Verification of Hypermedia Characteristic of RESTful Finite-State Machines

Henry Vu
University of Applied Sciences
Würzburg-Schweinfurt
Würzburg
henry.vu@fhws.de

Tobias Fertig
University of Applied Sciences
Würzburg-Schweinfurt
Würzburg
tobias.fertig@fhws.de

Peter Braun
University of Applied Sciences
Würzburg-Schweinfurt
Würzburg
peter.braun@fhws.de

ABSTRACT

Being an architectural style rather than a specification or a standard, the proper design of REpresentational State Transfer (REST) APIs is not trivial, since 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 and especially, hypermedia testing is not mentioned at all. To deal with this state of affairs, we have elaborated a Model-Driven Software Development (MDSD) approach for creating RESTful APIs. As this project matured, we also explored the possibility of Model-Driven Testing (MDT). This work addresses the challenges of hypermedia testing and proposes approaches to overcome them with MDT techniques. We present the results of hypermedia testing for RESTful APIs using a model verification approach that were discovered within our research. MDT enables the verification of the underlying model of a RESTful API and ensuring its correctness before initiating any code generation. Therefore, we can prevent a poorly designed model from being transformed into a poorly designed RESTful API.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering; Software verification and validation; Layered systems; Client-server architectures; Formal software verification;**

KEYWORDS

REST, RESTful Applications, RESTful Systems, Hypermedia, Hypermedia Testing, MDSD, MDT

ACM Reference Format:

Henry Vu, Tobias Fertig, and Peter Braun. 2018. Verification of Hypermedia Characteristic of RESTful Finite-State Machines. In *WWW '18 Companion: The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3184558.3191656>

1 INTRODUCTION

In the past years, the Web has become the de facto deployment environment for new software systems and applications. The Software as a Service (SaaS) Model has redefined Personal Computing.

This paper is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.

WWW '18 Companion, April 23–27, 2018, Lyon, France

© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.

ACM ISBN 978-1-4503-5640-4/18/04.

<https://doi.org/10.1145/3184558.3191656>

Nowadays, we seldom have to install new software on our PCs to consume services. Office productivity applications and corporate tools such as invoicing, purchasing and expense reporting systems have migrated to the Web. Banking, insurance and retail industries, to name a few, have been transformed profoundly by the emergence of web-based applications and Internet services [14]. Web APIs have become the vital backbones for these applications and services. While the number of Web APIs is increasing [1], the need for good API design has become more crucial than ever before. On the one hand, good APIs can be among a company's greatest assets, as customers invest heavily in buying, writing and learning them, but on the other hand, bad APIs can also be among a company's greatest liabilities as they result in never-ending streams of maintenance and support [4].

In 2000, Fielding [7] presented an architectural design approach for building network-based applications through principled use of architectural constraints called REpresentational State Transfer (REST) in his dissertation. In its essential form, REST requires that a server must adhere to certain constraints such as client-server, stateless, cache, uniform interface, layered system and hypermedia. Ever since RESTful APIs have become popular due to its promise of building lightweight stateless distributed services using simple Uniform Resource Identifiers (URIs) and Hypertext Transfer Protocol (HTTP) verbs to make web services broadly available for multiple front-end devices [5]. But those who claim to build RESTful APIs often neglect the hypermedia constraint. Fielding also criticizes in his blog [8] that many existing APIs are called RESTful even though they do not adhere to the hypermedia constraint. Furthermore, the development of RESTful APIs is difficult due to the lack of software frameworks to guide the implementation [16]. These circumstances lead 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 HTTP-requests and verifying the received responses [18].

An API defined as RESTful is fully matured when making use of hypermedia in order to model the relationship between resources. As put simply by [11] that hypermedia drives systems to transform their application state, or in other words as Fielding wrote in his blog [8], that responses of application states must contain a finite set of hyperlinks through which the user or automaton can obtain choices and select actions. He even emphasizes that if the engine of application state, and hence the API, is not being driven by hypertext, then it cannot be RESTful and therefore cannot be called a REST API. The term Hypermedia As The Engine Of Application

State (HATEOAS) exists to describe this constraint. Responses of these so called RESTful APIs do not provide any hyperlinks to navigate the client through an application workflow. This results in two major problems: a) Application states do not generate any follow up hyperlinks and clients are forced to construct these hyperlinks piece by piece which requires prior knowledge of all API endpoints, thus revealing the implementation detail 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. c) Besides the lack of a workflow, this described 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 endpoint must remain the same [15].

