2015 12th Working IEEE/IFIP Conference on Software Architecture

A conversation based approach for modeling REST APIs

Florian Haupt, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

Cesare Pautasso
Faculty of Informatics
University of Lugano (USI)
Lugano, Switzerland
cesare.pautasso@usi.ch

***Abstract*—Conversations are a well-known concept in service design to describe complex interactions between a client and one or multiple services. The REST architectural style constrains the characteristics of clients, servers and their interactions in REST architectures which consequently has an impact on conversations in such systems. The relation between conversations and REST architectures and how such RESTful conversations can be characterized has not been studied in detail yet. In this paper we discuss the characteristics of conversations in REST architectures and introduce an initial set of commonly used conversation types. Based on this, we propose to use conversations as a modeling tool for the design of REST APIs at a higher level of abstraction. We also introduce a corresponding interaction centric metamodel for REST APIs. The characterization of RESTful conversations enables a new interaction centric viewpoint on REST architectures which can be also applied for modeling REST APIs on an abstraction level that enables users to focus on the essential functionality of their REST API.**

Keywords—REST; Conversation; Model Driven Design

I. INTRODUCTION

Web services following the Representational State Transfer (REST) architectural style [1] publish a set of related resources that clients can discover following hyperlinks and interact with according to their uniform interface. Linking resources implies that clients will typically perform multiple interactions to achieve their goal and bring the application to a new stable state. It is possible to use the well-known concept of service conversation, borrowed from messaging systems [2], to indicate a set of basic HyperText Transfer Protocol (HTTP) request-response interactions that are driven by the same client interacting with one or more RESTful Web services.

In this paper we study the specific characteristics of RESTful conversations and introduce the concept of conversation type, which can be used to identify a set of interactions with a predefined structure. These recurring conversation types are commonly found in the field and go from relatively simple indirect resource lookups, or long running operations to more complex structures, such as the one used with collection resources and in the Try-Confirm/Cancel protocol for achieving atomicity in distributed transactions involving multiple REST APIs [3]. Conversation types play also a very useful role when modeling and describing the interface of a RESTful Web service at a higher level of abstraction, since, as we are going to show, they help to reduce the amount of details that need to be specified during the design of a RESTful API.

The rest of the paper is structured as follows. In section II we will introduce the concept of RESTful conversations and in section III we will show some types of such conversations. Section IV introduces an interaction centric metamodel for REST APIs whereas section V extends this to a conversation centric metamodel. The transformation between these models is discussed in section VI. Section VII gives an overview about relevant related work and section VIII closes the paper with a conclusion and outlook.

II. RESTFUL CONVERSATIONS

A conversation typically denotes a set of communication activities between two or more participants. In the context of REST we focus on the communication between a client and a RESTful Web API, i.e. a set of resources. Conversations between a client and a set of resources, what we call *RESTful conversation*, can be characterized as follows.

There are two types of *participants* in a RESTful conversation. *Clients* are interacting with an API to fulfill a certain goal. *Resources* are the building blocks of each RESTful Web API; they provide a uniform interface enabling to access and modify their state. An interaction between a client and an API may result in the creation or deletion of its resources, or in the retrieval and update of the representation of its resources.

The *communication primitives* used in a conversation are given by the uniform interface of the REST architecture. In the case of APIs making use of the HTTP protocol, each communication is initiated by the client and consists of a request followed by a response message. Together with the resource identifier, each request message includes the HTTP verb (e.g., GET, PUT, POST, DELETE) defining the operation to be performed on the resource.

Each basic communication is *stateless*, i.e. its successful processing by the server does not rely on any previous communication. All relevant data, i.e. the state, is contained in the message. The whole conversation comprising multiple

978-1-4799-1922-2/15 $31.00 © 2015 IEEE
DOI 10.1109/WICSA.2015.20

basic communication rounds may be stateful. The state of the conversation (indicating the progress within the conversation) is maintained and managed by the client. The course of the conversation is determined by the client (which is responsible for initiating the next request/response communication round with the selected resource) but can be influenced by the server (which may reply with one or more hyperlinks – or hypermedia controls [4] and affordances [5] – embedded within a resource representation or the corresponding metadata).

Resources can *redirect* a client interacting with them to other resources. This characteristic follows from the *Hypertext as the Engine of Application State* (HATEOAS) constraint of the REST architectural style. As a consequence, the course (or all possible courses) of a RESTful conversation is controlled by the resources and dynamically discovered by the client involved within the conversation. Whenever a client interacts with a resource, this resource is either the starting point of a conversation or the client has been forwarded to this resource by following the hyperlinks embedded in the representation of another resource.

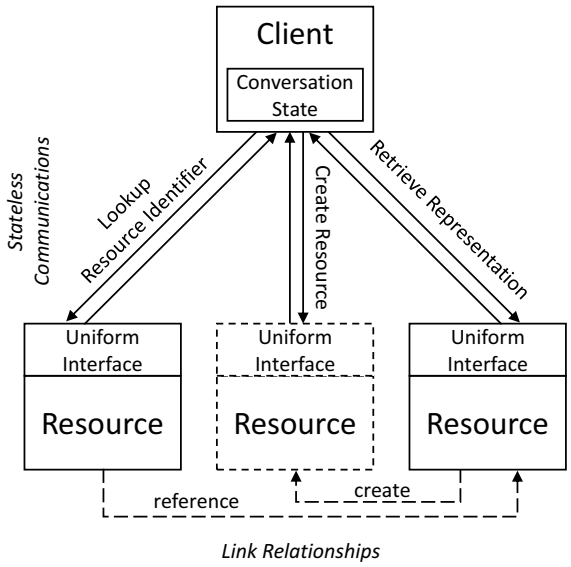

Fig. 1. RESTful conversation

The common characteristics of RESTful conversations are sketched in Fig. 1. To summarize, a RESTful conversation is a conversation between a client and one or more resources. Each basic communication is stateless and based on the uniform interface. The course of the conversation is controlled by the resources following the HATEOAS principle and driven by the client that is responsible for triggering the next request-response round and choosing the hyperlink to be followed. During a conversation, resources may be created or deleted.

