Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

TOBIAS FERTIG
MASTER THESIS

TOWARDS OFFLINE SUPPORT FOR RESTFUL APPLICATIONS

Submitted on September 27, 2016

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, September 27, 2016

License

Erlangen, September 27, 2016

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 more than short-term offline operations, yet many applications must keep functioning when going offline for more than a few seconds. Burdening the application with knowledge about offline status is undesirable. We define a function to derive a finite-state machine for the client side based on a formal model to describe RESTful systems as finite-state machine. We then extend existing caching approaches for offline operation so that a client-side proxy can transparently hide the offline status from the application for all derived states. We validate our solution with a proxy layer that covers all state-model derived test cases. Using our model and proxy, clients do not have to know and worry about whether they are online or offline.

Acknowledgments

Over the past six months I have received support and encouragement from a great number of individuals. Dr. Martin Jung and Samir Al-Hilank have been great mentors and their guidance has made this a thoughtful and rewarding journey. They also supported my idea for the topic of this thesis. I would like to thank Prof. Dr. Dirk Riehle for his support, time and patience not only during my master thesis but also during my whole studies. In addition, I would like to thank Prof. Dr. Peter Braun for spending hours listening to me talk about my research.

Last but not least I would like to thank my family and friends. Their patience and encouragement gave me the energy to work hard on this thesis.

Contents

1 Introduction

Representational State Transfer (REST) was introduced by Fielding (2000) within his dissertation. Developers ignoring constraints of REST were the reason for Fieldings blog post[1] in 2008. In it he explains that REST APIs must be hypertext driven in order to fulfill the hypermedia constraint. This constraint requires a stable internet connection to allow the client to communicate with the server. However, stable internet connections are not possible in any place at any time. Therefore, offline support on the client side would improve the user experience. We encountered several scenarios that would profit from offline support:

Farmers have to document every step of their daily work in the fields[2]. Connection problems are a common problem in agriculture and forestry. Farmers often have to document their steps with pen and paper. Service technicians are not always allowed to have internet access due to fear of industrial espionage. Customers using apps for shopping lists may have no access in the basement of a shop. Moreover, anybody will encounter connection problems while traveling around the world on top of a mountain or in the forests. An offline mode would increase the user experience in these situations.

1.1 Thesis Goals

This thesis proposes a solution for offline support in RESTful systems. Therefore, it has to be checked whether offline support and REST fit together. If Fielding's constraints for REST can be fulfilled despite providing offline support it will fit. Another goal of this thesis is to check what functionality can be made available offline. This thesis will define a model to decide whether a given functionality can be supported offline.

Within this thesis different approaches for offline support will be drafted. One approach will be implemented as prototype and discussed in the research chap-

[1]http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[2]https://www.landwirtschaftskammer.de/dueren/download/formulare/schlagkartei.htm

ter. The other approaches will be summarized in the elaboration chapter. The prototype will not be fully functional. The implementation will be limited to the features providing new findings for this topic. Therefore, the prototype will work in memory instead of using databases. Moreover, this thesis does not focus on prefetching algorithms for caching. We assume that the cache of our prototype is already filled and contains data to work with.

1.2 Thesis Structure

After this introduction the research chapter begins. The research chapter contains the findings of this thesis in paper format. It was limited to 6000 words by the professorship to match limitations of conferences. The research chapter itself follows a common paper structure:

It starts with an introduction. Afterwards, related work will be discussed, followed by our research questions. After explaining our research approach the research results will be summarized. Moreover, we will discuss our results and close up by a conclusion and future work.

The elaboration chapter contains additional information that did not make the cut for the research chapter. The challenges of supporting offline modes will be discussed in more detail. The implementation of our prototype will be explained. We will also explain additional approaches that were not chosen for the prototype. Finally, we will take a look on offline support for microservices.

Within the appendices we will give a short deployment manual for our prototype and some information about the contents of the CD.

2 Research Chapter

2.1 Introduction

Representational State Transfer (REST) was introduced by Fielding (2000) within his dissertation. Developers ignoring constraints of REST were the reason for Fieldings blog post[1] in 2008. In it he explains that REST APIs must be hypertext driven in order to fulfill the hypermedia constraint. This constraint requires a stable internet connection to allow the client to communicate with the server. However, stable internet connections are not possible in any place at any time. Therefore, offline support on the client side would improve the user experience. We encountered several scenarios that would profit from offline support:

Farmers have to document every step of their daily work in the fields[2]. Connection problems are a common problem in agriculture and forestry. Farmers often have to document their steps with pen and paper. Service technicians are not always allowed to have internet access due to fear of industrial espionage. Customers using apps for shopping lists may have no access in the basement of a shop. Moreover, anybody will encounter connection problems while traveling around the world on top of a mountain or in the forests. An offline mode would increase the user experience in these situations.

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[3]. Offline first supposes to write an app as if it needs no internet connection. Network features can be added once the app works offline. Nevertheless, developers have to implement offline support manually. 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

[1]http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[2]https://www.landwirtschaftskammer.de/dueren/download/formulare/schlagkartei.htm

[3]https://developer.chrome.com/apps/offline_apps

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. In Section 2.4 we describe how the questions will be answered and afterwards we show the results we got by research and implementation. Finally, we will give a short outlook on future work.

