Towards Offline Support for RESTful Systems

Tobias Fertig

University of Applied Science Würzburg-Schweinfurt
Würzburg
tobias.fertig@fhws.de

Peter Braun

University of Applied Science Würzburg-Schweinfurt
Würzburg
peter.braun@fhws.de

ABSTRACT

Representational State Transfer (REST) is an efficient and by now established architectural style for distributed hypermedia systems. However, REST has not been designed for offline operations, yet many applications must also keep functioning when going offline for more than a few seconds. Burdening the programmer with knowledge about offline status is undesirable. RESTful applications can be described by a formal model. Therefore, we define a function to derive a formal model of the proxy for handling offline support on the client-side. We then extend existing caching approaches so that a client-side proxy can transparently hide the offline status from the application. We validate our solution with a proxy layer that covers all test cases derived from the model. Using our model and proxy, clients do not have to know and worry about whether they are online or offline.

CCS CONCEPTS

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

KEYWORDS

REST, RESTful Applications, RESTful Systems, Offline Support, Hypermedia, Offline First, MDSD

ACM Reference Format:

Tobias Fertig and Peter Braun. 2018. Towards Offline Support for RESTful Systems. 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/10.1145/3184558.3191655>

1 INTRODUCTION

Representational State Transfer (REST) was introduced by Fielding within his dissertation [2]. Despite of the popularity of REST many developers ignored several constraints of REST. That is why Fielding's blog post explains that REST APIs must be hyper-text driven in order to fulfill the hypermedia constraint [3]. This constraint requires a stable internet connection to allow the client to communicate with the server uninterrupted. However, stable internet connections are not possible in any place at any time. Could it be possible to improve the user experience by providing offline support on the client side? We encountered several scenarios that would benefit from offline support:

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/10.1145/3184558.3191655>

Sitting in an airplane forces people to work in offline mode. Therefore, any application that does not provide offline support, cannot be used by passengers on airplanes. Cloud Computing also benefits from offline support. Current browsers use offline storage to enable Web or cloud applications working offline. Google suggests the *offline first* approach in their developer documentation (https://developer.chrome.com/apps/offline_apps). Offline first recommends to write an app as if it needs no internet connection. Network features can be added once the app works offline. Nowadays, developers have to implement offline support manually and they are also forced to distinguish between offline and online mode.

Is offline support in RESTful systems an oxymoron? Is there any way to reduce the developer's effort for implementing offline clients? To answer these questions we did some research on related work that will be discussed in the following section. The third section proposes the research questions for this paper, which will then be answered in the following three sections 4, 5 and 6. Afterwards, we show the results by using our model and its implementation. Finally, we will give a short outlook on future work.

2 RELATED WORK

Satyanarayanan proposed Coda, a distributed file system that runs using a client-server model [10]. It distinguishes a small number of trusted servers and a large number of untrusted clients. Coda uses a callback-based cache coherence: the server remembers which objects have been cached by a client. However, Fielding defined the stateless constraint for REST: the server has to be stateless to improve scalability of the overall system. Therefore, Satyanarayanan's approach is not applicable for RESTful systems [2].

Gonçalves and Leitão contributed a lot to enable offline execution in Web applications [5]. They described an offline model that contains a subset of the server logic. The interaction model describes that a local server has to work as a proxy which has to forward all requests if online. In offline mode the proxy has to save the offline request and return an offline response. If the client later goes online, all offline work has to be synchronized back to the server [6]. However, Gonçalves and Leitão focused on prefetching mechanisms to fill the offline cache which we consider to be part of our future work.

Disconnected operations are not exclusive to Web applications. In fact, distributed systems already consider them. Nevertheless, those considerations are not suitable for Web architecture. However, some distributed systems defined similar goals. For example, Demers et al. defined for their Bayou architecture the following goals: Firstly, an offline approach has to support devices with limited resources. Secondly, high availability for read and write operations should be ensured. Thirdly, there has to be a mechanism to support the detection of update conflicts. Finally, an application specific resolution of update conflicts has to be established [1].