III. RESTful Conversation Types

In this section we have collected four examples of RESTful conversations that happen in practice. The goal is to show that conversations do play an important role in non-trivial real-world client/server exchanges and also to provide concrete examples for the evaluation of the model driven approach described later in the paper. The conversations are presented by showing the sequence (or possible sequences) of request/response communication activities listed in a log of the HTTP interactions and also visualized using UML sequence diagrams.

A. Redirect

This simple, but also a very fundamental conversation type describes the communication of a client with a resource which then redirects the client to another resource. This conversation type realizes one level of indirection and therefore reduces coupling.

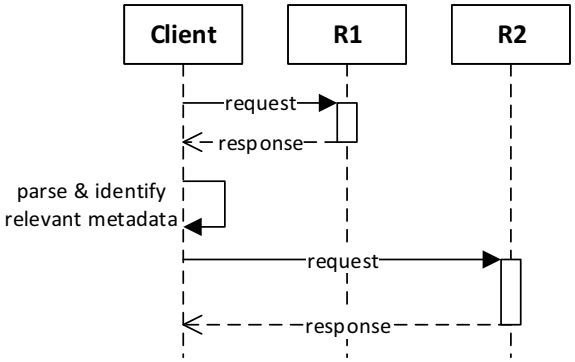

Fig. 2. Redirect conversation

The sequence of basic communications of a Redirect conversation is depicted in Fig. 2. The client first sends a request message to the resource R1. The response of R1 contains metadata redirecting the client to R2. The client reads the response, identifies the redirecting metadata and then sends a request to the resource R2.

```
GET /resource1 HTTP/1.1
```

```
HTTP/1.1 303 See Other
Location: /resource2
```

```
GET /resource1 HTTP/1.1
```

```
HTTP/1.1 200 OK
Link: </resource2>; rel="related"
```

```
GET /resource1 HTTP/1.1
```

```
HTTP/1.1 200 OK

<html>
  <a href="/resource2" rel="related">…</a>
</html>
```

Listing 1. Redirect examples

In HTTP based architectures there are multiple ways how this conversation type can be realized. Some examples are shown in Listing 1. After sending a request to the resource "/resource1", the response might contain the HTTP status code "303 See Other" [6], telling the client to issue the request again to the URL given in the "Location" header field. The resource might also respond with a "200 OK" status code together with a "Link" header field [7] containing the redirecting URL. Another realization of this conversation type is to deliver a

representation of the requested resource which then contains hyperlinks that redirect the client.

A common use case for redirect conversations are so called *home documents* of REST applications. The root resource of a REST application provides a set of links to the main resources of the application. The client accesses the root resource, selects an appropriate link, and then navigates to the linked resource. A practical example for this use case is given by the GitHub API. When accessing the root URL of the API, it provides the client with a set of URLs (and URL templates) pointing to resources realizing the core functionalities of GitHub, like for example user management or repository access. An excerpt of the home document of GitHub is shown in Listing 2.

```
{
 "current_user_url":
 "https://api.github.com/user",
 "repository_url":
 "https://api.github.com/repos/{owner}/{repo}",
 "team_url":
 "https://api.github.com/teams",
 "user_url":
 "https://api.github.com/users/{user}",
 "user_search_url":
"https://api.github.com/search/users?q={query}{
&page,per_page,sort,order}"
}
```

Listing 2. GitHub home document, excerpt from https://api.github.com/

B. Accessing Collections of Resources

Collection resources are a specific kind of resource which acts as a container for other resources. For example, the collection of products within a catalog, the pictures taken by a given user, the blog posts within a given year are all collections of resources of the same type. These resources can be listed by querying their collection. When created, they are added to the collection and conversely, when they are deleted they are removed from the collection. The main interactions of a conversation that manages a collection are shown in Fig. 3.

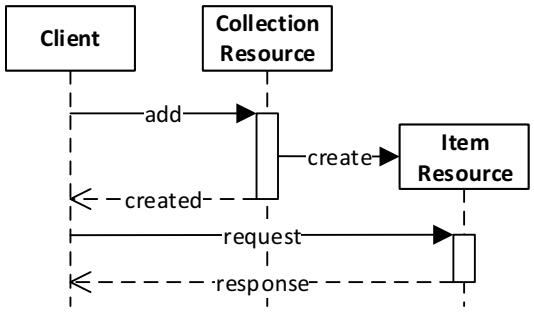

Fig. 3. Collection management conversation

The standard ATOMPUB protocol [8] defines how clients may perform such operations (addition, removal and enumeration) of entries within a collection. In ATOMPUB collections are named *feeds*, since the entries are considered with a temporal order, but the approach can be generalized/abstracted as in the following example conversations. An example for collection management using ATOMPUB is shown in Listing 3. A new entry is added to a collection. When a representation of the new entry is retrieved, a link to update it is provided (*edit* link relation). The link is followed by the client which updates the entry with a PUT request.

```
POST /blog HTTP/1.1
Content-Type: application/atom+xml;type=entry
Slug: my post
```

```
HTTP/1.1 201 Created
Location: /blog/my-post

<entry xmlns="http://www.w3.org/2005/Atom">
   <link rel="edit" href="/blog/my-post" />
</entry>
```

```
PUT /blog/my-post HTTP/1.1
```

```
HTTP/1.1 204 No Content
```

Listing 3. ATOMPUB entry creation and modification

For very large collections, it may be impractical for a service to return the entire index in a single response/representation. Thus clients may have to engage in a conversation with the service to retrieve the index and locate the resources within the collection they are interested in. As shown in Listing 4, the partial representation of a large collection will embed hyperlinks to the first, last as well as the next/previous set of entries. This way, clients can determine when they have scanned the entire collection and incrementally retrieve a manageable amount of entries.

```
GET /blog HTTP/1.1
```

```
<feed xmlns="http://www.w3.org/2005/Atom">
   <link rel="first" href="/blog" />
   <link rel="next" href="/blog/2" />
   <link rel="last" href="/blog/10" />
</feed>
```

```
GET /blog/2 HTTP/1.1
```

```
<feed xmlns="http://www.w3.org/2005/Atom">
   <link rel="first" href="/blog" />
   <link rel="prev" href="/blog" />
   <link rel="next" href="/blog/3" />
   <link rel="last" href="/blog/10" />
</feed>
```

Listing 4. Atompub large collection traversal

C. Try-Confirm-Cancel

The *Try-Confirm-Cancel* (TCC) pattern is used to design RESTful Web services that can participate in distributed atomic transactions [3]. Clients using them may temporarily change the state of a resource, e.g., when performing a booking request, and only later confirm the state transition, e.g., when all reservations have been successfully made. The TCC approach to distributed atomic transactions assumes that the resources that have been temporarily reserved will autonomously revert back to their original state, unless they are confirmed within a given timeframe. This way, if the client does not initiate the confirmation round the atomicity of the

distributed transaction will be guaranteed. Once the client begins the confirmation, it should use idempotent interactions that can be retried as many times as it is necessary to confirm all participant resources.

