Toward Data-driven Programming for RESTful Linked Data

Steffen Stadtmüller, Andreas Harth

Institute of Applied Informatics and Formal Descriptions Methods (AIFB)
Karlsruhe Institute of Technology, Germany
{firstname.lastname}@kit.edu

Abstract. Applications are increasingly focused on the use and manipulation of data resources distributed on the Web. Consequently REST gains popularity with its resource-centric interaction architecture and flexibility enabled by hypermedia controls, i.e., links between resources. The natural extension of Linked Data with RESTful manipulation possibilities for resources can bring advantages that can be leveraged to avoid a manual ad-hoc development of mashups. We propose a lightweight declarative rule language with state transition systems as formal grounding that enables the development of data-driven applications build upon the RESTful manipulation of Linked Data resources. We address the problem of developing a scaleable programming framework for Linked Data resources, that retains the advantages of the loose coupling fostered by REST.

1 Introduction

The Linking Open Data community has gained momentum over the last years with the trend towards opening up public sector and other data [2]. At the same time there is a strong movement in the Web community toward a resourceful model of services based on Representational State Transfer (REST [7]) which propagates the primacy of loose coupling. Flexibility, adaptivity and robustness are direct consequences from the loose coupling of REST and are particularly useful for software architectures in distributed data driven environments such as the Web [18].

According to the Richardson maturity model [19] REST is identified as the interaction between a client and a server based on three principles:

- the use of URI-identified resources
- the use of a constrained set of operations, i.e., the HTTP methods, to access and manipulate recourse states
- the application of hypermedia controls, i.e., the data representing a resource contains links to other resources. Links allow a client to navigate from one resource to another during his interaction.

The Linked Data design principles[1] also address the use of URI-identified resources and their interlinkage. However Linked Data is only concerned with the provisioning and retrieval of data. An extension of Linked Data with REST to allow for resource manipulation is therefore natural, and enables a lightweight, flexible distributed programming model on the web.

Following the motivation to look beyond the exposure of fixed datasets, the extension of Linked Data with REST technologies has been proposed and explored for some time [1, 27] and led recently to the establishment of the *Linked Data Platform*[2] W3C working group.

Especially in data driven scenarios an increased value comes from the combination of data and the functionality to manipulate them. The increased value of such compositions is reflected in the constant growth of the mashup ecosystem of web APIs [26].

In a REST architecture, client and server are supposed to form a contract with content negotiation, not only on the data format but implicitly also on the semantics of the communicated data, i.e., an agreement on how the data have to be interpreted [25]. Since the agreement on the semantics is only implicit, programmers developing client applications have to manually gain a deep understanding of the provided data, often based on natural text descriptions.

The combination of RESTful resources originating from different providers suffers particularly from the necessary manual effort to use them. The reliance on natural language descriptions has led to mashup designs in which programmers are forced to write glue code with little or no automation and to manually consolidate and integrate the exchanged data.

On the other hand, traditional service composition approaches that aim to decrease the manual effort lead to a tight coupling between client and server, i.e., they sacrifice flexibility and are prone to failures due to server-side changes. Traditional composition approaches often fail to leverage links between resources and do not provide straightforward mechanisms to dynamically react to state changes of resources. The reaction on state changes becomes especially important in a distributed programming environments, since a client can not ex ante predict the influence of other clients on the resources, i.e., REST does not allow a client to make assumptions on resource states.

We propose to leverage the combination of REST with Linked Data for a data and resource driven programming approach that enables a straight-forward development of applications build on semantic web resources. The main goal of the programming approach is a high degree of automation and the preservation of loose coupling by

- leveraging links between resources (i.e., hypermedia controls) provided by Linked Data
- specifying desired interactions dependent on resource states, which is enabled by a uniform state description format, i.e., RDF.

[1] `http://www.w3.org/DesignIssues/LinkedData.html`
[2] `http://www.w3.org/2012/ldp/charter`

The development of applications in a REST framework is especially challenging, since the hypermedia controls and the resource states can only be determined during runtime, however, programmers have to specify their desired interactions at design time. A further requirement for our programming approach in a web based environment is a fast and scaleable execution of the applications: A rapid interaction with resources from many different providers has to be possible to allow for the development of useable web based applications. Further, we want to cover a great variety of application and communication scenarios.