To deal with these problems we started our project *Generating Mobile Applications with RESTful Architecture* (GeMARA) in 2015 [13]. We proposed a Model-driven Software Development (MDSD) approach to take RESTful API development to a higher level of abstraction by using meta-models as input to generate RESTful APIs. We propose our own Domain Specific Language (DSL) to describe a model which is then to be translated into a RESTful API. Our DSL is capable of defining typical RESTful API components, for instance, resources with attributes and multiple application states described by a resource and an HTTP verb. Furthermore, we can transition between application states using hyperlinks and an application state can be protected from unauthorized access by adding specific grant feature to deal with authorization e.g. authentication (HTTP Basic or OAuth) and role-based access control. This way, we can enforce consistency and achieve a higher standard of quality by encapsulating reliable and well-known libraries, frameworks and RESTful best practices behind our DSL. This approach significantly accelerates the development process of RESTful APIs. A complete model for an RESTful API with two resources consists of only about a hundred lines of code. Once written, our model can be passed through generators to be transformed into ready-to-deploy source code which declares a fully functional RESTful API. This would require much more effort if implemented manually. Other than that, developers do not have to worry about implementation details such as: What should be the choice of programming language, frameworks, libraries, database or how to accomplish hyperlinks between application states? As this project matured, we also explored the possibility of Model-driven Testing [6] and achieved automated generation of functional test cases for our generated API. In 2017, we proposed several approaches towards hypermedia testing for RESTful systems [17] regarding both: server-side and client-side. This work is a further development of these ideas.

Nevertheless, using a meta-model to describe a RESTful API does not prevent the API designer from making mistakes, for example, wrong naming of resources or missing transitions for some application states and thus violating the hypermedia constraint. Each application state must have at least one incoming transition for a client to be able to reach it and each application state must have at least one outgoing transition, so the client does not get trapped in a dead-end. Consequently, a trapped client has to restart the application.

So, the question may arise of whether verifying hypermedia characteristics of this meta-model makes sense. And if yes, how can we distinguish hypermedia characteristics of the meta-model as such

to test? The goal of this model's verification process is to identify any error at the highest level of abstraction before triggering any source code transformation.

2 RELATED WORK

According to Fielding [8] RESTful systems must be hypertext-driven, in other words these systems are to be designed as Finite-State Machines (ϵ -NFA). In [20] and [9] the authors also present their ϵ -NFAs formal models for specifying RESTful APIs based on their understanding. The established literature concerning REST such as [12], [3] and [18] reveal little to no information about its quality assurance. Moreover, hypermedia testing is not mentioned at all. RESTful API testing by sending HTTP-requests and verifying the received responses was mentioned in [18].

RESTful API Modeling Language (RAML) [10] is also pushing the idea of MDSD for RESTful APIs by offering a formal model based on their DSL. The DSL is designed to describe the complete 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 this meta-model a fully functional RESTful API can be generated and tested. However, this model does not support the hypermedia aspect which enables navigating the client through the application via hyperlinks. This is why this approach violates Fielding's hypermedia constraint.

To the best of our knowledge, there is limited information about generating and testing hypermedia systems. We make use of the given MDSD approach within our GeMARA project to explore the possibility of Model-driven Testing with focus on hypermedia testing for RESTful APIs.

3 GENERATING MOBILE APPLICATIONS WITH RESTFUL ARCHITECTURE

We introduced the project *Generating Mobile Applications with RESTful Architecture* (GeMARA) in 2015 with the vision to generate robust RESTful APIs with test cases to assure Fielding's constraints [13]. The goal of our research project is to develop automated generators for distributed mobile applications based on RESTful architecture. Accordingly, applications are no longer programed but described as an abstract model written in our own DSL and from which source code and other artifacts can be generated.

