Chapter 5
Survey of Semantic Description of REST APIs

Ruben Verborgh, Andreas Harth, Maria Maleshkova, Steffen Stadtmüller, Thomas Steiner, Mohsen Taheriyan and Rik Van de Walle

5.1 Introduction

The REST architectural style assumes that client and server 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 [247]. In different application scenarios such an agreement requires vendor-specific content types for the individual services to convey the meaning of the communicated data. The idea behind vendor-specific content types is that service providers can

* The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

R. Verborgh (✉)
Multimedia Lab – Ghent University – iMinds, Gaston Crommenlaan 8 bus 201, Ledeberg-Ghent 9050, Belgium
e-mail: ruben.verborgh@ugent.be

A. Harth · M. Maleshkova · S. Stadtmüller
Institute AIFB, Karlsruhe Institute of Technology (KIT), Karlsruhe 76128, Germany
e-mail: harth@kit.edu

M. Maleshkova
e-mail: maria.maleshkova@kit.edu

S. Stadtmüller
e-mail: steffen.stadtmueller@kit.edu

T. Steiner
Departament de Llenguatges i Sistemes Informatics, Universitat Politècnica de Catalunya, Jordi Girona, 29, Barcelona 08034, Spain
e-mail: tsteiner@lsi.upc.edu

M. Taheriyan
Information Science Institute, University of Southern California, Admiralty Way, Suite 1001, Marina del Rey, CA 4676, USA
e-mail: mohsen@isi.edu

R. Van de Walle
Multimedia Lab – Ghent University – iMinds, Gaston Crommenlaan 8 bus 201, Ledeberg- Ghent 9050, Belgium
e-mail: rik.vandewalle@ugent.be

C. Pautasso et al. (eds.), *REST: Advanced Research Topics and Practical Applications,*
DOI 10.1007/978-1-4614-9299-3_5, © Springer Science+Business Media New York 2014

reuse content types and service consumers can make use of specific processors for the individual content types. In practice however, we see that many RESTful APIs on the Web simply make use of standard non-specific content types, *e.g.*, *text/xml* or *application/json* [150]. Since the agreement on the semantics is only implicit, programmers developing client applications have to manually gain a deep understanding of several APIs from multiple providers.

Common Web APIs are typically either exclusively described textually[1], or far less frequently—and usually based on third-party contributions—a machine-readable WADL [104] API description exists[2]. However, neither human-focused textual, nor machine-focused WADL API descriptions carry any *machine-processable* semantics, *i.e.*, do not describe *what* a certain API does. Instead, they limit themselves to a description of machine-readable in- and output parameters in the case of WADL, or a non-machine-readable prose- and/or example-driven description of the API in the case of textual descriptions. While this may suffice the requirements of developers in practice, the lack of semantic descriptions hinders many more advanced use cases such as API discovery or API composition.

Machine-interpretable descriptions can serve several purposes when developing client applications:

- One can generate textual documentation from the standardised machine-interpretable descriptions, which leads to a more coherent presentation of the APIs, similar to what JavaDoc has achieved in the Java world (see also Knuth's idea of literal programming [125]).
- A standardised way to access REST APIs introduce a higher degree of automation for high level tasks such as composition.
- Machine-interpretable descriptions facilitate a more structured approach to developing APIs, which means that automated tools can check coherence and the RESTful-ness of an API (e.g., according to the Richardson Maturity Model[3]).

With hypermedia being an essential constraint of the REST architectural style [78], the transitions between different application states of a REST API have to be hypermedia-driven. In that sense, semantic API descriptions complement the hypermedia-driven interactions by providing additional clues for machine clients that help them make informed decisions.

In a RESTful interaction with Web resources, only the constraint set of HTTP methods can be applied to the resources. The semantics of the HTTP methods itself is defined by the IETF [80] and do not need to be explicitly described for individual resources. We can distinguish between safe and non-safe methods, where safe methods guarantee not to affect the current state of resources. Additionally, 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 resources state of an

[1] For example, the Twitter REST API: https://dev.twitter.com/docs/api

[2] A large archive of WADL descriptions is available on GitHub: https://github.com/apigee/wadl-library

[3] http://martinfowler.com/articles/richardsonMaturityModel.html

application of a non-safe method has needs to be assessable before the actual invocation to allow clients to decide how to interact with the resources. The effected change of resources after applying an HTTP method can also depend on the communicated input data. This dependency between communicated input and the resulting state of resources has also to be subject of a description.

In the chapter we classify the various approaches for providing machine-interpretable descriptions of Web APIs. Section 5.2 surveys lightweight semantic descriptions. Section 5.3 introduces descriptions based on graph patterns (a subset of the SPARQL query language for RDF, a graph-structured data format). Section 5.4 covers logic-based descriptions. Section 5.5 covers JSON-based descriptions. Section 5.6 contains a description of two tools for annotating existing APIs, and Sect. 5.8 concludes.

5.2 Lightweight Semantic Descriptions

5.2.1 Syntactic REST API Descriptions

Web services enable the publishing and consuming of functionalities of existing applications, facilitating the development of systems based on decoupled and distributed components. **WSDL** [49, 254] is an XML-based language for describing the interface of a Web service. A WSDL service description specifies: (1) the supported operations for consuming the Web service; (2) its transport protocol bindings; (3) the message exchange format; and (4) its physical location. In this way, the WSDL description contains all information necessary for invoking a service and since it is XML-based, conforming to the WSDL xml schema, it is also machine-processable.