2.2 Related Work

This section gives an overview of related approaches for caching and offline behvavior. Besides web and mobile applications we will investigate additional fields of computer science. The end of this section contains a discussion about some frameworks for offline applications.

If we talk about caching one might think about hardware caching and how it is done by CPU and RAM. In contrast to web applications the data required by CPU or RAM is available on the hard disks at any time. This availability is the main difference to offline support in web applications. The caching on hardware level serves another purpose. Caching is mainly implemented to increase the performance (Handy, 1998). Caching in web applications can increase the performance but is also used to tolerate flaky internet connections.

Version Control Systems (VCS) like GIT are tools with offline support. They solve many problems of offline support like detection and resolving of conflicts as well as data synchronization with remote repositories. The main difference is that all features of the VCS are available on every client. Moreover, unique IDs can be generated by every client since no server-specific data is required. Additionally, clients do not have to struggle with limited data access as all required data is pulled from the repository. Therefore, approaches used by VCS do not fit into offline web applications.

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 fitting goals. Demers et al. (1994) defined the following goals for the Bayou Architecture which are also relevant for our work: 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.

Satyanarayanan (2002) proposed Coda, a distributed file system that runs with a client-server model. 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. Due to the stateless constraint defined by Fielding (2000) this approach is not applicable for RESTful systems.

Gonçalves and Leitão (2007) contributed a lot to enable offline execution in web applications. They described an offline model that contains a subset of all 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 register the offline request and return an offline response. If the client goes online, all offline work has to be synchronized with the server (Gonçalves & Leitão, 2009). However, Gonçalves and Leitão (2007) focused on prefetching mechanisms to fill the offline cache which is not part of this work.

Working offline is a common problem for some applications which is why frameworks handling this topic were released. Hoodie[4] is a library package designed for frontend web applications and follows the offline first approach[5]. Hoodie offers a local, user-specific database that is automatically synchronized with the server. Loopback[6] 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 can also be used to synchronize multiple backends, but its focus is not mainly 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 to new RESTful systems as well as to existing ones.

2.3 Research Questions

At first RESTful systems with offline support might seem like an oxymoron. Nevertheless, offline support is required in many applications. Therefore, the necessity of offline support is out of question but is it possible to provide it within RESTful systems? To fulfill the hypermedia constraint, REST relies on stable internet connections. In case of flaky connections the hypermedia constraint has still to be fulfilled.

The next step would be to think about the different categories and levels of offline support. The definition of a hierarchical structure of offline support seems useful. A possible categorization could differ according to its access to data. However, read-only access can also be enabled via caching. Therefore, it has to be clarified

[4]http://hood.ie
[5]http://offlinefirst.org
[6]https://loopback.io

whether caching itself represents one level of offline support. With this hierarchy we are able to describe on which level an offline model for REST could work.

Solutions for offline support in applications have to address functional and data-access problems. Software developers can implement those solutions in three ways: manually, by using frameworks and by code generation. Manual implementations often duplicate the server code and port it to the client. However, this approach is time-consuming and can lead to copy and paste errors. The use of frameworks can help but, as discussed in 2.2, most of them do not focus on REST constraints. Moreover, they only cover a subset of offline support, like read-only access while we try to find a generic solution for offline support. Since it will not be possible to enable all functionality in a generic way, we try to minimize the application specific part.

To sum it up the three research questions are the following: Are RESTful systems and offline support combinable? How is it possible to organize offline supported functionality hierarchically? And is there a generic solution for offline support in RESTful systems?

2.4 Research Approaches

These questions will be answered by the following research approaches: The first question can be answered by literature research. Therefore, the latest publications on REST will be reviewed. To answer the second question a literature research will be done, too. If no categorization is available, we have to define our own levels of offline support. To give an answer to the last research question we propose our model for offline support. Zuzak, Budiselic, and Delac (2011) proved that RESTful APIs can be modeled using finite-state machines. They used a nondeterministic finite-state machine with ε-transitions (ε-NFA) to explain the operation of RESTful systems. Considering their formal model we defined a transformation function Φ to convert the ε-NFA of the origin server into the ε-NFA of the proxy on client side.

2.4.1 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 $\delta : S \times (\Sigma \cup \varepsilon) \rightarrow P(S)$, where $P(S)$ is the power set of S and $F \subseteq S$ is the set of accepting states. Zuzak et al. (2011) described three main parts of ε-NFA operations: the *Input Symbol Generator*, the *Transition Function* and finally the *Current State*.

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 *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, ordered set of representations $S \subseteq P(Reprs) - \{\}$. In addition to Zuzak et al. (2011) 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. (2011) 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 transistion 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, $\delta : S \times (Reqs \times LTypes) \rightarrow P(S)$.

2.4.2 Deriving the Proxy Finite-State Machine

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. Therefore, the proxy has to convert the requests into responses while working offline. 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, $M_{\text{Proxy}} \subseteq M_{\text{Origin}}$.