Hoodie (<http://hood.ie>) is designed for frontend Web applications and follows the offline first approach (<http://offlinefirst.org>). Hoodie offers a local, user-specific database that is automatically synchronized back to the server. Loopback (<https://loopback.io>) is a Node.js API framework that enables offline synchronization via isomorphic JavaScript. The data replication is implemented as model-based data persistence. The framework was mainly developed to synchronize multiple backends. Therefore, its focus is not on offline support. Furthermore, those frameworks do not concentrate on RESTful systems and they need specific backends to run properly. This is why we tried to define a generic model to add offline support not only to RESTful systems under development but also to existing systems.

3 RESEARCH QUESTIONS

This section lists three research questions that have to be answered within this work and we also describe our research methodology:

- Q1 Are RESTful systems and offline support combinable?
- Q2 How is it possible to organize offline supported functionality hierarchically?
- Q3 Is there a generic solution for offline support in RESTful systems?

Questions Q1 will be answered by literature research in the following section. Afterwards, we will propose a six-level hierarchy to answer Question Q2 in Section 5. Question Q3 will then be answered by using a formal model to describe RESTful systems.

4 OFFLINE SUPPORT AND REST

To answer Q1 we review latest literature on REST for existing approaches to combine the REST architectural style with offline operations. Richardson [7] as well as Webber [12] do not talk about offline support at all, since they only focus on how to design and implement the server-side of such a RESTful system. However, Richardson wrote at least one chapter about caching but focused on HTTP [8].

In contrast to literature, searching the World Wide Web leads to different approaches to implement offline support for RESTful systems. REST inherently fits many offline scenarios, mainly because of statelessness and the use of caches. For example, safe requests, like GET can easily be supported in offline mode due to caching. Moreover, idempotent requests, like PUT, and DELETE can be queued on the client side quite easily for later delivery [4]. Of course, the necessary synchronization of modified data back to the server might cause severe situations where it is difficult or even impossible to solve. However, there is no significant difference whether these problems occur at the time the request was issued or later during synchronization.

Riva and Laitkorpi summarized challenges and constraints of a mobile environment that are beyond the typical REST design [9]. One of these constraints is the offline/online behavior. They state that flaky network connections are typical for mobile devices. Therefore, mobile services have to define a strategy for supporting offline operations. Since they are designing mobile services using REST, their opinion is that offline support is fit for RESTful systems. However, they do not have any explicit support to process requests offline within their solution.

5 HIERARCHIES OF OFFLINE SUPPORT

In research question Q2, we ask for a proper software design to implement offline support for RESTful clients. In other words, we ask for a hierarchy of different levels in which offline support can be provided depending on specific requirements. We propose a level oriented architecture shown in Figure 1.

The lowest level represents caching and enables the client to process read operations. In addition to Fielding, who defined the caching constraint to reduce network traffic, caching can also be seen as strategy for processing requests while being offline [2].

To achieve the second level of offline support, idempotent operations have to be supported. The HTTP specification defines that the methods GET, PUT and DELETE have to be idempotent [4]. Therefore, those requests should be processible on this level. The client will need a queue to store all requests that were issued offline. Since caching is transient by default it has to be persistent on this level in order to allow the restart of clients. Query operations are also idempotent but are not considered within this level.

Handling Create, Retrieve, Update and Delete (CRUD) operations is defined on the third level. Create operations are not always idempotent. Therefore, client and server have to ensure that those requests will not be executed multiple times. The client needs additional knowledge in order to create temporary resources containing all relevant information like identifiers. During synchronization the temporary resources must be removed from the cache. Requests that are sent after the create operation have to be redirected to correct URIs.

The fourth level now also includes query operations. On the client side it cannot be guaranteed that all data is available in the local cache. Queries can only return a limited collection of resources if executed offline. The client does not know whether a request would return the same collection in online mode. In addition to the previous levels, the client also needs to support paging, i. e. only a limited number of results is returned instead of all results.

The fifth level is dedicated to the decision whether complex business logic is only available on the origin server or also on the client side.

A perfect clone of the origin server would achieve the sixth level. The most specific implementations have to be made on this level. Moreover, all functionality has to be implemented on client side. This is why the highest level is an utopia that can never be reached.

