Towards Model-driven Hypermedia Testing for RESTful Systems

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

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

Abstract: Testing RESTful systems is a missing topic within literature. Especially hypermedia testing is not mentioned at all. We discuss the challenges of hypermedia testing that were discovered within our research. We will differ between client-side and server-side challenges since REpresentational State Transfer (REST) describes a client-server system. Therefore, both sides have to be considered. Hypermedia tests for the server have to ensure that there is no response without hypermedia links. However, the client also has to be hypermedia compliant. Thus, we propose to simulate a server update to check whether the client breaks. Since we use Model-driven Software Development (MDSD) to generate RESTful systems we also propose a model-driven approach for hypermedia testing. This allows us to generate tests for a server based on its underlying model. Moreover, we can build a crawler to verify our generated servers and to test all hypermedia links for different user roles. Any modification to the model can result in a server update, which can be used to test hypermedia clients.

1 INTRODUCTION

In 2015 we presented our project *Generating Mobile Application with RESTful Architecture* (GeMARA) (Schreibmann and Braun, 2015) in which we proposed a model-driven approach for creating RESTful APIs. As this project matures, we also explored the possibility of Model-driven Testing (MDT) (Fertig and Braun, 2015) and realized the lack of information about quality assurance for RESTful Systems.

However, the situation has not changed. Information about quality assurance of hypermedia systems is still missing. Therefore, we propose different challenges of hypermedia testing and possible approaches to overcome them. Since REpresentational State Transfer (REST) is an architectural style for client-server applications, it is important to cover both: the server-side as well as the client-side. Manual hypermedia testing is time-consuming and hard to maintain. Testing hypermedia APIs requires many similar structure like test cases, especially when different user roles and error cases are considered. Therefore, MDT is a helpful tool.

Due to our Model-driven Software Development (MDSD) approach we have an existing meta model that can be used for MDT. We already support functional testing via MDT (Fertig and Braun, 2015) and thus, we want to extend our approach to also support hypermedia testing. Until this point, our project (GeMARA) allows us to generate fully functional RESTful systems with functional test cases.

Some may argue that testing generated code does not make sense. Their main argument is that if the generators are correct, the produced code will also be correct. However, MDT is a possible way to achieve correctness within generators. The test cases will detect possible bugs within the generators that can then be fixed. Moreover, test cases can verify third-party components used by our platform code within the MDSD approach. Third-party libraries or frameworks can change over time. This requires much effort to check all dependencies manually. The generated tests can detect wrong behavior of the system after dependency updates. Finally, performance testing is an additional reason why testing generated code does make sense.

First, we will summarize related work and prove that there is a lack of information about testing hypermedia. Afterwards, we will discuss challenges for testing both: the server-side and the client-side. Furthermore, we will propose our approaches to solve these challenges. Finally, we will give a short outlook and discuss our future work.

Vu, H., Fertig, T. and Braun, P.
Towards Model-driven Hypermedia Testing for RESTful Systems.
DOI: 10.5220/0006353403400343
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 340-343
ISBN: 978-989-758-246-2

2 RELATED WORK

Fielding criticizes in his blog (Fielding, 2008) that many existing APIs are called RESTful even though they do not adhere to the hypermedia constraint. Furthermore, development of RESTful APIs is difficult due to a lack of software frameworks which guide their implementation (Vinoski, 2008). This circumstance leads to a widely-held misconception of RESTful API design among the developer community, subsequently there is a lack of existing RESTful APIs that adhere to the hypermedia constraint. Since hypermedia is not present there is no awareness for testing it. The common procedure of REST API testing simply consists of sending an HTTP-request and verifying an expected response (Webber et al., 2010). To the best of our knowledge, there is a lack of information about generating or testing hypermedia systems.

RESTful API Modeling Language (RAML) (Hevery et al., 2017) is also pushing the idea of MDSD. They offer a formal model based on their Domain-Specific Language (DSL) for defining RESTful APIs. The DSL is designed to describe the full API life cycle in a human readable format, which incorporates many RESTful specifications, such as URIs, authorizations, namespaces, media types and HTTP verbs. Based on their model a fully functional RESTful API can be generated and tested. However, their model does not support the hypermedia aspect, such as navigating the client through the application via hyperlinks. This is why their approach violates Fielding's hypermedia constraint.

Choi, Necula and Sen propose a method to transform manual testing into automated testing. (Choi et al., 2013). The user has to walk through the application manually in order to train their machine learning algorithm. Once the machine learning algorithm has a slight representation of the application model, it starts to generate user inputs to visit unexplored states of the application. The key feature of this algorithm is to avoid restarting the application. Nevertheless, this approach is a black-box testing method, since a tester and later-on the machine learning algorithm have to navigate exploratively through an application without any knowledge of the underlying model. Since our model-driven approach allows white-box testing, this approach does not meet our requirements.

