Hopcroft, J. and Tarjan, R. (1973). Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378.

Liggesmeyer, P. (2009). *Software-Qualität - Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, Berlin Heidelberg, 2. aufl. edition.

Project, T. S. (2018). Selenium. https://www.seleniumhq.org/. Last accessed on May 24, 2018.

Richardson, L. (2009). The Maturity Heuristic. https:// www.crummy.com/writing/speaking/2008-QCon/act3.html. Last accessed on May 16, 2018.

Richardson, L., Amundsen, M., and Ruby, S. (2013). *RESTful Web APIs*. O'Reilly Media.

Schreibmann, V. and Braun, P. (2015). Model-Driven Development of RESTful APIs. In *Proceedings of the 11th International Conference of Web Information Systems and Technologies*. INSTICC, SciTePress.

Square, I. (2017a). JavaPoet. https://github.com/square/javapoet. Last accessed on May 23, 2018.

Square, I. (2017b). OkHttp. http://square.github.io/okhttp/. Last accessed on May 23, 2018.

Taivalsaari, A. and Mikkonen, T. (2017). The Web as a Software Platform: Ten Years Later. In *Proceedings of the 13th International Conference of Web Information Systems and Technologies*. INSTICC, SciTePress.

Vu, H., Fertig, T., and Braun, P. (2017). Towards model-driven hypermedia testing for RESTful systems. In *WEBIST 2017 - Proceedings of the 13th International Conference on Web Information Systems and Technologies*.

Vu, H., Fertig, T., and Braun, P. (2018). Verification of hypermedia characteristic of restful finite-state machines. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 1881–1886, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

Webber, J., Parastatidis, S., and Robinson, I. (2010). *REST in Practice - Hypermedia and Systems Architecture*. "O'Reilly Media, Inc.", Sebastopol.

Wright, D. (2005). Finite State Machines. http://www4.ncsu.edu/drwrigh3/docs/courses/csc216/fsm-notes.pdf. Last accessed on May 16, 2018.

Zuzak, I., Budiselic, I., and Delac, G. (2011). *Web Engineering: 11th International Conference, ICWE 2011, Paphos, Cyprus, June 20-24, 2011*, chapter Formal Modeling of RESTful Systems Using Finite-State Machines, pages 346–360. Springer Berlin Heidelberg.