6 DERIVING OFFLINE SUPPORT

6.1 Overview of the Underlying Model

Every ϵ -NFA can be described as a tuple $(S, \Sigma, s_0, \delta, F)$, where S is a finite, non-empty set of states. Σ is a finite, non-empty set of symbols representing the input alphabet. $s_0 \in S$ is the initial state of the ϵ -NFA. δ is the state transition function $\delta : S \times (\Sigma \cup \epsilon) \rightarrow P(S)$, where $P(S)$ is the power set of S and $F \subseteq S$ is the set of accepting states. Zuzak, Budiselic and Delac described three main parts of ϵ -NFA operations: the *Input Symbol Generator*, the *Transition Function* and finally the *Current State* [13].

They mapped a RESTful system to the ϵ -NFA formal model as follows: Let *Reqs* be a finite set of valid requests, let *Metas* be a finite set of metadata key-value pairs, let *LTypes* be a finite set of link types and let *MTypes* be a finite set of media types. Finally, let

Figure 1: A six-level hierarchy for offline support.

$Reprs$ be a finite set of resource representations: $Reprs \subseteq data \times P(Metas)$ where one metadata element defines the media type of the representation.

Next, let Ops be a finite set of resource manipulation methods. The set of states S of the ε -NFA are the application states, where an application state is defined as a non-empty and ordered set of representations $S \subseteq P(Reprs) - \{\}$. In addition to Zuzak et al. we also include the Ops in the definition of states $S \subseteq (P(Reprs) - \{\}) \times Ops$. Therefore, an application has different states for manipulating and retrieving representations. Moreover, the initial state s_0 represents the initial application state, which is called dispatcher state. Zuzak et al. also defined the set of accepting states. In our opinion a RESTful system does not contain such accepting states, since it runs until it is undeployed. Therefore, let F be empty. The set of input symbols Σ of the ε -NFA represents requests $Reqs$ and their corresponding link types $LTypes$, $\Sigma \subseteq Reqs \times LTypes$. The transition function δ represents the translation of input symbols into requests, processing of requests into responses and integration of response representations into the next application state, $\delta : S \times (Reqs \times LTypes) \rightarrow P(S)$.

6.2 Deriving the Proxy Model

We distinguish between the origin server and the proxy on client-side. If the RESTful system has to work offline, all requests will be handled by the proxy. The origin is responsible for the conversion of requests into responses and the proxy has to convert the requests into responses while working offline, too. We defined two different limitations for offline mode: functional limitations and data limitations. The ε -NFA of the proxy is a subset of the ε -NFA of the origin. However, if the origin only serves Create, Retrieve, Update and Delete (CRUD) operations the proxy ε -NFA may contain all states of the origin.

Processing images and creating PDF files are two examples for functional limitations. Every state handling those functional limitations has to be removed from the proxy ε -NFA. Data limitations due to missing availability have to be considered, too. This could be the case with statistical evaluations but also simply with query endpoints. Therefore, every state has to be evaluated as to whether it returns a collection of resources. If working on a cached subset of data is sufficient, the state can be part of the proxy ε -NFA. If all data is needed for the processing of requests, the state has to be removed from the proxy.

By using the model of Zuzak, Budiselic and Delac we define a function Φ . This function uses the ε -NFA of the origin as input and

converts it to the ε -NFA of the proxy, $\Phi(M_{Origin}) \rightarrow M_{Proxy}$. Since both machines are described as tuples $(S, \Sigma, s_0, \delta, F)$, function Φ defines rules for every element of the tuple.