Building Internet-scaled distributed hypermedia systems using REST requires some deep knowledge, particularly in the design phase. REST has become a major paradigm, but unfortunately it means different things to different people. Some call it a standard, others call it a specification, while REST purists and its creator Fielding consider it as an architectural style. So, in order to fix these misconceptions, we have carefully analyzed Fielding's dissertation [7] to derive REST key components for our model from our experience in implementing several RESTful APIs over the last years [13]. The focus lies in the development of a meta-model that has the ability to describe key components of REST and to form their relationships via hypermedia in a convenient way while considering all other REST constraints.

The most important component in our meta-model to describe RESTful API is the application state. An application state is a pair of one HTTP method and one resource. It represents a valid REST

request to access a resource [33]. Figure 1 shows a simplified class diagram that expresses our application state concept. Also one of the central key elements in a RESTful system is the resource. It is important to note that resource is not a storage object but is, instead, a conceptual entity [2]. A resource represents a single object or a collection of objects. The intention of HTTP verbs should be fixed, and developers should not be free to choose wrong verbs. The four basic operations to create, read, update and delete (CRUD) resources are mapped to the four HTTP verbs POST, GET, PUT and DELETE. This mapping is unambiguous regarding to the HTTP specification [19]. A transition is our formal way of modeling relationships between application states. A client can navigate from one application state to another via transitions. Technically speaking, a transition is comprised by a hyperlink, a media type and a relation type. A relation type is akin to the rel attribute of HTML link tags. Our application state can be protected from unauthorized access by adding specific grant feature to deal with authorization e.g., role-based access control and authentication with HTTP Basic or OAuth [13].

Figure 1: Simplified UML class diagram for application state from [13].

4 HYPERMEDIA CHARACTERISTICS

Since this work is dedicated to Fielding’s hypermedia constraint and how to use a model-driven approach to ensure its presence, we will discuss further details regarding its formal definition as ϵ -NFA.

Our first challenge is to determine whether our meta-model is a ϵ -NFA or not. As mentioned before, a RESTful API benefits from the hypermedia constraint as it introduces an application workflow, making the API’s design more convenient. By lacking this feature, an API is only static, hence forcing the client-side to hard-code every API call in the right order. The concept of ϵ -NFA revolves around a finite number of states and possible state-to-state transitions. Speaking in REST term, a client can only be in exactly one state at any given time and it can only change its current state by navigating through a directed transition. ϵ -NFA-compliance means that every state within an application is accessible.

In order to simplify understanding, it is necessary to present an application example which will be the basis to demonstrate our further approaches. Let’s assume, we want to build an e-commerce

platform - an online shop that sells items. A user can create, update or delete items. We will explain the resources of this application with the aid of a UML class diagram in Figure 2. The user resource is kept simple, it only has one attribute `userName` of data type `String` and one attribute `password` of data type `Password` for authentication purpose. The shop resource has a `name` attribute and a `website` attribute, both of data type `String` and a list of items which are subresources. The item subresource is made of `name` (`String`) and a `price` (`Float`). This example represents a typical web application within the e-commerce domain. For our hypermedia testing purposes we design this application as a model with all must-have features of a common RESTful API such as primary and subresources, application states, transitions. Therefore, we will refer our later solution approaches to this example.

Figure 2: UML class diagram of the application example.

Our main goal for the model’s verification process is to ensure that every generated RESTful API by our model (GeMARA) is a ϵ -NFA. Our model has one dispatcher state and multiple application states. The dispatcher state represents the initial state of the ϵ -NFA. Every application state is defined by an HTTP verb and a resource. Clients can navigate from one application state to another via transitions. A transition is defined by a hyperlink, a mediatype and a relation type. The first challenge is to check whether our model is ϵ -NFA-compliant. The client should be able 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 3 shows an example ϵ -NFA with possible states for our application example. For the sake of clarity, we omitted transitions from every state back to the dispatcher state and self-pointing transitions. After a check, there are two problems within this example. First, there is a missing outgoing transition from the DELETE Shop application state. This would cause a client to be stuck after selecting and deleting a shop. Second, a client cannot reach the PUT Shop application state because there are no transitions leading to it.

Figure 3: ϵ -NFA example of an API with missing transitions.