Example 1. Lin et al. [14] propose a scenario where the search for information objects identifies additional actions that can be performed on the objects. E.g., the search for a movie presents actions such as reading reviews, adding it to a netflix queue and listening to the soundtrack. Our approach goes beyond the presentation of actions from different providers, and allows to develop clients that execute the actions in a specified dynamic manner: A client in which users can look for movies; if the review rating of an identified movie is above a threshold the movie is added to the users netflix queue and the soundtrack is played. Such clients can be applications such as apps for a handheld device or be deployed on the web themselves as encapsulated program resources.

In this paper we describe

- how self-descriptive resources can be designed to enable loosely coupled clients;
- a service model for REST based on state transition systems as formal grounding;
- a declarative rule-based execution language to allow an intuitive specification of the interaction with resources from different providers;
- an execution engine as artifact to perform the defined interactions in a scaleable manner.

Example 2. To illustrate how we address the problem of designing a programming framework for REST enabled by Linked Data we use the example of a web blog API in the remainder of the paper. On the web blog users have accounts and blog entries are shown in a timeline. A blog is a suitable example, since it requires many different types of data centric interactions beyond simple data retrieval, i.e., create, read, update and delete of blog entries on the timeline and information about the users. Further in a social media context the value of composition can easily be observed, i.e., a client wants to share the functionality or data from other services with the users of the blog. In particular we are looking at a scenario where resources from a restaurant recommendation service are used to create new blog entries.

The rest of the paper is structured as follows: In Section 2 we detail the existing work. In Section 3 we describe the methods with which we intend to leverage the advantages of Linked Data based REST architectures. We conclude in Section 4.

2 Related Work

Pautasso introduces an extension to BPEL [17] to allow a composition of REST and traditional web services. To allow for a BPEL composition REST services are wrapped in WSDL descriptions.

There are several approaches that extend the existing WS-* stack with semantic capabilities by leveraging ontologies and rule-based descriptions (e.g., [22, 6, 4]) to achieve an increased degree of automation in high level tasks, such as service discovery, composition and mediation. Those approaches extending WS-* became known as Semantic Web Services (SWS). An Approach to combine RESTful services with SWS technologies in particular WSMO-Lite [24] was investigated by Kopecky et al. [11]. In contrast to SWS do REST architectures not allow to define arbitrary functions, but are constrained to a defined set of methods and are build around another kind of abstraction: the resource. Therefore our approach is more focused on resource/data centric scenarios in distributed environments (e.g., in the Web).

The scripting language S [3] allows to develop Web resources for REST interactions with a focus on performance due to parallelisation of calculations. In their definition resources can make use of other resources, thus also enabling a way of composing REST services. S does not explicitly address flexibility aspects of REST and has no explicit facilities to leverage hypermedia controls or to infer required operations from resource states.

RESTdesc [23] is an approach in which RESTful Linked Data resources are described in N3-Notation. The composition of resources is based on an N3 reasoner and stipulates manual interventions of users to decide which hypermedia controls should be followed.

Hernandez et al. [10] proposes a model for semantically enabled REST services as a combination of pi-calculus [15] and approaches to triple space computing [5] pioneered by the Linda system [9]. They argue, that the resource states can be seen as triple spaces, where during an interaction triple spaces can be created and destroyed as proposed in an extension of triple space computing by Simperl et al. [20].

Similar to the idea of triple spaces is the composition of RESTful Linked Data resources in a process space, proposed by Krummenacher et al. [12] based on resources described using graph patterns. Speiser and Harth [21] propose similar descriptions for RESTful Linked Data Services. Our approach shares the idea that graph pattern described resources read input from and write output to a shared space. We want to improve on this approach by providing a rigid service model and a more explicit way of defining the interaction with resources.

3 Methodology

In this section, we describe in more detail how we want to address the challenges we face in the development of a flexible and scalable programming framework. We address

– *resource descriptions* to allow to predict the effect of the execution of a functionality before invocation;
– a formal *service model* as grounding to describe the interactions that are offered and RESTful Linked Data resources, potentially spread over different servers;
– an *execution language* to instantiate a concrete interaction between a client and resources, which preserves the adaptability, robustness and flexibility of REST.

3.1 Resource Descriptions

In a RESTful interaction with Linked Data resources only the HTTP methods can be applied to the resources. The semantics of the HTTP methods itself is defined by the IETF[3] and do not need to be explicitly described.