Every state without functional and data limitation is part of the finite set of states of the proxy. Functional limitations can be filtered by media type. A state is a non-empty, ordered set of representations, while a representation contains at least the media type as metadata key-value pair. Therefore, the function $mtime(s)$ returns the media type of the representations. $MTypes_{Online}$ is the finite set of media types that can only be provided in online mode. The function $count(s)$ verifies that enough data is available offline so that a non-empty, ordered set of representations can be returned in the response. The rules for filtering the origin states is defined as follows:

$$S_{Proxy} = \{s | s \in S_{Origin} \setminus \{ mtype(s) \notin MTypes_{Online} \wedge count(s) > 0 \} \} \quad (1)$$

Due to additional limitations it may be necessary to remove some of the input elements from the proxy. Since an input element is defined as the combination of $Reqs$ and $LTypes$, the $LType$ can be used to decide whether the input element should be part of the proxy. Therefore, we define a function $ltype(\sigma)$ that returns the $LType$ of the input element. $LTypes_{Online}$ is the finite set of link types that can only be provided in online mode. Function Φ uses the following rule for filtering the input elements:

$$\Sigma_{Proxy} = \{ \sigma | \sigma \in \Sigma_{Origin} \setminus \{ ltype(\sigma) \notin LTypes_{Online} \} \} \quad (2)$$

The initial state or dispatcher of the state machine is the same for both, the origin and the proxy. The dispatcher contains information about the next available states. Therefore, no data is needed and there are no functional limitations. It can be ensured that the dispatcher state will not be removed from the proxy by the defined rules of Φ :

$$s_{0Proxy} = s_{0Origin} \quad (3)$$

The transition function δ is also transformed by Φ . The translation of input symbols in requests, the processing of requests to responses and the integration of response representations into the next application state is defined as $S \times \Sigma \rightarrow P(S)$. The proxy may have a smaller power set due to some missing states and input

Figure 2: a) The method we used for verifying our work. We used the defined function Φ on an existing REST API manually. Afterwards we implemented the proxy layer for the first time based on the finite-state machine returned by Φ . Finally, we evaluated the proxy layer. b) Describes the method in future work, where the proxy is generated automatically.

elements, thus the transition function itself has to be filtered:

$$\delta_{Proxy} = \{(s, \sigma) | s \in S_{Proxy} \wedge \sigma \in \Sigma_{Proxy} \wedge (s, \sigma) \in \delta_{Origin}\} \quad (4)$$

The accepting states should only contain states $s \in S_{Proxy}$ if one follows the definition of accepting states by Zuzak et al. Since we have a different opinion, our set of accepting states remains empty in the proxy as well as in the origin. Therefore, we defined the rule:

$$F_{Proxy} = F_{Origin} \quad (5)$$

The function Φ transforms any ϵ -NFA of an origin server into an ϵ -NFA for a proxy on client-side. This allows a developer to implement an offline mode in their RESTful system.

6.3 Using the Transformation Function

Figure 2 shows the next steps of our method under a): at first, we have to define an application sample. Afterwards, the origin ϵ -NFA will be transformed into the proxy ϵ -NFA by manual application of function Φ . The resulting proxy will then be implemented and evaluated. If the proxy can support the application during offline mode, we consider Φ as correct.

7 AN OFFLINE FRAMEWORK

The RESTful system we used as an example implementation will be described in this section. Afterwards, we derive the proxy ϵ -NFA by the manual application of our function Φ . Therefore, the set of rules described in Section 6.3 is used. All relevant concepts for the implementation are also discussed in this section.

Figure 3 shows the ϵ -NFA of the sample application which does not contain transitions for error cases for sake of clarity. The sample application has users as primary resource and to-dos as sub-resource. The states for the to-do sub-resource are summarized as asterisk since they are similar to the CRUD states for the users' resource.

Creating a new user requires information such as username, email address, and password. However, the returned representations should not contain password information on retrieval. Therefore, we define two media types for the user resource. The to-do resource requires only one representation, and therefore, only one media type.

Input elements were defined as requests and their corresponding link types. Input elements can affect different resources. However, they can be grouped by their *LType* since the same *LType* has an equivalent effect on different resources.

The next step is to apply the function Φ manually. Therefore, the finite set of states as well as the finite set of input elements have to be filtered. Based on the defined rules we dropped the states

for profile images and passwords. Furthermore, the input elements with the corresponding link types were removed. Figure 3 shows the resulting ϵ -NFA of the proxy on the left side of the dashed line.