A TCC distributed atomic transaction between multiple resources can be also seen as a conversation involving a client interacting with multiple participant services that will first provide the client with a hyperlink referring to the temporary reservation resource and later receive either a confirmation (PUT) or cancellation (DELETE) request addressed to the same reservation resource. An example is shown in Listing 5. Participants that have independently timed-out will respond with a 404 status code, while if the confirmation is successful they will respond with 200.

Try	`POST /booking HTTP/1.1`
	`HTTP/1.1 302 Found` `Link: /booking/A; rel="tcc"`
Confirm	`PUT /booking/A HTTP/1.1`
	`HTTP/1.1 200 OK`
Cancel	`DELETE /booking/A HTTP/1.1`
	`HTTP/1.1 204 No Content`
Confirm after timeout	`PUT /booking/A HTTP/1.1`
	`HTTP/1.1 404 Not Found`
Cancel after timeout	`DELETE /booking/A HTTP/1.1`
	`HTTP/1.1 404 Not Found`

Listing 5. Try (POST) Confirm (PUT) Cancel (DELETE), including timeout

D. *Long running Requests*

In some scenarios it may be disadvantageous for clients to wait for their requests to be completely processed by the service since this may block their processing. It may also happen that a service is busy when the request arrives and it may want to delay its processing without keeping the client waiting for a potentially long time. To avoid dealing with network timeouts, which may occur for clients that wait for too long, it is possible to use the conversation shown in Listing 6.

The client sends the original request to the "job manager" resource, carrying a payload with the input data to be processed. The server will accept the request and respond immediately with a hyperlink referring to the "job resource" that the client can use to track the progress of the request. The client will periodically poll the given resource with a GET request, whose response will determine if the long running request has been completed. If the response is 200, the client must repeat the same request again; if the response is 303 the client will be redirected to another resource whose representation will contain the results of the long running request. Thus the client will follow the hyperlink and perform one last GET request to retrieve the output data.

As with every background form of processing, carried out asynchronously by the service, it is possible for clients to cancel it by issuing a DELETE request on the job resource identifier. Similarly, clients that have successfully retrieved the final results may want to DELETE them from the service.

Send request	`POST /job HTTP/1.1`
	`HTTP/1.1 202 Accepted` `Content-Location: /job/20150112`
Polling	`GET /job/20150112 HTTP/1.1`
	`HTTP/1.1 200 OK`
	`GET /job/20150112 HTTP/1.1`
	`HTTP/1.1 303 See Other` `Location: /job/20150112/output`
Read result	`GET /job/20150112/output HTTP/1.1`
	`HTTP/1.1 200 OK`
Cancellation	`DELETE /job/20150112 HTTP/1.1`
	`HTTP/1.1 204 No Content`
Cleanup	`DELETE /job/20150112/output HTTP/1.1`
	`HTTP/1.1 204 No Content`

Listing 6. Long running requests (with cancellation and cleanup)

This conversation thus covers the whole lifecycle of a long running request, from its creation to its completion and cleanup or cancellation. The main interactions of the conversation are summarized in Fig. 4. All aspects of the long running request (the request itself, its progress status, its results) are turned into a resource that the client can discover by following hyperlinks and interact with using the HTTP uniform interface.

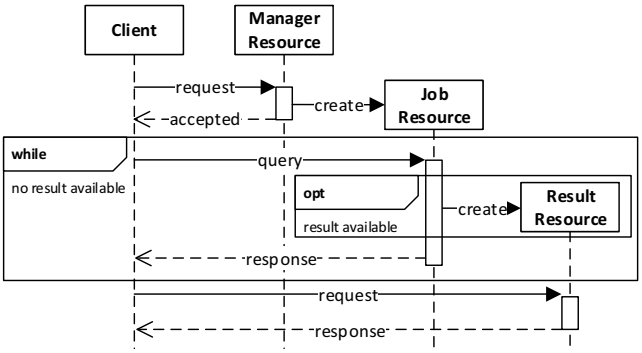

Fig. 4. Long running request conversation

```
POST /{AccountId}/vaults/
                {VaultName}/jobs
GET /{AccountId}/vaults/
                {VaultName}/jobs/{JobID}
GET /{AccountId}/vaults/
                {VaultName}/jobs/{JobID}/output
```

Listing 7. AWS Glacier job management requests

A real world example for a long running request conversation can be found in the AWS Glacier REST API[1]. Glacier is a cloud service for storing infrequently used "cold" data. Retrieving data archived in Glacier typically takes around

[1] http://docs.aws.amazon.com/amazonglacier/latest/dev/job-operations.html

3-5 hours[2] and is therefore realized as a long running request conversation. The interactions used for a data retrieval request are sketched in Listing 7.

IV. An Interaction Centric Metamodel for REST APIs

After discussing conversations in context of REST and introducing some RESTful conversation types in the previous sections, here we aim at applying conversation types for modeling REST APIs. In this section, we will introduce the basics of REST API modeling and motivate the idea to use conversation types as modeling tool for REST APIs.

The design and realization of REST APIs is a challenging task. There have been several studies conducted that show that most APIs calling themselves RESTful are in fact not [9][10][11]. The violation of constraints that define the REST architectural style in most cases leads to APIs that miss some of the desired quality attributes of REST compliant APIs like cacheability, scalability or loose coupling. As a consequence, new methods and techniques are needed that help service designers and developers to create REST compliant APIs.