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 et al. (2011) we defined a function Φ. This function uses the ε-NFA of the origin as input and converts it to the ε-NFA of the proxy, $\Phi(M_{\text{Origin}}) \to M_{\text{Proxy}}$. Since both machines are described as the tuple $(S, \Sigma, s_0, \delta, F)$, 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 *mtype*(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 *countReps*(s) verifies that enough data can be 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:

$$S_{Proxy} = \{s | s \in S_{Origin} \setminus \{mtype(s) \notin MTypes_{Online}; countReps(s) > 0\}\}$$

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 defined a function *ltype*(σ) 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:

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

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:

$$s_{0_{Proxy}} = s_{0_{Origin}}$$

The transition function δ is also not 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 $S \times \Sigma \to P(S)$. The proxy may have a smaller power set due to some missing states and input elements, but the transition function itself remains unchanged:

$$\delta_{Proxy} = \delta_{Origin}$$

The accepting states should only contain states $s \in S_{Proxy}$ if one follows the definition of accepting states by Zuzak et al. (2011). 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:

$$F_{Proxy} = F_{Origin}$$

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

2.4.3 Using the Transformation Function

The next step is to define an application sample. Afterwards, the origin ε-NFA will be transformed into the proxy ε-NFA by the manual use of 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. Since we mentioned only a general set of rules for Φ in Section 2.4.2, we will now give some more examples.

The states are filtered via media types due to functional limitations. The media types *application/pdf* and *image/** are two examples that have to be removed from the proxy ε-NFA. Image processing functionality is too complex to be implemented on the client side. Some clients may not support image processing libraries or have only limited hardware resources. Since the generation of PDF files is also not available on every client, we blacklisted this media type as well.

The proxy is allowed to provide endpoints that return a collection of resources. However, we have to take into consideration that the proxy can only use cached data. It may occur that a request handled by the proxy will return a different non-empty set of resource representations than the origin would have returned. If the proxy has to process a request whose state is not available in offline mode, the proxy will return the http status code *503 - Service Unvailable* and will redirect to the initial state of the ε-NFA.

2.5 Research Results

In this section the three research questions will be answered. The first one by literature, the second by our own hierarchy and the last one by describing the concept for an implementation.

2.5.1 RESTful Systems and Offline Support

Current literature about REST does not cover offline support. Richardson, Amundsen, and Ruby (2013) as well as Webber, Parastatidis, and Robinson (2010) do not talk about offline support at all since they mainly focus on the server side. However, Tilkov (2013) wrote at least one chapter about caching but focused on http.

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. Idempotent requests can be queued on the client side quite easily for later delivery. However, in case of dynamically created resources the client has to provide URIs. In case of non-idempotent requests the client needs a mechanism to detect conflicts. Non-idempotent requests can not be easily repeated in case of failure.

Riva and Laitkorpi (2009) summarized challenges and constraints of a mobile environment that are beyond the typical REST design. 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.

Our opinion is similar to Riva and Laitkorpi (2009): RESTful systems can be built with offline support. The following challenges have to be considered:

- *Client-provided IDs.* In case of dynamically created resources the client has to create the ID for the resource as well as the URI. If an ID is already in use to a concurrent client, conflict resolution has to be implemented. A resource created via POST request will per definition get a server-provided URI. It is reasonable to provide temporary IDs in offline mode that will be replaced with actual IDs as soon as the synchronization with the backend is running.
- *Data synchronization.* Executing requests offline is not sufficient. It is also necessary to synchronize the changed resources with the backend. Conflicts may appear and have to be resolved. Preconditions like etags may also change during synchronization and have to be considered.
- *Lost-Update problems.* Two different clients can work on the same resources. One client can work offline on cached resources while the other one sends requests to the backend. In this case the data has to be merged. If merge conflicts appear, the client or the server has to resolve them.

- *API Hooks.* Hooks in RESTful APIs are a common scenario. Hooks for email notifications are an example for API hooks. Whether a hook should be executed after synchronization can not be decided generally. It would be possible to include timestamps in every request. The server can then decide to execute the hook if necessary.
- *Subscriptions.* Many APIs support subscriptions of resources. Without subscriptions a client sends requests to the backend to check if a resource has changed. To reduce the amount of requests, a client can subscribe for resources. If the resource changes the server sends a notification to the client. If the client is offline the server has to queue the notification and deliver it later.
- *Authorization and access rights.* The backend returns hypermedia links to resources based on user roles and access rights. If a client has offline support, it has to be ensured that cached resources can only be accessed by users with corresponding rights.

Despite of all those challenges it is possible to enable offline support for RESTful systems.

2.5.2 Hierarchies of Offline Support

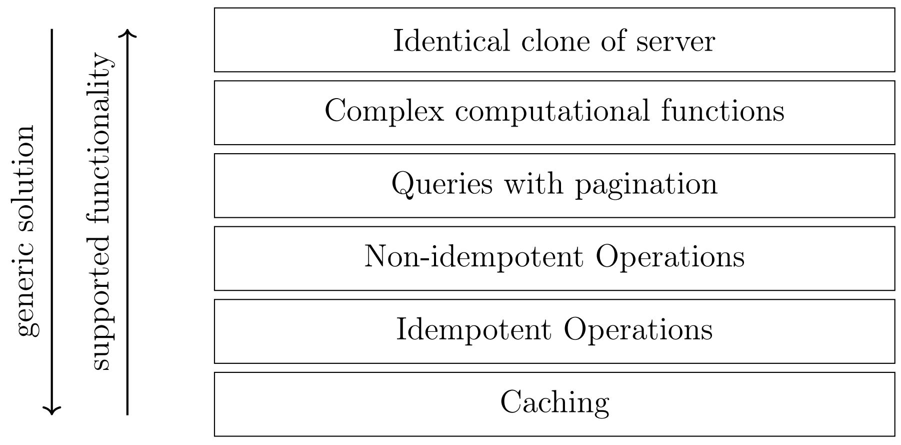

Figure 2.1: A six-tier hierarchy for offline support.