The derived proxy will be implemented as prototype for evaluation. The proxy should then enable offline support on the fourth level of our hierarchy. The essential component of the first level is the cache. Since the media types of the example application are based on JSON the cache component should be able to handle finite sets of key value pairs. The second level requires the cache to be persistent in order to allow the client application to restart without losing the current state of data. However, for this prototype we will only implement an in-memory cache for simplicity.

An additional component has to enqueue all requests that were issued offline. The enqueued requests can later be replayed to the server. The fourth level can also serve non-idempotent requests which may lead to conflicts. Furthermore, lost update problems may occur if multiple users are updating same resources. Therefore, the proxy implementation has to detect those conflicts. If conflicts were detected the proxy has to solve them either by itself or by using a callback function asking the application for help. The component for request execution itself has to interpret incoming requests and manipulate the cached data as well as handling queries. Therefore, a component for filtering cached resources and enabling pagination is required.

8 DISCUSSION

8.1 Implementing the Prototype

To achieve transparency for the developers of RESTful clients, we decided to decorate an existing HTTP client. The processing of requests can be seen in Figure 4. The request handler determines whether the client is online. In online mode the requests will be directly sent to the origin server. In offline mode all requests will be redirected to the proxy. The proxy will respond to the requests based on cached resources, if possible. Afterwards, the requests are enqueued for later synchronization. GET requests will not be stored in the queue since they were already answered by the proxy and do not modify representations of resources. As soon as the client goes online again, the queue gets synchronized to the origin server.

Offline support was implemented with states. There is one client state for every HTTP method. Each state has the capabilities to operate on the persistent cache. After the required CRUD operation was executed a response is built. The response contains all hypermedia links that the origin server would have returned except for those removed by function Φ . However, it is possible that an application sends requests to cached hyperlinks that are not part of the proxy ϵ -NFA. The proxy implementation will detect those requests and replies with status code 503 - *Service Unavailable*.

Figure 3: The finite-state machine for the sample to-do-list application. The asterisk represents the default CRUD states for the to-do sub-resource similar to the user states.

8.2 Testing the Prototype

For testing, we implemented a random request generator that can easily generate hundreds of valid requests. Moreover, we manually prepared an in-memory cache so that we would not have to bother with requests for filling the cache. Since our prototype cannot dynamically load the required information via *OPTIONS* request we also implemented the application specific classes.

The test cases are using two clients: one that works offline and one that works online. Since the offline client is the test subject, the online client is only necessary to produce some conflicts during synchronization. The request generator uses different probabilities for each HTTP method. Since the sample application is focused on resource manipulation requests, we increased the probability for those.

The tests were successful and proved that our proxy prototype works correctly. Since the proxy is a subset of the origin ε -NFA, it is clear that the proxy itself is also a RESTful system.

8.3 Suitability of the Proxy Layer

Fielding defined five mandatory constraints for REST and the principle of layered systems is one of them [2]. An additional layer for offline support, fits this requirement. The hypermedia constraint is also fulfilled due to the proxy returning all offline supported hypermedia links within responses.

The proxy layer has to manipulate some of the requests while replaying the queue. According to Fielding “intermediary components can actively transform the content of messages because the messages are self-descriptive and their semantics are visible to intermediaries.” [2, 5.1.6] The manipulation of requests is thus allowed and possible.

The server side does not have to care about offline support since the proxy is not dependent on the server. The server does not have to provide additional information for the proxy layer within resource representations. The developers do not have to change the server implementation if they want to add offline support to their RESTful systems.

Using a proxy layer has some disadvantages. The proxy layer provides its own cache of resources. If the application using the proxy layer also implemented a cache, the mobile client could reach its memory capacity. This issue has to be considered while planning the offline support for an existing application. If the application is

built from scratch the developer could consider not implementing an own cache on top of the proxy layer.

Another disadvantage is that the proxy layer can only process the requests that are part of the input elements of the proxy ε -NFA. If the layer above requests a cached hyperlink that cannot be processed, the proxy will return a *503 - Service Unavailable* response. Therefore, the layers above the proxy layer have to handle those responses. If the application itself implement offline support those conflicts could be avoided.