The second goal of the model testing is to perform a hypermedia semantic check. In a hypermedia system there are certain state-to-state transitions that are considered bad design, and therefore should be avoided. For instance, a DELETE state should not lead to a PUT Single Resource state because this constellation does not make sense from the API designer's point of view. Which specific single resource should the client be navigated to after deleting a single resource? Instead, it would be better to lead the client to a GET collection state of the resource. To avoid such bad design decisions, we perform hypermedia semantic check after the model verification process. This process is able to search for such inappropriate constellations within our model and inform the API designer.

5 VERIFICATION OF HYPERMEDIA CHARACTERISTICS

The model of our MDS approach is described by a textual Domain Specific Language (DSL). This model also embodies the construction plan of the API which means it can be peer reviewed by other experts to check its correctness. Although this verification process can be carried out manually, one should not do this without the aid of existing tools. Since our model already provides concrete information about every application state and its possible transitions, we are able to perform an automatic verification process.

First of all, we check whether our model is ϵ -NFA-compliant. ϵ -NFA-compliance means that every state within an application is accessible and there is no dead state for a client to get trapped in. This test is to be executed directly before the transformation process. The algorithm is straight forward: A loop checks every application state within the model. Whenever there is a missing incoming or outgoing transition, it would throw an appropriate exception to make the API designer aware of this. The transformation process can start if the model is correct - meaning every application state has at least one incoming and one outgoing transition. The goal of this ϵ -NFA check is to identify any error at the highest level of abstraction before triggering any source code transformation. This ensures that every state is reachable and there are no dead-ends, thus violating against Fielding's hypermedia constraint. For example if we take a look at Figure 3, the dead-end at the application

state DELETE Shop and the unreachable application state PUT Shop would be detected.

On the implementation level, our meta-model is comprised of Java objects. This model includes all necessary information for the generators to transform it into a RESTful API. The ϵ -NFA check process takes advantage of this and is able to extract all required information from the model to carry out its verification without any additional external libraries.

Now we have to make sure that there are not any inappropriate state-to-state transitions within our meta-model. This should represent a schematic framework for any API designer to avoid bad design decisions. Noticeably more work would be needed to address this topic in order to create more general guidelines of whether a transition is inappropriate and whether a design is considered as good or not. So far, our study has looked at some specific cases.

Our meta-model is capable of describing six different types of state: GET Dispatcher State, GET Single Resource, GET Collection Resource, POST Single Resource, PUT Single Resource and DELETE Single Resource. Since our meta-model characterizes a ϵ -NFA with directed state-to-state transitions, we take a look at every single state type and discuss whether it has inappropriate outgoing transitions:

GET Dispatcher State: This state represents the entry point of the RESTful API. Therefore, it should be linked directly to a GET Single Resource, a GET Collection Resource or a POST Single Resource. Any other configuration would make little sense since a PUT state or a DELETE state requires a client to preselect an existing resource to modify or to delete it.

GET Single Resource: This state can be linked directly to any other state type because there are no inappropriate outgoing transitions from this state. From a GET Single Resource we can modify it, delete it or go to another Resource depending on the intended application workflow.

GET Collection Resource: This state should not be directly linked to a DELETE or a PUT state. A client located in this application state obtains a set of entities of a specific resource. A DELETE or a PUT state would assume the client wants to manipulate this whole set at once. But there are specific use cases that demand this kind of application workflow.

POST Single Resource: After creating an entity, a client should not be directed to a DELETE state because it would be uncommon to delete an entity right after creating it.

PUT Single Resource: A PUT state should lead directly back to the entity it modified, hence a GET Single Resource state. An outgoing transition from a PUT state to a DELETE state should be avoided due to reason mentioned above.

DELETE Single Resource: After deleting an entity, a client should not be directed to a PUT state because a PUT state requires a client to preselect an entity in order to modify it. It would be more convenient to redirect the client to the next single resource or to a collection of the resource.

Figure 4 illustrates an example of an appropriate workflow within our e-commerce application.

Figure 4: Example of an appropriate application workflow.