We defined a hierarchy with six tiers of offline support. A higher tier supports more functionality. However, additional challenges have to be solved at higher tiers. In contrast to generic solutions on lower tiers more specific implementations are also needed at higher tiers.

Figure 2.1 shows our six levels. The lowest tier represents caching and enables the client to process read operations while offline. In addition to Fielding (2000), who defined the caching constraint to reduce network traffic, caching can also be seen as strategy for processing requests while being offline. Due to the caching constraint this tier can be achieved by every RESTful system.

To achieve the second tier of offline support, idempotent operations have to be supported. The HTTP specification defines that the methods DELETE, GET and PUT have to be idempotent. Therefore, those requests should be processible on this tier. In addition to caching the client will need a queue to store all requests that were executed offline. The cache needs no additional functionality since delete and update operations should be available in every cache. However, caching is transient but has to be persistent on this tier. A client may have to reboot while working offline and the work should still be available afterwards. Query operations are also idempotent but are often marked as uncacheable in RESTful systems. Therefore, additional methods are needed since queries can not easily be processed by the default caching strategies. Queries are thus not supported on the second tier.

Create, Retrieve, Update and Delete (CRUD) operations define the third tier. 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 IDs. During synchronization the temporary resources must be removed from the cache. Requests that follow up the create operation have to be redirected to correct URIs.

The fourth tier also includes query operations. Queries often return a collection of resources. Therefore, the whole data is queried and the resources are filtered by query parameters. The client cannot guarantee that all data is available in cache. An query 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 tiers the client also needs to support paging.

Access to all data stored on the backend can not be guaranteed by the client. The client only has access to data that was cached before the network connection failed. However, distinguishing between functionality that can be implemented on the client side and functionality that can not be implemented is necessary. The fifth tier contains all functionality that can somehow be enabled independent of effort in implementation.

A perfect clone of the server represents the sixth tier. The most specific implementations have to be made on this tier. Moreover, all functionality has to be implemented on client side. Therefore, this tier contains all functionality that is not possible to implement on client side and has access to all data available on the server. This is why the highest tier is an utopia that can never be reached.

2.5.3 An Offline Framework for RESTful Systems

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 use of our function Φ. Therefore, the set of rules described in 2.4.3 is used. All relevant concepts for the implementation are also discussed in this section.

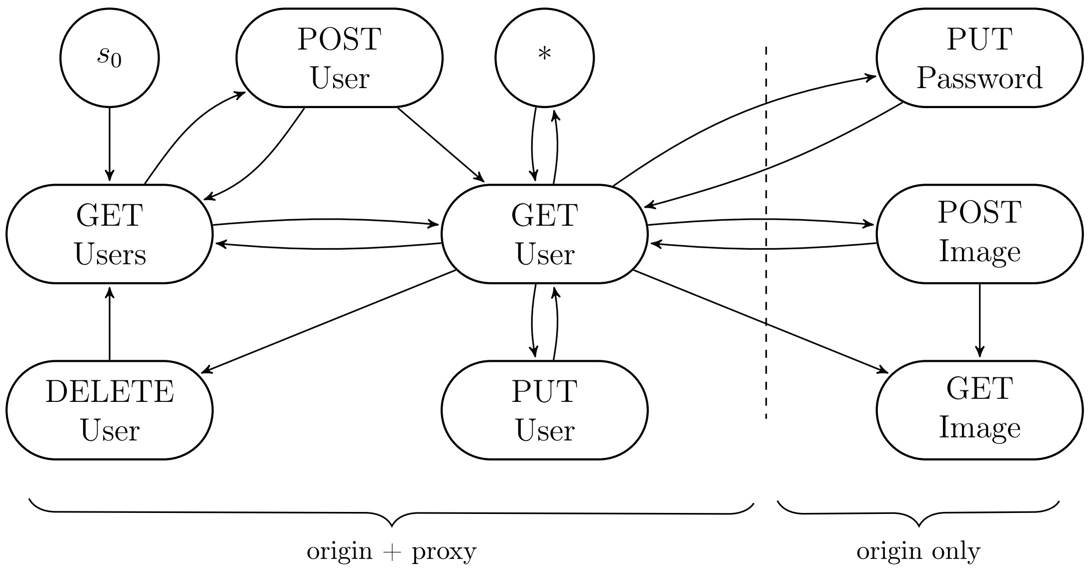

Figure 2.2: The finite-state machine for the sample to-do-list application. The asterisk represents the default CRUD states for the to-do sub-resource similar to the user states.