9 CONCLUSION AND FUTURE WORK

We implemented offline support for RESTful systems to achieve a better user experience in several scenarios. Therefore, we answered the question whether REST and offline support are combinable and defined a hierarchy of six levels for offline support. With the help of Zuzak’s formal model we derived the function Φ to retrieve the ε -NFA for the proxy layer. We implemented the proxy that resulted from the manual use of Φ . Since our tests were successful we proved that our model is correct.

We reached the goals of Demers et al. [1]. Some functionality was removed from the proxy due to limited resources on the clients. We achieved a high availability of reads and writes and conflicts can be detected via Etags. The prototype also follows the model of Gonçalves and Leitão for working offline [5]. A local server works as proxy, saves the offline requests, and returns offline responses. When going online all saved requests are synchronized with the origin server. If the client is back online it sends all saved requests directly to the origin server.

The implemented proxy layer only served as a prototype. In the future the prototype has to be refactored and extended. The cache as well as the queue have to be stored persistently to enable restarting of the client application. Moreover, the information given in the application specific classes can be loaded dynamically from the origin server. The proxy has to understand the message of the server and parse the information correctly.

In the future we also plan to implement the fully automatically application of function Φ to a model of the RESTful application. With regards to the work of Schreibmann and Braun it may be possible to implement Φ within their model-driven approach to enable automated generation of proxy layers for modeled RESTful APIs [11]. Figure 2 shows the future process by using the model-driven approach in *b*): the required RESTful application will be

Figure 4: The client sends requests to the proxy when working offline. The proxy enqueues every POST, PUT and DELETE request for synchronization. GET requests can be dropped since they do not modify representations. The client sends requests to the origin server directly when working online. For the sake of clarity we do not include responses.

described in the model, afterwards the proxy will be generated by the automatic use of function Φ and can finally be deployed. This would also allow us to generate application specific classes and functionality. Therefore, it could be possible to generate a proxy layer above the fourth level of our hierarchy for offline support. The generated proxy layer could then enable even more functionality while being offline.

REFERENCES

- [1] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. 1994. The Bayou Architecture: Support for Data Sharing Among Mobile Users. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*. 2–7.
- [2] R.T. Fielding. 2000. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine.
- [3] R.T. Fielding. 2008. REST APIs must be hyper-text driven. (10 2008). <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>.
- [4] R.T. Fielding and J. Reschke. 2014. RFC 7231, Hypertext Transfer Protocol – HTTP/1.1: Semantics and Content. (2014). <http://tools.ietf.org/html/rfc7231>.
- [5] Edgar E. M. Gonçalves and António Menezes Leitão. 2007. Offline execution in workflow-enabled Web applications. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. 204–207.
- [6] Edgar E. M. Gonçalves and António Menezes Leitão. 2009. Using Common Lisp to Prototype Offline Work in Web Applications for Rich Domains. In *Proceedings of the 6th European Lisp Workshop (ELW '09)*. ACM, New York, NY, USA, 18–27.
- [7] L. Richardson, M. Amundsen, and S. Ruby. 2013. *RESTful Web APIs*. O'Reilly Media.
- [8] L. Richardson and S. Ruby. 2008. *RESTful Web Services*. O'Reilly Media.
- [9] Claudio Riva and Markku Laitkorpi. 2009. *Service-Oriented Computing - ICSOC 2007 Workshops: ICSOC 2007, International Workshops, Vienna, Austria, September 17, 2007, Revised Selected Papers*. Springer Berlin Heidelberg, Chapter Designing Web-Based Mobile Services with REST, 439–450. https://doi.org/10.1007/978-3-540-93851-4_42
- [10] M. Satyanarayanan. 2002. The Evolution of Coda. *ACM Trans. Comput. Syst.* 20, 2 (May 2002), 85–124.
- [11] Vitaliy Schreibmann and Peter Braun. 2015. Model-Driven Development of RESTful APIs. In *Proceedings of the 11th International Conference of Web Information Systems and Technologies*. INSTICC, SciTePress, 5–14.
- [12] J. Webber, S. Parastatidis, and I. Robinson. 2010. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media.
- [13] 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.