Table 1: Overview of HTTP methods

Method	safe	requires input	intuition
GET	x		Retrieve the current state of a resource.
OPTIONS	x		Retrieve a description of possible interactions.
DELETE			Delete a resource
PUT		x	Create or update a resource.
POST		x	Send input as subordinate to a resource.

Table 1 shows an overview of the HTTP methods. We can distinguish between safe and non-safe methods, where safe methods guarantee not to affect the current states of resources. Further, some of the methods require additional input data to be provided for their invocation. The communicated input data can be subject to requirements that need to be described to allow an automated interaction. Furthermore, the effect on the state of resources an application of a non-safe method has, can depend on the input data. The dependency between communicated input and the resulting state of resources also needs to be described. Therefore, only the non-safe HTTP methods that require input data need further description mechanisms. Note, that states of not directly addressed resources can also be influenced by non-safe HTTP methods that require input data.

The state of a Linked Data resources is expressed with RDF. It is sensible to serialise the input data, i.e., data that is submitted to resources to manipulate their state, in RDF as well. To convey the resulting state change after application of a HTTP method we use RDF output messages. In previous work [16] we analysed the potential of graph patterns, based on the syntax of SPARQL[4], to describe required input as well as their relation to output messages. The resulting

[3] `http://www.ietf.org/rfc/rfc2616.txt`
[4] `http://www.w3.org/TR/rdf-sparql-query/#GraphPattern`

graph pattern descriptions are attached to the resource and can be retrieved via the *HTTP OPTIONS* method on the respective resource. Therefore the resources stay self-descriptive, i.e., their current state can be retrieved with *HTTP GET*, the possibilities to influence their state with *HTTP OPTIONS*.

Example 3. In our web blog example the user accounts, the timeline and the blog entries are resources a client can interact with. Further the restaurant recommendations are resources from another service

- Simply reading the blog entry, requires a GET on the entry without any input data and does not change the state of any resource.
- Applying a DELETE on an a blog entry does not require input; its effect is inherently defined by the method: the entry is erased, i.e. a state change.
- Submitting a new entry means to POST input data to the timeline. In a composed scenario the input data can be derived from the restaurant recommendations. The result of the POST is the creation of a new resource. Further, if the input contains optionally a username, a link could be set in the corresponding user account to the newly created blog entry. The information about input requirements and the relation to resulting resource states are described with the graph patterns attached to the timeline.

Figure 1 illustrates the timeline resource of our example, with a set of entries in the current state and the graph pattern that describe how a new entry can be POSTed.

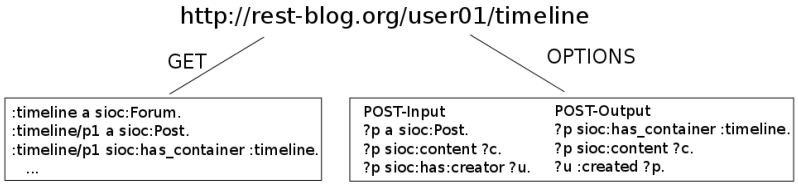

Fig. 1: Self-descriptive resource: current state can be accessed with GET, input/output description with OPTIONS

3.2 REST Service Model

A REST service can be identified with the resources it exposes. An interaction within a REST architecture is based on the manipulation of the states of the exposed resources.

We want to develop a service model, that allows to formalise the functionalities exposed by a service based on Linked Data resources. A formal service model serves as rigid specification of how the use of individual HTTP methods influences resource states and how these state changes are conveyed to interacting clients.

We model a Linked Data-based RESTful service as a REST state transition system (RSTS) similar to a state machine as defined by Lee and Varaiya [13]. The behavior of the clients themselves is not in the scope of this model, rather all possible interaction paths of a client with the resources are formalised.