3 SERVER-SIDE TESTING

According to Fielding (Fielding, 2008) RESTful systems must be hypertext-driven, in other words these systems are to be designed as Finite-State Machines (FSMs). In (Zuzak et al., 2011) and (Hernández and García, 2010) the authors also present their formal models for specifying RESTful APIs as FSMs based on their understanding. Our main goal for the server-side testing is to ensure that any RESTful API that is generated by our model (GeMARA) is a FSM. Our model has one dispatcher state and multiple application states. The dispatcher state represents the initial state of the FSM. Every application state is defined by an HTTP verb and a resource. A client can navigate from one application state to another via transitions. A transition is defined by an hyperlink, a mediatype and a rel type. A rel type is akin to the rel attribute of HTML link tags. The first challenge is to check whether our generated RESTful API is FSM-compliant. This allows a client to start from the dispatcher state, visit every application state and go back to the dispatcher state without getting stuck in a dead-end. Figure 1 shows an example FSM with possible states for the user resource. For the sake of clarity we did not illustrate transitions from states back to the dispatcher state.

Responses of application states must contain a finite set of hyperlinks through which the user or automaton can obtain choices and select actions (Fielding, 2008). Another challenge of server-side testing is to check whether the generated RESTful API is delivering appropriate hyperlinks based on the client's active user role. Assuming our application is an ecommerce platform: A customer role is allowed to view items in a shop, whereas a shop admin role can create, update or delete items in his shop. Each role is only permitted to see its designated hyperlinks to navigate through the application, otherwise the authorization concept would be corrupt. This would require enormous effort if implemented manually for a system with a dozen roles and hundreds of application states. However, we can already generate test cases based on user roles and authorization headers (Fertig and Braun, 2015), for this purpose we only have to extend our previous generators.

Since our model already provides concrete information about every application state and its possible transitions, we could perform a static analysis as a first step to check whether the model is FSM-compliant. We check whether every application state has at least one incoming and one outgoing transition. This ensures that every state is reachable and there are no dead-ends. The goal of the static analysis is to identify any error at the highest level of abstraction before triggering any source code generation. At the same time the static analysis would make the API designer aware of any missing transition in his model. For example the dead-end of the application state

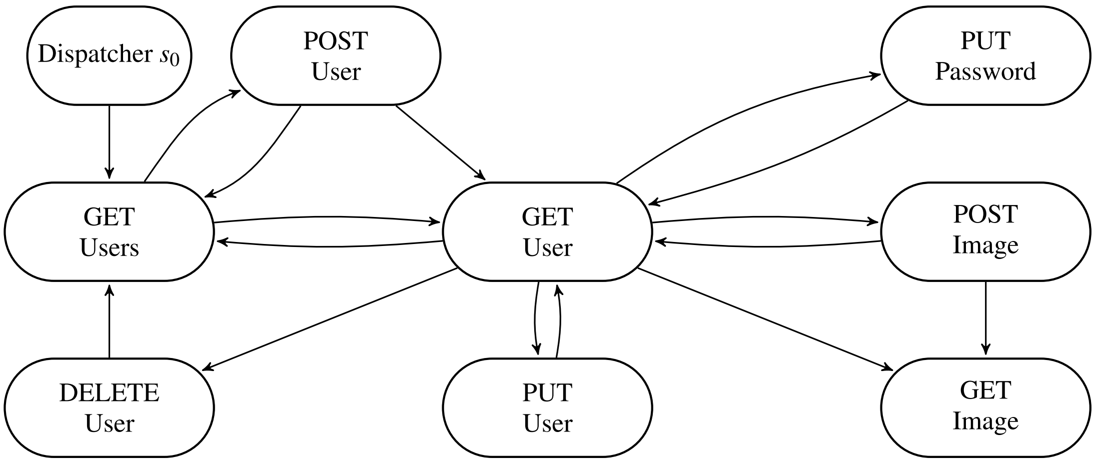

Figure 1: The finite-state machine for our example user resource. Error cases are not visualized.

GET Image in Figure 1 would be detected.

Once the static analysis is successfully carried out, the system will generate a functional server. Following this, we continue with a dynamical analysis with an HTTP-crawler. The goal of this step is to guarantee the correct functionality of the generated API. In addition we also have to ensure that whenever the request leads to an error the client will be redirected to its previous state or at least to the dispatcher state. Moreover, the HTTP-crawler will verify the behavior for different user roles. The HTTP-crawler will pass itself off as one of predefined user roles and follows any given hyperlink and expects only hyperlinks that are designated to its current user role.