The most common example for offline frameworks is an application for to-do-lists. Therefore, we also chose a to-do-list application. Figure 2.2 shows the ε-NFA of the sample application. The diagram does not contain transitions for error cases. Moreover, every state has a transition to the initial state. 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, on retrieval the returned representation should not contain the password. Therefore, we defined two media types for the user resource: one containing the password for user creation and updating of passwords as well as one without password for user retrieval. The to-do resource requires only one representation and therefore only one media type. The finite set of media types is defined as follows:

$$
\begin{aligned}
MTypes = \{&image/*;\\
&application/user.default + json;
\end{aligned}
$$

$$application/user.userwithoutpassword + json;$$
$$application/todo.default + json\}$$

We described the set of states on an abstract level since every resource of a given type can put the application into equivalent states. Since the *Ops* are part of a state it is possible to summarize the states using the *Ops*. The user resources therefore have one state per CRUD operation. An additional update state is defined to support password changes. Since the user resource embeds a profile picture two more states are required: one for uploading a profile picture as well as another one for retrieving the profile picture. The to-do resources do not require additional states besides those serving the CRUD operations. For users and also to-dos we defined query states to filter the resources on the server.

We also described the set of input elements on an abstract level. 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 example application defines the set of *LTypes* as follows:

$$\begin{aligned} LTypes = \{&createUser; updateUser; updatePassword; queryUsers;\\ &getSingleUser; deleteUser; uploadImg; getImg; createTodo;\\ &updateTodo; queryTodos; getSingleTodo; deleteTodo\}\end{aligned}$$

The next step after defining the ε-NFA for the origin 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 retrieving and uploading profile images. The state for updating user passwords will also be dropped from the proxy state machine for security reasons. Furthermore, the input elements with the link types *updatePassword, uploadImg* and *getImg* have also to be removed. Figure 2.2 shows the resulting ε-NFA for 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 tier of our hierarchy. The fourth tier can be implemented in a generic way due to the small amount of application specific features. The resulting proxy layer can thus be easily reused for other applications.

The essential component of the first tier 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 tier requires the cache to be persistent in order to allow the client to reboot without losing work. However, for this prototype we will only implement an in-memory cache.

An additional component has to enqueue all requests that were executed offline. The enqueued requests can later be replayed to the server. The fourth tier can also serve non-idempotent requests which may lead to conflicts. Furthermore, lost update problems may occur if multiple users are updating the 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. Idempotent operations can easily be processed by retrieving, replacing or deleting the cache entry. On the contrary non-idempotent requests need additional capabilities like generating temporary URIs or IDs. Since we want to achieve the fourth tier our prototype also has to handle queries. Therefore, a component for filtering cached resources and enabling pagination is required. However, the prototype will not implement filters since they are application specific and have to be provided by the developers using the generic proxy layer.

2.6 Discussion

This section describes how the proxy layer is implemented. Afterwards, the set up of the test cases is explained. The test results will show that the function Φ is working. Finally, the disadvantages of using a proxy layer are discussed.

2.6.1 Implementing the Prototype

To achieve transparency for the developers of RESTful clients, we decided to decorate a common http client. The prototype is based on the apache http client [7]. The developer of the application can use our offline client instead of the apache http client. Since we used the decorator pattern our client provides the same interface (Gamma, Helm, Johnson, & Vlissides, 1995).

The abstract method *doExecute* was overwritten and does now call a request handler that will be provided by a factory based on the http method (Gamma et al., 1995). The request handler determines whether the client is online. In online mode the request will be enqueued and the queue will be replayed to the server. The response of the last request is the required one and will be returned to the caller. If the client goes offline during synchronization the handler will switch to offline processing. In offline mode the current request will be enqueued and afterwards the offline processing will start.

[7]https://hc.apache.org

Offline processing 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 build. The response contains all hypermedia links that the server would return except for those removed by our 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 answers them with a *503 - Service Unavailable* response.

The prototype is a generic solution for RESTful clients. However, it depends on three classes that have to be extended by the developer since they are application specific. The *ViewDefiner* defines the keys of the finite sets of key value pairs that are described via media type headers. The *StateFinder* and *StateTransitions* describe the ε-NFA of the proxy. If the backend provides an *OPTIONS* endpoint the information those three classes provide could be dynamically loaded.

2.6.2 Testing the Prototype

For testing we wrote a random request generator that can easily generate hundreds of 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 needed 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. Since we used the formal model of Zuzak et al. (2011) to derive our proxy, it is ensured that the proxy itself is also a RESTful system. The proxy is a subset of the origin and allows offline support on the fourth level of our hierarchy described in 2.5.2. It would be possible to achieve a higher level only at the cost of implementing more application specific code. However, the function Φ ensures the result to be a ε-NFA that can be used by a proxy implemented for the fourth level.

2.6.3 Suitability of the Proxy Layer

Fielding (2000) defined five mandatory constraints for REST. Layered system is one of those constraints. Therefore, introducing an additional proxy layer fits those constraints. The other layers on the client side would not be affected by the proxy layer which can thus easily be replaced or changed. The hypermedia constraint is also fulfilled due to the proxy returning all offline processable hypermedia links within responses.

The proxy layer has to manipulate some of the requests while replaying the queue. According to Fielding (2000) in Section 5.1.6 “intermediary components can actively transform the content of messages because the messages are self-descriptive and their semantics are visible to intermediaries”. 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 limits. 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.

2.7 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 tiers 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. (1994). 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 (2007) for working offline. A local server works as proxy, registers the offline requests, and returns offline responses. When going online all offline work is synchronized with the server. If the proxy is online it forwards all requests directly to the 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 rebooting of the client. Moreover, the information given in the application specific classes can be loaded dynamically from the backend. The proxy has to understand the message of the server and parse the information correctly.