6 EVALUATION AND DISCUSSION

The main goal of this work is to verify hypermedia characteristics of RESTful ε -NFA using our MDSD approach. The presence of hypermedia is crucial for a RESTful system since it distinguishes between a static API and a RESTful API with pre-defined application workflow via hypermedia links (HATEOAS). Our verification process reveals errors at the highest level of abstraction, guaranteeing that a model is designed as a ε -NFA before triggering source code transformation. Moreover, it informs the API designer of any problem in case of error.

Since our meta-model itself is comprised of Java objects, we can simply access its properties and loop through every application state to make sure that each one has at least an outgoing and an incoming transition. Each time an API designer triggers code transformation of his model, the ε -NFA check will be initiated. In case of missing transitions, it will provide an appropriate exception, letting the API designer know which application state is problematic and whether it is missing an incoming, or an outgoing transition, or both. If there is no transition missing, the code transformation can start. The implementation for this approach works flawlessly with no further complication, consequently the goal set is achieved.

Furthermore, we also introduced hypermedia semantic check which provides a guideline for designing RESTful APIs. This is a set of rules that outlines inappropriate state-to-state transitions. Another purpose of this work is to educate API designers to avoid bad design decisions, thus making an application workflow more convenient. These rules can be extended as our meta-model of GeMARA advances. As for now, this hypermedia semantic check takes place right after the ε -NFA check and before the code transformation process. It produces a warning whenever one of these rules is violated.

However, our findings concerning model-driven model testing prove to have limitations. We are able to access properties of our model because its format is based on Java objects on the implementation level. But any other model-driven approach that relies on graphical model format or other DSLs may need additional investigation to apply automatic model testing.

7 CONCLUSION AND FUTURE WORK

In this work we pointed out that many APIs are called RESTful even though they do not comply with Fielding's hypermedia constraint.

A RESTful API requires usage of hypermedia to guide its application workflow, else it only meets the requirements of a static server. Despite its importance, this is still a missing topic in literature. To facilitate the development process of RESTful APIs, we started our project GeMARA in 2015, in which we proposed a model-driven approach for creating RESTful APIs. Our vision is to generate robust RESTful APIs along with tests to assure Fielding's hypermedia constraint.

We have ensured that our model is designed as a ε -NFA by executing an automatic hypermedia check. Manual model review is time-consuming and hard to maintain, especially a model with dozens of resources and hundreds of application states. This model verification process is a useful tool because it prevents source code generation from badly designed models and makes API designers aware of their mistakes.

During the implementation of the hypermedia check, we came to realize that our model offers far more possibilities to carry out an extensive model validation. We discovered that we can check for hypermedia semantic which prevents certain state-to-state transitions being set. This topic is worthy of additional research because we must be careful not to overgeneralize. Thus, more time is needed to create more general guidelines of whenever a transition is inappropriate and whenever a model design is considered as good. Furthermore, there are additional tasks that a model analysis is capable of, such as checking for resource, state or transition naming by convention. These model tests can contribute to towards better model design and ultimately better generated RESTful APIs.

To guarantee that the static meta-model is a correct ε -NFA, is not sufficient. We also have to test the generated source code from any ε -NFA-compliant model as mentioned in [17]. Code generators could have bugs, thus transforming a correct model into faulty source code. This means in hypermedia term that even if every application state within a model has at least one incoming and one outgoing transition, the actual generated API from this ε -NFA-compliant model could still have missing transitions at some application states. Responses of application states must contain a finite set of hyperlinks through which the user or automaton can obtain choices and select actions [8]. This means that we have to check whether response of each application state provides a correct finite set of hyperlinks at runtime.

RESTful 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. Once our server-side hypermedia testing process is fully automated, we will address the client-side hypermedia testing. Our motivation on the client-side testing is to find out whether a client is hypermedia-driven or not. This assumption can be confirmed if a running client can handle certain types of change within the server. We will discuss the degree to which a server can be changed without negatively impacting hypermedia clients. In our future works, we will address the practicality of automating server updates for hypermedia client testing.

A long-term goal is taking model-driven software quality assurance to a higher level of abstraction. For example, we could start working on test generation at a higher level of abstraction by generate test cases based on user acceptance criteria. User Acceptance Test is an expensive and time-consuming task. Our crawler can

generate sequences of inputs to imitate the behavior of a user to test general use case scenarios. This would reduce the amount of work significantly.