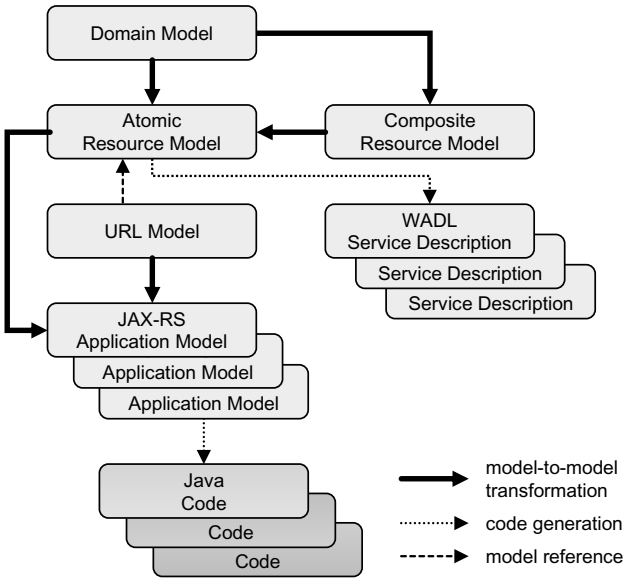

Fig. 5. Metamodel for MDSD based design and creation of REST APIs [12]

Our approach to ease the creation of RESTful APIs has been proposed in [12]. The main idea is to follow a model driven software design (MDSD) approach for the design and realization of REST APIs. The approach is based on a set of metamodels shown in Fig. 5. As a starting point a service designer can model a service in a domain specific and REST independent way (*Domain Model*). The domain model can then automatically be transformed into a resource model which can afterwards be refined and customized by the service designer. Alternatively, a service designer can also start modeling the resource model without providing a domain model. For the resource modeling, two different models have been defined, the *Atomic Resource Model* and the *Composite Resource Model* (we will discuss them soon).

One important difference between the metamodel described in [12] and already existing metamodels for REST APIs is that the resource models in [12] do not specify any URLs. The definition of a URL structure for the resource model is contained in a separate model, the *URL Model*. Following this modeling approach, the definition of the resource model as well as the documentation of the modeled REST API generated from the model do only specify links between resources, but no specific URLs. Linking resources and then navigating through an API based on links realizes the HATEOAS constraint of the REST architectural style, an important feature to achieve loose coupling between client and server and also to enable the description of many conversation types.

The *atomic resource model* allows describing a REST API based on its fundamental (atomic) ingredients, like resources, methods or representations. The idea of the *composite resource model* is to allow aggregating (composing) multiple elements of the atomic resource model into new and coarser grained modeling constructs to enable modeling on a higher level of abstraction. In the following, we will extend and refine the work of [12] by introducing an *interaction centric metamodel* as atomic resource model as well as a *conversation centric metamodel* as composite resource model.

The interaction centric metamodel is shown in Fig. 6 as UML class diagram. The interaction centric metamodel comprises only the white elements, the red elements are part of the conversation centric metamodel extension that will be discussed later. One core entity of the metamodel is the *Resource*. A resource can have a name and be marked as being an entry resource. Entry resources are the starting point for interacting with a REST API, there has to be at least one entry resource defined for each API. The URLs of entry resources are supposed to be well known to all clients of an API, which then use these resources as a starting point to navigate through the API. Most REST APIs define exactly one entry resource, often called the root resource.

Each resource can support interactions based on any of the methods defined by HTTP, which are all (except CONNECT and TRACE) defined as separate entities in the metamodel. Interactions using these methods have some common characteristics but also some fundamental differences, the metamodel therefore defines a corresponding inheritance hierarchy. All interactions are derived from the common superclass *InteractionBase*. The *HEAD* and *OPTIONS* interactions are direct children of this class. Both methods do not support any request or response entity: request and response messages consist only of a HTTP header without any body. For all other methods (GET, PUT, POST, DELETE) the response message may contain an entity representation, represented by the *InteractionWithResponseEntity* class. Each interaction supporting a payload in the response message refers to one or multiple *representations* it supports. Each representation is of a specific *MediaType* and can be associated with a *Schema* describing its structure (e.g. an XML/JSON schema document) and an *Example* showing how a representation may look like (e.g. a JSON or XML document).