The synchronization process can also be improved. Multiple PUT requests on the same resource could be combined into one single request. That would simplify the synchronization and merging process. However, the order of the requests would not remain the same compared to the current replaying process. Check whether such a combination of requests can be done without impediments will need additional work. Currently the proxy layer ignores queued GET requests. However, they could be used to update the caches during synchronization.

Since we only used the function Φ manually an automation of this function would be useful. Concerning the work of Schreibmann and Braun (2015) it may be possible to implement Φ within their model-driven approach to enable automated generation of proxy layers for modelled RESTful APIs. Using the model-driven approach would 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 processing. The generated proxy layer could then enable even more functionality while being offline.

3 Elaboration Chapter

This chapter will discuss the challenges of enabling offline support for RESTful systems. Afterwards, we will give some insight into the implementation of the example server and client. Finally, we will explain additional approaches that were discovered during this thesis.

3.1 Challenges

Section 2.5.1 mentions several challenges that have to be passed while enabling offline support for RESTful systems. Some challenges were already discussed within the research chapter and addressed by our prototype. However, we also considered some solutions for the other challenges.

3.1.1 Hooks

Hooks are a common scenario in APIs. An example for API hooks could be email notification if a resource was updated. When working offline hooks can not be executed since the proxy does not know about them. Since the synchronization to the backend is a replay of enqueued requests the hooks will be executed during the replay. However, a hook execution may become irrelevant hours after the request was handled offline on the client side. This can not be decided generally for all hooks. The developer has to decide whether his API contains hooks that can become irrelevant.

If the API contains hooks that can be ignored during request replay, the developer has to ensure that every request contains a timestamp. This timestamp defines when the request was originally triggered. The backend can then decide if a given hook has to be executed or not. Another approach is to distinguish between offline and online mode. However, this approach reduces the transparency.

3.1.2 Authorization and Access Rights

Hypermedia links returned from the backend are generated depending on the user role. Moreover, representations can be different for different user roles. The proxy has to ensure that cached data of one user can not be accessed by another user. Therefore, user specific storage has to be implemented. The dynamic creation of hyperlinks could be achieved by the options endpoint. The options endpoint will only return the accessible states for the requesting user role. If the user sends a request to a forbidden state, the proxy will return a *503 - Service Unavailable* response since the proxy does not know the state.

3.1.3 Resource Subscriptions

Resource subscriptions are common practice for Web APIs. Without subscriptions polling is the only way to detect changes. The client sends GET requests in a given time interval to retrieve the latest resource representations. Most of the polls are wasted since resources are not updated all the time. Therefore, many APIs enable subscriptions to reduce the network traffic. The client can subscribe a given resource and the server sends notifications to all registered clients if an update occured. The clients can then request the resource to get the latest representation.

If the client has offline support, the server needs additional functionality to handle the notifications. If the server is not able to reach the client, the client has gone offline. The notification has to be enqueued on server side for later delivery. However, if the server resends the notification in a given time interval the same drawbacks occur as is the case with polling.

Clients with offline support could be treated differently from those being online all the time - except during failure. An offline working client has to synchronize its data. During synchronization the client can also update all relevant subscriptions. The server no longer has to enqueue notifications if it cannot reach the offline clients. Another approach would be to unsubscribe from resources before going offline. However, in case of flaky network connections the client may not be able to unsubscribe. The server would therefore need a fallback method.

3.1.4 Resolution of Merge Conflicts

Multiple clients working on the same resources can lead to lost updates if not handled properly. Therefore, conditional requests are used in RESTful APIs. The client has to add an etag to writing requests. The server can determine whether the client knows the latest representation and can either accept or refuse the modification. If the response has the status code *412 - Precondition Failed* the conflict handler of the prototype will be called. We offer two default implementations: firstly, the client wins handler and secondly, the server wins handler. Moreover, custom implementations are possible but have to be implemented by the developer.

A custom implementation would allow auto-merging of representations. Every key value pair has to be checked separately. If the latest representation of the server and the current representation of the client contain changes in different key-value pairs an auto-merge would be easy. However, there are some issues with embedded resources. One client could update the embedded resource itself while another updates the enclosing resource. The auto-merging functionality has to detect the conflict. Therefore, the etag of the enclosing resource also has to cover the embedded resource. Otherwise, the changes of the embedded resource will be overwritten.

3.2 Implementation Details

This section will cover details of the prototype implementation. Although the prototype is a solution for clients of RESTful systems, we needed a working backend in order to test the prototype. Therefore, we implemented a REST API fulfilling the ε-NFA of Figure 2.2. The server side as well as the client side are working in memory only. We did not implement any persistence accept for the uploaded profile pictures. Those are stored on the hard disk.

3.2.1 The Server Side

The server side is implemented with *jersey* and *jax-rs*[1]. The class *ResourceConfig* from jersey has to be extended for configuration. The ResourceConfig is used to register all required classes for the REST API. To provide application specific media types a converter class has to be registered. All service classes are also registered.

We divided our implementation into multiple packages. The important ones are: *storage, states, services and models.* The storage package contains the in-memory storage for the resources. Since we only use hash maps to map IDs to resources we do not go much into detail here.

The services package contains all service classes which provide the endpoints for requests. An arriving request will be handled by the corresponding method of

[1]https://jersey.java.net

the service class. Each method invokes the corresponding state of the ε-NFA and starts the processing of the request. The state returns the requested response and the service class forwards it to the client.