Definition 1. *An RSTS is defined as a 6-tuple* $RSTS = \{R, \Sigma, I, O, M, \delta\}$ *with*

- *a set of resources* $R = \{r_1, r_2, ...\}$
- *a set of states* $\Sigma = \{\sigma_1, ..., \sigma_m\}$ *with* $\sigma_k = (\bigcup_{r_i \in R} \overline{r_i^k})$ *a complete state of the RSTS with*
  - $\overline{r_i^k}$ *the RDF representation of the state of* $r_i \in R$ *in state* σ_k
- *input alphabet* $I = \{(r, g) : R \times G\}$ *where*
  - G *the set of all possible RDF graphs*
- *output alphabet* $O = \{(c, o) : C \times 2^{\overline{R}}$ *where*
  - C *the set of all HTTP status codes*
  - $\overline{R} = \bigcup_{k=1}^{m} \bigcup_{r_i \in R} \overline{r_i^k}$*, the set of all possible states of all resources*
- *the set of HTTP methods*[5] $M = \{GET, DELETE, PUT, POST\}$
- *update function* $\delta : \Sigma \times I \to \Sigma \times O$*, which can be decomposed in*
  - *output functions* $\delta_\mu^o : \Sigma \times I \to O$ *for every* $\mu \in M$ *given by*
    - $\delta_{get}^o(\sigma_k, (r_i, \emptyset)) = (c\,,\, \overline{r_i^k})$
    - $\delta_{delete}^o(\sigma_k, (r_i, \emptyset)) = (c\,,\, \emptyset)$
    - $\delta_{put}^o(\sigma_k, (r_i, g)) = (c\,,\, \sigma_{k+1} \backslash \sigma_k)$
    - $\delta_{post}^o(\sigma_k, (r_i, g)) = (c\,,\, \sigma_{k+1} \backslash \sigma_k)$
  - *state change functions* $\delta_\mu^s : \Sigma \times I \to \Sigma$ *for every* $\mu \in M$ *given by*
    - $\delta_{get}^s(\sigma_k, (r_i, \emptyset)) = \sigma_k$
    - $\delta_{delete}^s(\sigma_k, (r_i, \emptyset)) = \sigma_k \backslash \{\overline{r_i^k}\}$
    - $\delta_{put}^s(\sigma_k, (r_i, g)) = \sigma_{k+1}$
    - $\delta_{post}^s(\sigma_k, (r_i, g)) = \sigma_{k+1}$

R defines the set of URI identified resources that are (potentially) exposed by the service. Note that the set of resources can be infinite, since a service can allow to create additional resources.

A state $\sigma_k \in \Sigma$ in the RSTS is defined as the set of states of all resources that are (potentially) exposed by the service, serialised with RDF. We consider the serialisation of a state of a resource that does not exist to be an empty set. Therefore the complete serialisation of a state in the $RSTS$ is finite, since only the existing resources have to be described.

The transitions between states are described with an update function δ that maps from a state in RSTS and an element of an input alphabet to a state in RSTS and an element of an output alphabet. The elements of the input alphabet I are tuples defining an addressed resource and RDF input data. The elements of the output alphabet O are tuples consisting of an HTTP status code and RDF output data.

[5] For brevity we focus here on the four most important methods. Other methods can be added analougously

The update function can be decomposed in *output functions* δ^o_μ, that just maps to the output, and *state change functions* δ^s_μ, that just map to a state, for every HTTP method $\mu \in M$ respectively.

The intuition behind the state change functions is that a state transition in the RSTS is effected by influencing resource states with HTTP methods. Safe methods that do not change any resource states, describe self-transitions, i.e., transitions that start and end in the same state.

Resources not necessarily allow the use of all HTTP methods. Note that all state change functions are defined for every resource, i.e., every resource can be addressed with all methods: If a resource does not allow for the application of a specific method the state change function describes a self-transition.

The intuition behind output functions is, that the application of an HTTP method on a resource also results in a defined output, that communicates the success with an HTTP status code. Further, the output contains an RDF message that describes the by the HTTP method effected state change.

Figure 2 illustrates a state transition in RSTS where an entry is POSTed to a blog timeline. Note, that a client could derive the input for the POST method from the states of other resources.

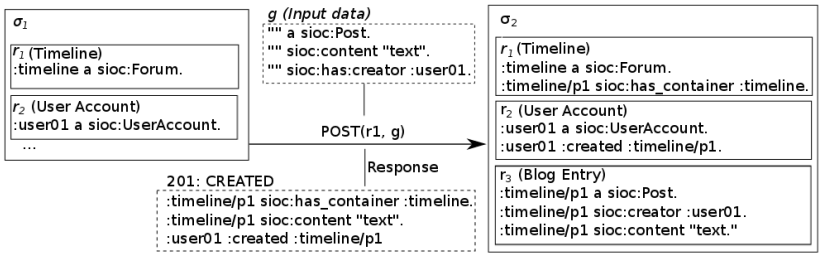

Fig. 2: State transition of a RSTS, with excerpts of two states.