[2] http://aws.amazon.com/glacier/

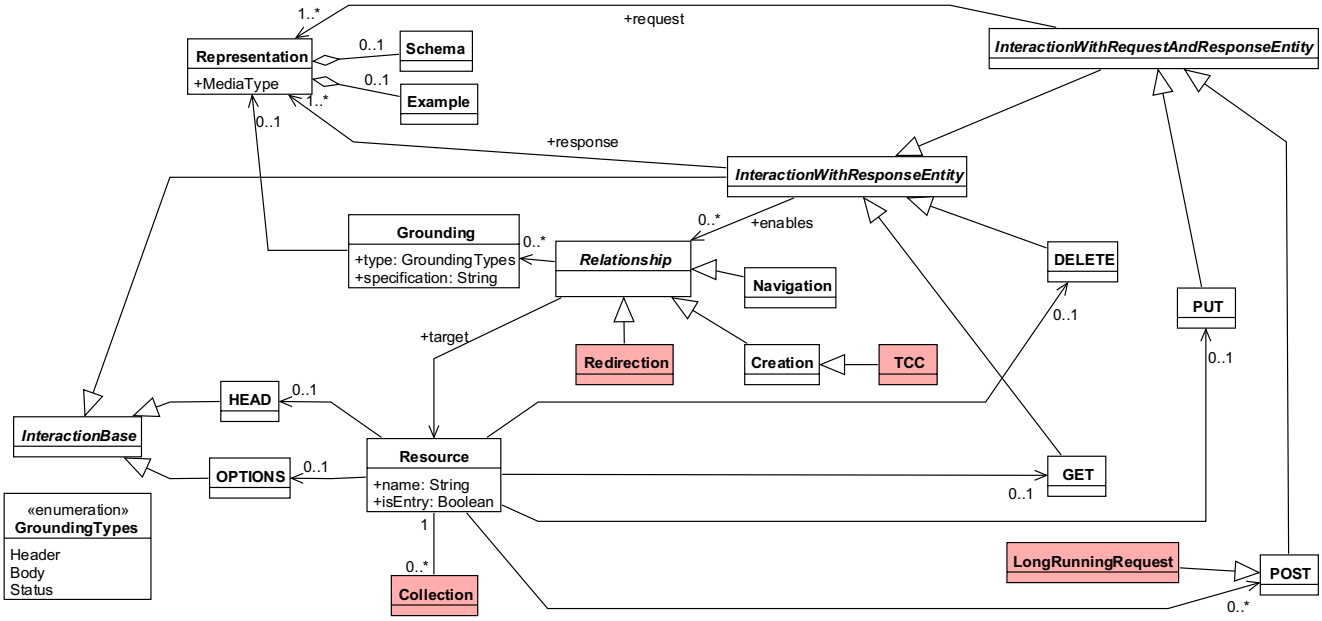

Fig. 6. Interaction centric Metamodel for REST APIs with conversation centric extension (elements of the conversation centric metamodel are highlighted)

Interactions based on PUT and POST do not only support to receive a response entity but also to send a request entity. The *InteractionWithRequestAndResponseEntity* class indicates this with the reference to the representations for the request.

The metamodel described so far allows for modeling resources together with the interactions they support. Another feature at least as important is the interconnection between related resources. Sending a POST request to a resource may result in the creation of another resource. Sending a GET request to a resource may return data that can be used to access other resources, typically by following hyperlinks. The relationship between two resources, e.g. that one resource can be used to create another resource or that the representation provided by one resource allows addressing another resource, must be realized by clients that interact with the source resource of the relationship within a conversation. Therefore, in our metamodel each interaction may be connected with a *Relationship*. The base relationship types defined in the metamodel are *Navigation* and *Creation*. Each relationship points to a resource which is the target of the relationship. The *Grounding* class can be used to provide additional information about how a relationship is realized. The conversation involving the navigation from one resource to another may be realized by sending a "303 See Other" status code, by using the Link header or by providing links in the representation sent in the body of the message (see also Listing 1). A grounding is specified by defining the *GroundingType* together with a specification, giving additional details for the selected grounding type, for example the name of the relevant header field. If the grounding is of the type "Body" then it is connected to the representations it refers to.

To demonstrate the application of our metamodel, Fig. 7 shows a model of a simple REST API supporting a collection conversation as introduced in section III B and also shown in Listing 3. Each interaction with the API starts at the *MyAPI* resource, the root (entry) resource of the API. A GET interaction (G1) with this resource returns a response in XML representation (RP1) which enables navigating (R1) to the *Blog* resource. The corresponding grounding (GR1) specifies that the hyperlink to the Blog resource is provided using a Link header. The Blog resource supports two interactions. The GET interaction (G2) returns an ATOMPUB representation (RP4)

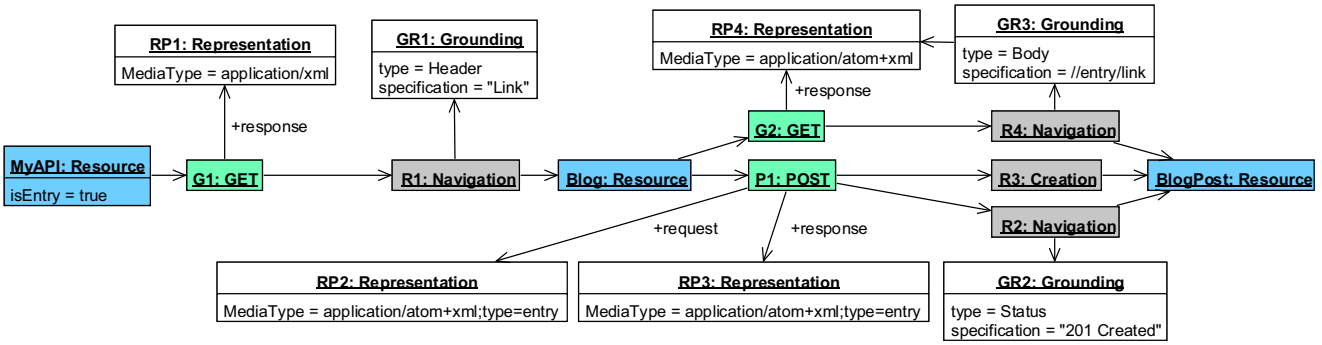

Fig. 7. Modeling a collection, interaction level

and thereby enables navigating (R4) to the *BlogPost* resource. The grounding (GR3) contains an XPath expression that specifies where in the representation of the Blog resource the hyperlinks for navigating to the BlogPost resources are contained. The second interaction supported by the Blog resources is a POST interaction (P1) enabling to create (R3) a BlogPost resource as well as to navigate (R2) to the created resource. The POST interaction can process ATOMPUB Entry Feed representations (RP2) and also returns the same type of representation (RP3). Navigating to a newly created resource is realized, as defined in the corresponding grounding (GR2), by returning a status code of "201 Created" which in turn means that there will also be a Location header containing a hyperlink to the created resource.

The interaction centric metamodel for REST APIs introduced in this section is defined as part of a model driven approach for creating REST APIs. The final goal of model driven software design is the ability to generate executable code out of a model of an application. The level of detail shown in our metamodel is required to achieve this goal while keeping flexibility for the modeler. A major drawback is, as seen in the example model shown in Fig. 7, that models soon become too complex and thus incomprehensible, a potential source for modeling errors. This altogether leads to the requirement for raising the modeling approach to a higher abstraction level, providing less but more powerful modeling constructs that ease modeling, improve intelligibility and hide repetitive details. In the following section, we will show how we do so by using RESTful conversation types.