REFERENCES

- [1] 2017. Growth in Web APIs since 2005. <http://www.programmableweb.com/api-research>. (2017). Last accessed on Jan 23, 2018.
- [2] M. Amundsen. 2017. REST The short version. <http://exyus.com/articles/rest-the-short-version/>. (2017). Last accessed on Jan 23, 2018.
- [3] Mike Amundsen. 2017. *RESTful Web Clients - Enabling Reuse Through Hypermedia*. O'Reilly Media, Sebastopol.
- [4] Joshua Bloch. 2014. How to design a good API and why it matters. <http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf>. (2014). Last accessed on Jan 23, 2018.
- [5] H. Ed-doui, J. L. C. Izquierdo, M. Tisi A. Gâşmez, and J. Cabot. 2016. EMF-REST: Generation of RESTful APIs from Models. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC'2016, SciTePress.
- [6] Tobias Fertig and Peter Braun. 2015. Model-driven Testing of RESTful APIs. In *Proceedings of the 24th International Conference on World Wide Web Companion (WWW '15 Companion)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1497–1502. <https://doi.org/10.1145/2740908.2743045>
- [7] R.T. Fielding. 2000. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine.
- [8] R.T. Fielding. 2008. REST APIs must be hyper-text driven. <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>. (10 2008). Last accessed on Jan 23, 2018.
- [9] Antonio Garrote Hernández and MarĀĵa N. Moreno GarcĀĵa. 2010. A Formal Definition of RESTful Semantic Web Services. In *Proceedings of the First International Workshop on RESTful Design (WS-REST '10)*. ACM, New York, NY, USA, 39–45. <https://doi.org/10.1145/1798354.1798384>
- [10] M. Hevery, J. Musser, P. Rexer, U. Sarid, and I. Lazarov. 2017. RAML. <http://raml.org/>. (2017). Last accessed on Jan 23, 2018.
- [11] Savas Parastatidis, G. S. Jim Webber, and I. S. Robinson. 2010. The Role of Hypermedia in Distributed System Development. In *Proceedings of the First International Workshop on RESTful Design*. WS-REST, ACM.
- [12] L. Richardson, M. Amundsen, and S. Ruby. 2013. *RESTful Web APIs*. O'Reilly Media. <https://books.google.de/books?id=ZXDGAAAQBAJ>
- [13] V. Schreibmann and P. Braun. 2015. Model-Driven Development of RESTful APIs. In *Proceedings of the 11th International Conference of Web Information Systems and Technologies*. INSTICC, SciTePress.
- [14] Antero Taivalsaari and Tommi Mikkonen. 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.
- [15] Stefan Ulrich. 2014. Wer REST will, muss mit HATEOAS ernst machen. <https://jaxenter.de/wer-rest-will-muss-mit-hateoas-ernst-machen-489>. (2014). Last accessed on Jan 23, 2018.
- [16] S. Vinoski. 2008. RESTful Web Services Development Checklist. *IEEE Internet Computing* 12, 6 (Nov 2008), 96–95. <https://doi.org/10.1109/MIC.2008.130>
- [17] H. Vu, T. Fertig, and P. Braun. 2017. Towards model-driven hypermedia testing for RESTful systems. In *WEBIST 2017 - Proceedings of the 13th International Conference on Web Information Systems and Technologies*.
- [18] Jim Webber, Savas Parastatidis, and Ian Robinson. 2010. *REST in Practice - Hypermedia and Systems Architecture*. "O'Reilly Media, Inc.", Sebastopol.
- [19] www.w3.org. 2018. HTTP Specification. <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>. (2018). Last accessed on Jan 22, 2018.
- [20] Ivan Zuzak, Ivan Budiselic, and Goran Delac. 2011. *Web Engineering: 11th International Conference, ICWE 2011, Paphos, Cyprus, June 20-24, 2011*. Springer Berlin Heidelberg, Chapter Formal Modeling of RESTful Systems Using Finite-State Machines, 346–360. https://doi.org/10.1007/978-3-642-22233-7_24