The possible interactions with the resources of several services can easily be formalised together in one RSTS as the side-by-side composition [13] of the transition systems of the individual services. Intuitively the side-by-side composition results in an RSTS, whose states contain the set of the states of the resources from both services.

The defined service model serves as formal grounding of the execution language described in Section 3.3. However, the self-descriptive resources are sufficient to define a RESTful services:

- The current state of Linked Data resources - and therefore the state of the RSTS - can be accessed as RDF.
- The possible transitions and the state they result in is declared with the graph pattern descriptions.

3.3 Execution Language

In a resource-driven environment applications retrieve and manipulate resources exposed on the Web. Since the resources can potentially be accessed by a multitude of clients, applications have to react dynamically on the state of the resources. Therefore an important factor in the development of resource-driven applications is the dependency between the invoked transitions and resource states..

Definition 2. *The dependency between the invoked state transitions (i.e., applied HTTP methods) and the states of resources is that*

1. *input data for the transition is derived from RDF detailing the states of resources and/or*
2. *the transition is only invoked if the resources are in a specified state.*

Example 4. A client might want to post a restaurant recommendations to the timeline of the blog, but only of restaurants in towns the users live in. The restaurant recommendations are resources from another service. The retrieved RDF state representations of the recommendations are used to POST entries to the timeline (1). However, a recommendation is only POSTed if a user account specifies that a user lives in the town of the recommended restaurant (2).

Therefore, state transitions that are to be invoked, have to be defined, to specify the interaction of a client with RESTful Linked Data resources and congruously the desired path through the RSTS. Further the conditions, subject to the current states of resources, under which a specific transition is to be invoked have to be specified.

To allow programmers to formalise their desired interactions we propose a declarative rule-based execution language.

Definition 3. *A rule ρ is of the form $\mu(r) : -Q$ where $\mu \in M$, $r \in R$ and Q a conjunctive query.*

The head of a rule corresponds to an update function of the RSTS in that they describe an HTTP method that is to be applied to a resource. The rule bodies are conjunctive queries that allow programmers to express their intention under which condition a method is to be applied. Thus, programmers can define an interaction pattern with a set of rules for their client applications.

The use of conjunctive queries is motivated by the idea that clients have to maintain a knowledge space (KS) in which they store their knowledge about the states of the resources they interact with [12]. KS is filled with the RDF data the client receives after applying an HTTP method, as defined by the output functions of the RSTS. The output always informs the client about the current state after the application of the method.

Concretely SPARQL queries can be employed, which are evaluated over KS. Queries are also used to dynamically, i.e., during runtime

- derive input data from the states of other resources, as stored in KS and

– identify the resource an HTTP method has to be applied to, i.e., leveraging hypermedia controls.

To derive input data from the states of other resources, as stored in KS, *construct* queries can be employed. In case no input is required, an *ask* query is sufficient.

To preserve the flexibility provided by REST our execution language has to be able to make use of links in the resource states to other resources. Rather than specifying a resource explicitly to which a method is applied, a SPARQL *select* query can be used to extract the URI of the addressed resource from KS.

Example 5. In our Web blog example a user account could provide links to the accounts of the friends of the user. A client that wants to GET data about the friends of a specified user can instead of addressing the friends accounts directly, SELECT the friends from the retrieved data in KS after retrieving the information about the specified user. Thus every time the client performs its interactions the friends are identified at runtime and only the current friends are retrieved.

Further the execution language allows to define input and output of programs as graph patterns. The input pattern describes the structure of data that can initially be imported in KS to start the interaction as defined by the rules. After the interaction is completed the output pattern is evaluated over KS, thus extracting output data as result of the interaction. The notion of input and output allows to deploy the defined interaction itself as a resource in the Web, e.g., as servlet or cgi-bin, which allows to encapsulate the interaction with resources and expose the functionality of a program. Note that the input pattern and output pattern are equivalent to the description pattern of other resources as described in section 3.1.

An interpreter that can be integrated in applications can be used as as execution engine for the rule language. The engine can implements the KS as well as the functionality to invoke an interaction with resources as defined with the execution language.