V. Modeling REST APIs based on Conversations

Modeling a REST API has been so far described as the task of identifying the set of resources the API shall provide, how they are interconnected and which interactions each resource supports. However, if we broaden this perspective, we can also describe modeling as the task of identifying which conversation types a REST API shall support. Doing so, we are able to push the modeling task to a higher level of abstraction. As already demonstrated in section III, supporting a conversation in most cases comprises multiple resources and interactions. Therefore, when modeling in terms of conversations instead of modeling single resources and their basic interactions, the modeler of a REST API can focus on higher level capabilities of a REST API ("what conversations do I want to support") instead of lower level design details ("how do I support the conversations").

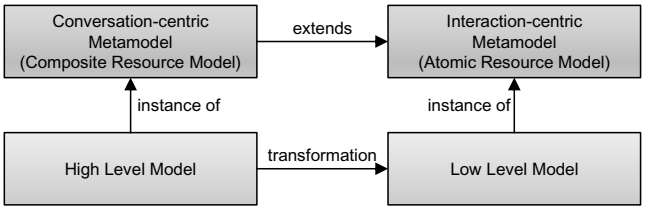

Fig. 8. Modeling approach summary

The main idea of the modeling approach introduced in this section is summarized in Fig. 8. By introducing conversations as modeling elements we allow for the creation of models on a high level of abstraction (high level model) which are instances of the conversation centric metamodel. The conversation centric metamodel is an extension of the interaction centric metamodel. High level models can be transformed into low level models which are instances of the interaction centric metamodel.

The example shown in Fig. 7 models a REST API supporting a collection conversation. The model comprises resources together with their supported interactions, which altogether enables the API to participate in a collection conversation. When switching from an interaction centric modeling approach to a conversation centric modeling approach, the model becomes far less complex as the corresponding example in Fig. 9 shows. In this model, we describe on a higher level that the *MyAPI* resource supports a collection conversation for a collection of *BlogPost* resources. Any details about the inner structure of the conversation like which additional resources are involved and which interactions are needed are hidden.

Fig. 9. Modeling a collection, conversation level

To enable modeling a REST API using conversations, we require a corresponding metamodel. In the example shown in Fig. 9 the modeling element *Blog* of the type *Collection* does not comply with the existing interaction centric metamodel. The model shown is a mix of known modeling elements as well as new, conversation specific modeling elements. Although we introduce conversation support as modeling construct we still need the ability to model, as before, resources and their interactions in disaggregated form. Parts of a REST API can be modeled using conversations whenever applicable, but it is always possible to resort to the basic elements of the interaction-based metamodel.

The extension of our interaction centric metamodel to also support conversation centric modeling is shown in Fig. 6. The conversation centric metamodel comprises both, the white elements of the original interaction centric metamodel as well the red extension elements. The *Collection* conversation is directly associated to a resource. A resource can support multiple collections and a collection points to exactly one resource (the resource the collection manages). Although a collection describes a relationship between two resources, it is not associated to any interaction (unless the *Navigation* and *Creation* relationships). In case of a collection conversation, the involved interactions are hidden by the high level modeling constructs. The Try-Confirm-Cancel (*TCC*) conversation extends the metamodel as a subclass of the Creation relationship. TCC describes the creation of resources, but as a tentative action that has to be confirmed, cancelled or that automatically times out. The *LongRunningRequest* conversation extends the metamodel as a specialization of a POST interaction. As POST interactions describe data processing requests in general, the long running request conversation is used for such requests that cannot (or should not) return an immediate response.

The transformation of conversation centric models, like the one shown in Fig. 9, into interaction centric models, like shown in Fig. 7, is an important part in our overall modeling approach that has not been discussed so far. In the following section we will introduce our template-based approach for this transformation.

VI. Templates and their Expansion

By introducing the conversation centric metamodel for REST APIs we enable service designers to create less complex and more understandable models on a higher level of abstraction (conversations instead of interactions). However, to integrate this metamodel into our model driven software development approach [12], we need to be able to transform conversation centric models into interaction centric models. In the following we will call this transformation an *expansion*, as it in general replaces a single modeling element by a set (or graph) of interconnected elements. One important aspect when designing this expansion is the observation, that there are typically multiple expansions possible for the same conversation. In section III we introduced some RESTful conversation types. For the *Redirect* conversation type we showed different ways to realize it (see Listing 1). The same applies for the *Collections* conversation type. The example shown in Listing 3 realizes collection management based on the ATOMPUB standard. However, it might have also been possible to realize the same conversation based on the Collection+JSON media type [13] or by using the XLink standard [14].

In this section we introduce the concept of *Templates* as a means to describe the expansion between the conversation centric and interaction centric metamodel. A template comprises all that is needed to transform a conversation type modeled in the high level model into a set of resources and interactions that realize this conversation. As there are in general multiple of such expansions possible, a template includes all these alternatives and also provides the possibility to select one of the applicable expansions.

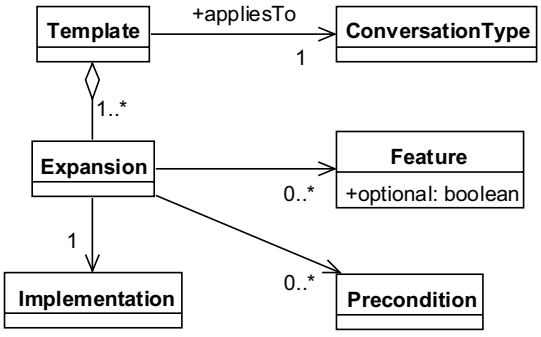

Fig. 10. Metamodel for templates

A formal description of the structure of a template is shown in Fig. 10 as UML class diagram. The transformation from elements of a conversation centric model into an interaction centric model is realized by an *Implementation* (which may be a piece of code, a XSLT stylesheet, graph grammar rules or any other artifact that implements the model transformation). The *Expansion* attaches additional information to an implementation, namely its *Features* as well as the *Preconditions* for its applicability. The *Template* in turn is a container for a set of expansions that all apply to the same *Conversation Type,* i.e. they all result in models supporting the same conversation type. The most important parts of the template are the *Preconditions* and *Features* each expansion may be associated with. In the following, we will discuss them in more detail.