The dynamical analysis is considered successful when every user role is able to travel and perform all its permissions by generating at least one proper request for each permission. This again can be verified by the underlying model. Furthermore any undesired behavior can reveal bugs within the generators.

4 CLIENT-SIDE TESTING

Real-world businesses change and so must their underlying systems (Josuttis, 2007). Fielding describes evolvability as the degree to which a component can be changed without negatively impacting other components (Fielding, 2000). Our server is a component that will evolve through time which means it is exposed to updates and changes depending on its real-world needs. These changes will affect its clients and thus, client-side testing has to ensure that clients are hypermedia-driven. Hypermedia clients should still work after a server update. Therefore, clients are not allowed to make any assumption about URIs (Prescod, 2002). As a result, hypermedia clients are more robust, adaptable and resilient to server updates which will reduce the need to versioning and repeated redeployment (Amundsen, 2015).

APIs can be consumed by third-party clients. Clients that make proper use of hypermedia would require less manual adaption to server updates than those that do not. Our motivation is to find out whether a client is hypermedia-driven. If a running client can handle a simulated server update, then we assume this client as hypermedia-driven.

In the following we will discuss a few example changes that have to be considered in order to simulate a server update:

First, we take a look at changes concerning URIs. In case of updating or deleting URIs the client should still work because hyperlinks are not hard coded. If new functionalities are added, the client should be able to work with old functionalities. We assume that the client cannot deal with new rel types. However, Amundsen presents another approach for this issue in his talk (Amundsen, 2015). He proposes that servers should emit templates in order to enable clients to deal with new functionalities.

Secondly, changes concerning rel types should be considered. The client has knowledge about rel types within our approach, therefore, renaming rel types is not supported without client update. However other approaches may support renaming (Amundsen, 2015). Clients can not deal with new rel types but should be able to work with old functionalities. Deleting rel types causes no problem to the client since hyperlinks, which contain rel types, are not hard coded.

Lastly, resource representations could change due to server updates. There is no general solution to this issue. Hypermedia clients following Amundsen's approach (Amundsen, 2015) can easily adapt to changes within resource representations due to the emitted templates. However, other approaches may need additional investigation.

The manual effort to simulate a server update would not be worthwhile. Nevertheless, our model-driven approach allows us to generate test servers without additional effort. The first step of our current workflow is to define a RESTful API model. After that our generators use this model to generate a working server. To simulate a server update we take the same model and apply server update changes to it. For this purpose, we need to build a generator that can apply random changes to an existing model for example adding, updating and removing URIs. Afterwards, the modified model will be passed through the generator again. As a result, the generator will produce a fully functional updated test server. If the client does not break after redirecting it to the test server, we assume the client as hypermedia-driven.

5 CONCLUSION

Our vision is to generate robust RESTful APIs along with tests to assure Fielding's hypermedia constraint. Once the proposed steps above are carried out, we will gain deeper understanding of generating and testing hypermedia systems.

This knowledge will be the foundation for us to start working on test generation at a higher level of abstraction. For example, we can generate tests based on user acceptance criteria. User Acceptance Test (UAT) is an expensive and time-consuming task (Hambling and van Goethem, 2013). Our crawler can generate sequences of inputs to imitate the behavior of a user to perform general use case scenarios. This would reduce the amount of work significantly.

Based on the client-side testing we can derive testing guidelines for hypermedia clients. Any front-end developer can follow these guidelines to implement robust hypermedia clients.

REFERENCES

Amundsen, M. (2015). Learning Client Hypermedia from the Ground Up. http://amundsen.com/talks/2015-06-ndcoslo/2015-06-ndcoslo-slides.pdf. Last accessed on Mar 17, 2017.

Choi, W., Necula, G., and Sen, K. (2013). Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 623–640, New York, NY, USA. ACM.

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 Mar 17, 2017.

Hambling, B. and van Goethem, P. (2013). *User Acceptance Testing: A Step-by-step Guide*. BCS Learning & Development Limited.

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.

Hevery, M., Musser, J., Rexer, P., Sarid, U., and Lazarov, I. (2017). RAML. http://raml.org/. Last accessed on Mar 17, 2017.

Josuttis, N. M. (2007). *SOA in Practice - The Art of Distributed System Design*. "O'Reilly Media, Inc.", Sebastopol, 1. aufl. edition.

Prescod, P. (2002). REST and the Real World. http://www.xml.com/pub/a/ws/2002/02/20/rest.html. Last accessed on Mar 17, 2017.

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*, pages 5–14. INSTICC, SciTePress.

Vinoski, S. (2008). RESTful Web Services Development Checklist. *IEEE Internet Computing*, 12(6):96–95.

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

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.