The states package contains the ε-NFA of the server. Therefore, we implemented four abstraction layers. *AbstractState* is the first layer. It defines the default methods every state must have. The next layer adds caching funtionality to the states. The third layer contains the abstract classes for different http methods and the last layer contains all concrete classes. Every resource has its own states for each http method. Each concrete state class overwrites the method *define-TransisitionLinks()* that defines all hypermedia links the client can reach from this state.

The models package contains all resource models as well as additional views. For example, the user resource has two different views: one including the password and another one without password. Every view extends the *AbstractView-Model<T extends AbstractModel>* class. A view is a decorator that decorates the underlying model. All getter and setter methods use the underlying getter and setter methods except those for password in case of the user without password view. The *ViewConverter* class is used to convert an object of a model into a given view. The *ViewMerger* class is used to merge the latest representation of the server with the representation attached to a writing request.

3.2.2 The Client Side

The client side is also divided into multiple packages. The important ones are: *cache, offline and sync*. The cache package contains all classes needed for caching. The offline package contains all functionality for offline execution of requests. Finally, the sync package contains the code for data synchronization between client and server.

According to our hierarchy in Section 2.5.2 the caching has to be persistent. However, we did not implement a persistent cache within the prototype since we would not achieve new knowledge from it for this thesis. The cache is implemented as in-memory solution and stores all JSON representations as key value maps.

The offline package contains the ε-NFA of the proxy on client side. Since we tried to avoid code duplication on the client side the ε-NFA is a generic one. The client side also has four abstraction layers. The layers are analogue to the server side: the first layer with all methods each state requires; the second layer for caching; the third layer to define different behavior for each http method and finally, the concrete state classes that use the in-memory cache. In contrast to the server side we do not need concrete state classes for every resource since we use a generic ε-NFA. In addition to the state machine this package also contains the client models. Those are finite sets of key value pairs and can be serialized and deserialized by *genson*. The paging functionality for queries is also implemented within this package.

The sync package contains the synchronization handlers as well as the conflict handlers for data synchronization between the server and client side. Every http method has its own synchronization handler. For example, a post request has to fetch the representation of the created resource after it was executed. The representation contains additional information like the actual ID and the etag. Afterwards, the following put or delete request has to be updated in order to use the actual etag for modification. Moreover, a put request has to replace the etag of the following put or delete request as well. If conflicts occur during synchronization the conflict handlers are triggered. The conflict handler has to resolve the conflict and thus update the current request to allow execution. The handler can also decide to drop the current request.

A client application using the proxy layer may have cached temporary links. The proxy has to replace temporary links in requests if the corresponding resource has already been synchronized. Therefore, the proxy has to know the mapping of temporary links to actual links.

In case of failure the conflict handler can not recover. A *CouldNotRecoverException* will be thrown. After catching the exception a callback to the application using the proxy layer will be executed. The callback gets access to the whole queue containing all requests executed offline. The application can then decide what should be done. Thus, the offline data will not be lost upon failure.

Since our prototype is written in java and only tested using test cases a class is needed to decide whether the proxy is online. Therefore, we defined an interface *NetworkStatus* with the method *isOnline()* which is used to determine whether the proxy is in offline mode. Within the test cases we can thus easily switch between online and offline mode without emulating flaky internet connections.

3.3 Additional Approaches

This section will explain two additional approaches for enabling offline support for RESTful systems. Since we could not implement prototypes for every approach within this thesis, these approaches need additional thoughts and verification.

3.3.1 Using Code-On-Demand

Fielding (2000) defined the code-on-demand constraint as optional for RESTful systems. Modern web pages use this code-on-demand functionality by retrieving javascript from the servers. However, this constraint allows to enable offline support in RESTful systems. The client asks the server to deliver the code needed to work offline.

This approach allows to enable offline support on the fifth level of our hierarchy. Although the code is placed on the backend, it is very application specific. Moreover, the developer has to implement the features both for the client side and the server side.

This approach cannot be used for every use case. Reloading code during runtime is not allowed on every platform. iOS applications are not allowed to download additional code that adds functionality[2]. Therefore, this approach can be used within a company that can deploy the app without app store for their employees. However, if an app should be sold in app store, this approach is not applicable.

3.3.2 Using Servers Emitting Templates

Amundsen introduced another approach to implement hypermedia clients in his talk at NDC Oslo[3]. Within this talk he explained how to eliminate code from the client. Since code on the client produces tight coupling with the service Amundsen tries to reduce the amount of code needed on the client. He moves specific knowledge of addresses, inputs and workflow out of the client app and places it into the message. Therefore, all relevant information is emitted by the server. Amundsen implemented a hypermedia client with hypermedia templates. He got the server to emit templates and the client to use the server's templates.

We had the idea to also use those templates for offline support. Amundsen's templates contain a description of all functionality that can be done with a given resource. Adding information for the client about what to do in offline mode would be possible. However, the client would still need code for storing the data persistent or queueing requests. Furthermore, this approach does not eliminate the known challenges, although user rights could be handled more easily, since the server could consider them during template creation. Another drawback is the client would only have access to the templates that were cached while working online. Moreover, the offline support would no longer be hidden from the backend.