A. Expansion Preconditions

One reason for having multiple expansions for the same conversation is that they might not always be applicable in all cases. An example for this is given by the Redirect conversation type and the example shown in Listing 1. If a redirection is realized by sending a corresponding status code like "303 See Other", there is exactly one target for the redirection. When realizing a redirection using the Link header it is however possible to include one or more hyperlinks, i.e. the redirection can have multiple (maybe alternative) targets to be selected by the client. Given that, a precondition for an expansion that realizes a redirection by sending appropriate status codes is that the redirection has exactly one target. In contrast, expansions realizing the redirection based on the Link header or based on hyperlinks in the representation would not be associated with this precondition.

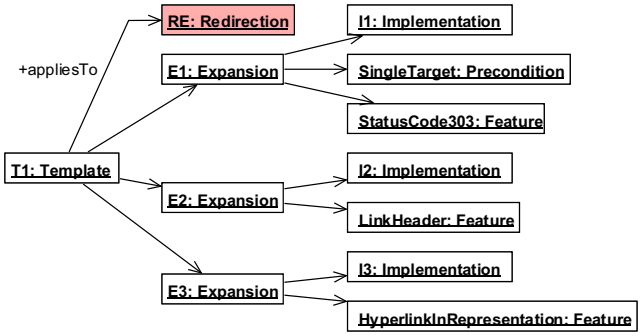

Fig. 11. Template for redirect conversation type

The general structure of the template for the Redirect conversation type is shown in Fig. 11. The template *T1* refers to the *Redirection* element which is part of the extended metamodel shown in Fig. 6. The template defines three expansions that correspond to the examples shown in Listing 1. As discussed before, the expansion based on using status codes for redirection is associated with a Precondition whereas the other expansions are not.

B. Expansion Features

Whereas the concept of preconditions is needed to determine which expansions are applicable at all, features can be used to select and configure one of multiple applicable expansions. In case of the redirection conversation and as shown in Fig. 11, the expansion realizing the redirection using the Link header may be associated with the feature "Link header" and the expansion realizing the redirection based on status code may be associated with the feature "status code

303" (the "status code 303" feature includes that the Location header contains the redirecting link, this is already set by the HTTP specification and therefore not modeled as a separate feature like the "Link header"). During the application of a transformation the information given by the features may then be used to select one of multiple applicable expansions.

As shown in the template metamodel in Fig. 10, features may also be marked as being optional. When applying an expansion with optional features, these features can be activated or deactivated and thereby configure the way the expansion is performed. The example of a collection management conversation shown in Listing 3 is realized based on the ATOMPUB standard, which can be described as a feature of the expansion. The same expansion may in addition realize the collection conversation using the Collection+JSON media type, enabling a client to decide by content negotiation which representation it wants to access. These two different representations for collections, ATOMPUB and Collection+JSON, can be described as optional features. The structure of the corresponding template is shown in Fig. 12. When applying the template, it can be selected if either both representations or only one of them should be generated.

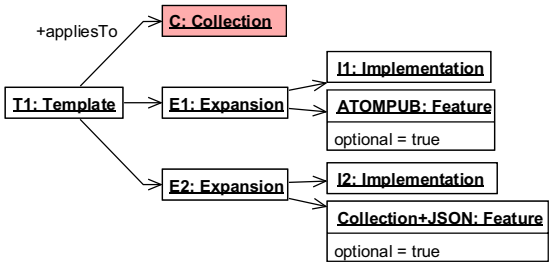

Fig. 12. Template for collection management conversation type

C. Applying Templates

After introducing the general structure of templates and giving some examples, we will now shortly discuss how templates are applied to a conversation centric model of a REST API. As described in the metamodel for templates (Fig. 10), a template applies to exactly one conversation type. In addition we assume that there exists exactly one template defined for each conversation type.

For each conversation type that occurs in the model of a REST API, the corresponding template is selected. In the first step, the preconditions for each of the expansions of the template are evaluated. Expansions with unfulfilled preconditions are discarded. In the second step, one of the remaining expansions has to be chosen. The decision can be based on the set of features provided by each expansion; the decision can be made by a human user or by any selection criteria. After an expansion has been selected, it has to be checked if there are any optional features associated with it. For each optional feature it has to be defined, if the feature shall be realized by the expansion or not. In the last step, the implementation for the selected expansion is retrieved and applied to the model. The decision about optional features that shall be realized by the expansion is passed to the implementation of the expansion as input parameters.

D. Realizing Expansions

For the realization of the expansions we use attributed graph grammars [15], a mature and well-understood technique often used for model transformations [16]. Attributed graph grammars define graph transformations as replacements rules based on typed graphs. Before the rules of an attributed graph grammar can be defined, a type graph has to be created representing the element types that may be part of a graph. In our work, the type graph corresponds to the metamodel shown in Fig. 6. Afterwards, for each expansion an associated graph grammar rule has to be defined. We use the AGG tool[3] for defining and executing the graph transformation rules. AGG allows for graphical modeling of both, the type graph as well as the transformation rules. Another advantage of AGG is that it already allows the definition of preconditions that can be associated with transformation rules, a convenient way to realize preconditions associated with expansions.

VII. Related Work

Thanks to hypermedia and the uniform interface (e.g., idempotent receiver semantics are implicitly given), applying conversations to REST gives an elegant solution to some of the conversation description challenges identified by [17], where the importance of conversations and the need for services to describe the supported conversation types was originally discussed in the context of messaging middleware. A good starting point showing how conversations were introduced for traditional WSDL-based services is [18], where the authors develop the concept starting from a survey of e-commerce Web portals, which were however analyzed abstracting away the underlying HTTP interactions. More recently, the need for an architecture-centric approach to deal with the complexity of consistently configuring message-based service systems was illustrated in [19]. The authors define a message-centric extension for the xADL architectural description language and describe how to generate the corresponding message routing configuration for the specific message bus.