To achieve a fast scalable interpreter the execution engine can be build similar to a query engine, which allows a multithreaded, parallel evaluation of multiple queries (e.g., based on the Rete algorithm [8]).

To enable a wide variety of applications the engine can include an extension to support the interaction with REST resources that are not based on Linked Data. The engine can store data entities (e.g., binaries, JSON documents) received from such services separately. A triple pointing to a received non-RDF entity can be included in KS, thus the entities can be used in the logic of the execution rules. However, an interaction with such non-RDF entities requires to fall back to a more mashup-like programming approach.

4 Conclusion

In this paper we described how Linked Data Resources can be extended with descriptions for RESTful manipulation. The natural extension of Linked Data with

RESTful manipulation of resources enables a framework with uniform semantic resource representations for REST architectures. We have proposed to exploit the advantages resulting from the combination of REST and Linked Data in a programming framework for the Semantic Web. We have sketched a declarative rule-based execution language with a state transition system as formal grounding and the challenges we address with these language, i.e., achieving scalability and performance while preserving the flexibility and robustness of REST. Further we outlined an execution engine for the language.

References

1. Berners-Lee, T.: Read-write linked data (Aug 2009), `http://www.w3.org/DesignIssues/ReadWriteLinkedData.html`
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. IJSWIS 5(3), 122 (2009)
3. Bonetta, D., Peternier, A., Pautasso, C., Binder, W.: S: a scripting language for high-performance restful web services. In: PPOPP (2012)
4. Cardoso, J., Sheth, A.: Semantic Web Services, Processes and Applications. Springer (2006)
5. Fensel, D.: Triple-space computing: Semantic web services based on persistent publication of information. In: INTELLCOMM. pp. 43–53 (2004)
6. Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., Domingue, J.: Enabling Semantic Web Services: The Web Service Modeling Ontology. Springer (2006)
7. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
8. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. AIJ 19(1), 17–37 (1982)
9. Gelernter, D.: Generative communication in linda. ACM TOPLAS 7, 80–112 (1985)
10. Hernández, A.G., García, M.N.M.: A formal definition of restful semantic web services. In: WS-REST. pp. 39–45 (2010)
11. Kopecky, J., Vitvar, T., Fensel, D.: Microwsmo: Semantic description of restful services. Tech. rep., WSMO Working Group (2008)
12. Krummenacher, R., Norton, B., Marte, A.: Towards Linked Open Services. In: FIS (2010)
13. Lee, E.A., Varaiya, P.: Structure and Interpretation of Signals and Systems. Addison-Wesley (2011)
14. Lin, T., Pantel, P., Gamon, M., Kannan, A., Fuxman, A.: Active objects: actions for entity-centric search. In: WWW. pp. 589–598 (2012)
15. Milner, R.: The polyadic pi-calculus. In: CONCUR (1992)
16. Norton, B., Stadtmüller, S.: Scalable discovery of linked services. In: RED (2011)
17. Pautasso, C.: Restful web service composition with bpel for rest. DKE 68(9), 851–866 (2009)
18. Pautasso, C., Wilde, E.: Why is the web loosely coupled?: a multi-faceted metric for service design. In: WWW. pp. 911–920 (2009)
19. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media (2007)
20. Simperl, E., Krummenacher, R., Nixon, L.: A coordination model for triplespace computing. In: COORDINATION (2007)

21. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: ESWC (2011)
22. Studer, R., Grimm, S., Abecker, A. (eds.): Semantic Web Services: Concepts, Technologies, and Applications. Springer (2007)
23. Verborgh, R., Steiner, T., Deursen, D.V., de Walle, R.V., Valls, J.G.: Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In: NWeSP (2011)
24. Vitvar, T., Kopecky, J., Zaremba, M., Fensel, D.: Wsmo-lite: Lightweight semantic descriptions for services on the web. In: ECOWS. pp. 77–86 (2007)
25. Webber, J.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly (2010)
26. Weiss, M., Gangadharan, G.R.: Modeling the mashup ecosystem: structure and growth. RADMA 40(1), 40–49 (2010)
27. Wilde, E.: Rest and rdf granularity (May 2009), `http://dret.typepad.com/dretblog/2009/05/rest-and-rdf-granularity.html`