[2]https://developer.apple.com/app-store/review/guidelines/
[3]http://amundsen.com/talks/2015-06-ndcoslo/

3.4 Offline Support for Microservices

This thesis focuses on offline support for human-driven hypermedia clients. However, we had some thoughts on machine-driven hypermedia clients. Amundsen distinguishes between those two kinds in his upcoming book *Learning Client Hypermedia*, since both kinds have different challenges[4]. Compared to our prototype, intelligent merging strategies are needed for machine-driven hypermedia clients. Our prototype uses a callback function to ask the user of the application for help if the conflict cannot be resolved by itself.

Microservices are often designed as RESTful systems. Most of them provide an API that can be used by applications. However, there are microservices that are also used by other microservices. For example an authentication service will be used by other microservices rather than by the user application itself. Enabling offline support for microservices is different for human-driven and machine-driven hypermedia clients. Moreover, offline support might not be reasonable for every microservice.

[4]http://amundsen.com/blog/archives/1157

Appendix A Deployment Manual

The implementation of this thesis contains a fully functional server, a client proxy layer and test cases using this proxy layer. The server as well as the proxy layer only work in-memory, no data is stored persistently. In order to run the test cases properly the server has to be deployed on an application server like tomcat. Since the project is a maven project it can easily be done by maven. The *pom.xml* is configured to deploy the backend on the application path */todolist*. We are using the tomcat 7 maven plugin from apache.

The command to deploy the backend for the first time is *mvn tomcat7:deploy*. Afterwards, the backend has to be redeployed, so just use *mvn tomcat7:redeploy* instead. Please note that it is necessary to have a correct *settings.xml* file in the .m2 directory. Listing 3.1 shows our *settings.xml* that worked for deployment via maven.

Listing 3.1: Example of settings.xml for Maven.

```
<settings>
 <profiles>
  <profile>
   <id>compiler</id>
    <properties>
     <JAVA_1_8_HOME>PATH-TO-JAVA-HOME</JAVA_1_8_HOME>
    </properties>
  </profile>
 </profiles>
 <activeProfiles>
  <activeProfile>compiler</activeProfile>
 </activeProfiles>
 <servers>
  <server>
   <id>tomcat-localhost</id>
    <username>username</username>
    <password>12345</password>
  </server>
 </servers>
</settings>
```

After the deployment of the backend on localhost the test cases can be executed. If the backend is not deployed on localhost the base url in the class *TestBase.java* has to be replaced.

Appendix B Content of the CD

The CD contains the source code of this thesis as well as the source latex files. Therefore, two folders exist: *latex* and *source*. The source code can be deployed as described in A. The PDF file of this thesis was compiled using TeXLive-2016 as system-wide TeX distribution on Mac OS X El Capitan.

References

Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., & Welch, B. (1994, December). The bayou architecture: support for data sharing among mobile users. In *Mobile computing systems and applications, 1994. wmcsa 1994. first workshop on* (pp. 2–7). doi:10.1109/WMCSA.1994.37

Fielding, R. (2000). *REST: architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California, Irvine).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gonçalves, E. E. M. & Leitão, A. M. (2007, September). Offline execution in workflow-enabled web applications. In *Quality of information and communications technology, 2007. quatic 2007. 6th international conference on the* (pp. 204–207). doi:10.1109/QUATIC.2007.28

Gonçalves, E. E. M. & Leitão, A. M. (2009). Using common lisp to prototype offline work in web applications for rich domains. In *Proceedings of the 6th european lisp workshop* (pp. 18–27). ELW '09. Genova, Italy: ACM. doi:10.1145/1562868.1562871

Handy, J. (1998). *The cache memory book*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Academic Press. Retrieved from https://books.google.de/books?id=-7oOlb-lCpMC

Richardson, L., Amundsen, M., & Ruby, S. (2013). *Restful web apis*. O'Reilly Media. Retrieved from https://books.google.de/books?id=ZXDGAAAAQBAJ

Riva, C. & Laitkorpi, M. (2009). Service-oriented computing - icsoc 2007 workshops: icsoc 2007, international workshops, vienna, austria, september 17, 2007, revised selected papers. In E. Nitto & M. Ripeanu (Eds.), (Chap. Designing Web-Based Mobile Services with REST, pp. 439–450). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-540-93851-4_42

Satyanarayanan, M. (2002, May). The evolution of coda. *ACM Trans. Comput. Syst. 20*(2), 85–124. doi:10.1145/507052.507053

Schreibmann, V. & Braun, P. (2015, May). Model-Driven Development of RESTful APIs. In *Proceedings of the 11th international conference of web infor-*

mation systems and technologies (pp. 5–14). INSTICC. Lisbon, Portugal. SciTePress.

Tilkov, S. (2013). *REST und HTTP : Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt. Retrieved from https://books.google.de/books?id=pJF-ngEACAAJ

Webber, J., Parastatidis, S., & Robinson, I. (2010). *Rest in practice: hypermedia and systems architecture*. Theory in practice series. O'Reilly Media. Retrieved from https://books.google.de/books?id=1D24-cGQRdsC

Zuzak, I., Budiselic, I., & Delac, G. (2011). Formal modeling of restful systems using finite-state machines. In S. Auer, O. Díaz, & G. A. Papadopoulos (Eds.), *Web engineering: 11th international conference, icwe 2011, paphos, cyprus, june 20-24, 2011* (pp. 346–360). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-22233-7_24