Model driven service engineering for RESTful APIs has been introduced in [20] defining an extensive metamodel comprising structural as well as behavioral aspects. The applicability of UML for the model driven development of REST APIs is demonstrated in [21]. The authors present a modeling approach tightly integrated with UML that is based on a REST specific UML profile and is tailored to Java EE environments. A complementary work on model driven development for REST is presented in [22], proposing an iterative approach for defining and improving model transformations. In addition to model driven approaches for REST APIs there has been developed several description languages for REST APIs. The *Web Application Description Language* (WADL) [23] has been developed as the REST counterpart to the Web Services Description Language (WSDL) [24] but has never been significantly adopted in practice. Current state of the art approaches for describing REST APIs include the *RESTful API Modeling Language* RAML[4] dedicated to a technical description approach and

[3] http://user.cs.tu-berlin.de/~gragra/agg/index.html
[4] http://raml.org/spec.html

Swagger[5] which enables the automated generation of user friendly API documentation.

VIII. Conclusion and Future Work

This paper introduced the concept of RESTful conversations, whereby multiple request/response interactions of one client with one or more resources published by various RESTful Web APIs are considered as a whole. RESTful conversations emerge from the navigation of a client within a Web of hypermedia relationships. Clients drive forward the progress of the conversation by issuing additional request messages, while services may influence the course taken by the client by embedding hyperlinks to related resources in the representations sent as part of the responses. Some conversations have a regular structure, which can be abstracted into a conversation type. In this paper we have collected four real-world conversation types, which can be realized with different concrete HTTP request-response interactions.

The main contribution of this paper lies in the use of RESTful conversations for modeling purposes. In particular, the description and specification of a RESTful Web API can be greatly simplified by using conversations. The second part of this paper shows how a model driven approach for describing and realizing RESTful Web APIs can be extended to support conversations. This way, the abstraction level of the service descriptions is raised, the corresponding model gets significantly simplified and the modeler of a REST API can focus on higher-level capabilities of a REST API ("what conversations do I want to support") instead of lower level design details ("how do I support the conversations"). The conversation centric model can then be automatically expanded into a fine-grained and interaction centric model using graph transformation techniques. The generated model will be further refined and extended to drive the code generator to build a service that can participate in different types of RESTful conversations.

The work on RESTful conversations presented in this paper assumes one client interacting with one REST API. For future work this might be extended to also cover scenarios comprising multiple clients as well as conversations including multiple REST APIs. Another aspect that can be elaborated in addition is the use of callback mechanisms as part of a conversation. The question on how to model APIs which combine together multiple basic conversation types also remains open. The conversation type examples shown in this paper can be used as a starting point for creating an extensive collection of conversation types supported by todays REST APIs. It would be interesting to investigate if such a collection can serve as a first step towards a (RESTful) conversation pattern language.

Acknowledgment

The authors would like to thank Gregor Hohpe, Silvia Schreier, and Guy Pardon for their valuable feedback.

[5] http://swagger.io/

References

[1] R.T. Fielding and R.N. Taylor, "Principled design of the modern Web architecture", ACM Trans. Internet Technol. 2, May 2002: 115-150.

[2] G. Hohpe and B. Woolf, "Enterprise integration patterns: Designing, building, and deploying messaging solutions", Addison-Wesley Professional, 2004.

[3] G. Pardon and C. Pautasso, "Atomic Distributed Transactions: a RESTful Design", WS-REST, 2014.

[4] J. Webber, S. Parastatidis and I. Robinson, "REST in Practice", O'Reilly, 2010.

[5] R. Verborgh, M. Hausenblas, T. Steiner, E. Mannens, and R. van de Walle, "Distributed affordance: An open-world assumption for hypermedia", WS-REST, 2013.

[6] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, 2014, http://www.ietf.org/rfc/rfc7231.txt.

[7] M. Nottingham, "Web Linking", RFC 5988, 2010, http://www.ietf.org/rfc/rfc5988.txt.

[8] J. Gregorio and B. de Hora, "The Atom Publishing Protocol", RFC 5023, 2007, http://www.ietf.org/rfc/rfc5023.txt.

[9] D. Renzel, P. Schlebusch, and R. Klamma, "Today's top 'RESTful' services and why they are not RESTful", WISE, 2012.

[10] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating web apis on the world wide web", ECOWS, 2010.

[11] P. Adamczyk, P. H. Smith, R. E. Johnson, and M. Hafiz, "REST and Web services: In theory and in practice", REST: from Research to Practice, Springer New York, 2011.

[12] F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A model-driven approach for REST compliant services", ICWS, 2014.

[13] M. Amundsen, "Application/vnd.collection+json", IANA Media Type Registration, https://www.iana.org/assignments/media-types/application/vnd.collection+json

[14] S. DeRose, E. Maler, D. Orchard and N. Walsh, "XML Linking Language (XLink) Version 1.1", W3C Recommendation, http://www.w3.org/TR/xlink11/.

[15] G. Rozenberg and H. Ehrig, "Handbook of graph grammars and computing by graph transformation", World Scientific Publishing Company, 1999.

[16] H. Ehrig, U. Prange, and G. Taentzer, "Fundamental theory for typed attributed graph transformation", Graph transformations, Springer Berlin Heidelberg, 2004. 161-177.

[17] G. Hohpe, "Let's have a conversation", Internet Computing, IEEE 11.3 (2007): 78-81.

[18] B. Benatallah, F. Casati, and F. Toumani, "Web service conversation modeling: A cornerstone for e-business automation", Internet Computing, IEEE 8.1 (2004): 46-54.

[19] C. Dorn, P. Waibel, and S. Dustdar, "Architecture-Centric Design of Complex Message-Based Service Systems", Service-Oriented Computing, Springer Berlin Heidelberg, 2014. 184-198.

[20] S. Schreier, "Modeling RESTful applications", Proceedings of the Second International Workshop on RESTful Design. ACM, 2011.

[21] R.V.V. Sanchez, R.R. de Oliveira, and R. Pontin de Mattos Fortes, "RestML: Modeling RESTful Web Services", REST: Advanced Research Topics and Practical Applications. Springer New York, 2014.

[22] M. Siikarla, M. Laitkorpi, P. Selonen, and T. Systä, "Transformations have to be developed ReST assured", Theory and Practice of Model Transformations , Springer Berlin Heidelberg, 2008.

[23] M. Hadley, "Web Application Description Language", W3C Member Submission, http://www.w3.org/Submission/wadl/.

[24] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1", W3C Note, http://www.w3.org/TR/wsdl

All links were last followed on 27.02.2015