WSDL Similarly, to Web services, which provide access to the functionality of existing components, Web APIs and Web applications conforming to the REST paradigm, provide access to resources by using the WWW as an infrastructure platform. Based on this parallel between the two types of services, WSDL was extended to Version 2.0 [254], which can also be used for formally describing RESTful services. As a result WSDL is a machine-processable, platform- and language-independent form of describing Web services and RESTful services alike.

The difficulty of using WSDL for RESTful services, is that it was not especially designed for resource-oriented services and as a result, everything has to be described in an operation-based manner. In addition, WSDL introduces some difficulties with specifying the message exchange format and limits HTTP authentication methods. Moreover, the most important drawback is that it lacks support for simple links. There is no mechanism in WSDL 2.0 to describe new resources that are identified by links in other documents. However, one of the most important characteristics of RESTful services is that they consist of collections of interlinked resources. Finally, the adoption of WSDL as means for describing Web APIs would requite that all providers update or completely change their websites with documentation, moving

away for using only text in HTML form. Similarly developers would need to learn to deal with WSDL instead of simply reading a natural language description.

WADL In contrast to WSDL, the Web Application Description Language (WADL, [104]) was especially designed for describing RESTful services in a machine-processable way. It is also XML-based and is platform and language independent. As opposed to WSDL, WADL models the resources provided by a service, and the relationships between them in the form of links. A service is described using a set of resource elements. Each resource contains descriptions of the inputs and method elements, including the request and response for the resource. In particular, the request element specifies how to represent the input, what types are required and any specific HTTP headers. The response describes the representation of the service's response, as well as any fault information, to deal with errors.

Currently, neither WADL nor WSDL are widely accepted and used for Web APIs and RESTful services. A Google search for WADL files returns only 49 unique results[4], while from the popular Web 2.0 applications only delicious[5] and YahooSearch[6] have WADL descriptions. The main difficulty of using WADL descriptions is that they are complex, in comparison to text-based documentation, and require that developers have a certain level of training and tool support that enables the application development on top of WADL. This complexity contradicts with the current proliferation of Web APIs, which can be greatly attributed to simplicity and direct used of the infrastructure provided by the Web, which enable the easy retrieving of resources only thought an HTTP request, directly in the Web browser.

Web APIs evolve rather autonomously without conforming to a shared set of guidelines or standards, especially evident by the fact the documentation is usually given in natural language as part of a webpage. The developer has to decide what structure to use and what information to provide. As a result, everyone who is able to create a Web page is also able to create a Web API description.

However, plain text/HTML descriptions, in contrast to WSDL and WADL descriptions, are not meant for automated machine-processing, which means that if developers want to use a particular service, they have to go to an existing description Web page, study it and write the application manually. Therefore, current research proposes the creation of machine-interpretable descriptions on top of existing HTML descriptions by using **microformats** [119]. Microformats offer means for annotating human-oriented Web pages in order to make key information automatically recognisable and processable, without modifying the visualization or the content.

hREST One particular approach for creating machine-processable descriptions for RESTful services by using microformats is hRESTS (HTML for RESTful Services) [129]. hRESTS enables the marking of service properties including operations, inputs and outputs, HTTP methods and labels, by inserting HTML tags within the HTML. In this way, the user does not see any changes in the Web page, however,

[4] Search done on October 5th, 2009
[5] http://delicious.com/
[6] http://search.yahoo.com/

based on the tags, the service can be automatically recognized by crawlers and the service properties can directly be extracted by applying a simple XSL transformation. The result is an HTML page that also contains the syntactical information of the described Web API and therefore, no longer relies solely on human interpretation. Versioning in hRESTS is dealt with the same way as in microformats through backwards-compatible additions of class names[7], potentially requiring wrapping elements.

RDFa An alternative to using hRESTS is offered by RDFa [2] that enables the embedding of RDF data in HTML. RDFa is similar to using microformats, but is somewhat more complex and offers more HTML markup options, as opposed to hRESTS. Approaches, based on making existing RESTful service descriptions machine-processable by using HTML tags are simpler and more lightweight as opposed to WSDL and WADL. In addition, as already mentioned, they can be applied directly on already available descriptions, rather then creating new service descriptions from scratch. The adoption by developers is also easier, since the creation of a machine-processable RESTful service description is equivalent to Web content creation or modification.

5.2.2 MicroWSMO/SA-REST

In contrast to research in the area of Semantic Web Services, which has been quite prolific, the number of semantic frameworks targeted at capturing Web API characteristics is relatively limited. Web APIs have only recently achieved greater popularity and wider use, thus raising the interest of the research community. In this section we discuss the two main approaches—MicroWSMO and SA-REST, aiming to support a greater level of automation of common service tasks through employing semantics. We also consider further description languages and ontologies, including ReLL and ROSM.

MicroWSMO MicroWSMO [128] is a formalism for the semantic description of Web APIs, which is based on adapting the SA-WSDL [73] approach. MicroWSMO uses microformats for adding semantic information on top of HTML service documentation, by relying on hRESTS for marking service properties and making the descriptions machine-processable. It uses three main types of link relations: (1) **modelReference**, which can be used on any service property to point to appropriate semantic concepts identified by URIs; (2) **liftingSchemaMapping** and (3) **loweringSchemaMapping**, which associate messages with appropriate transformations (also identified by URIs) between the serialization format such as XML and a semantic knowledge representation format such as RDF. Therefore, MicroWSMO, based on hRESTS, enables the semantic annotation of Web APIs in the same way in which SA-WSDL, based on WSDL, supports the annotation of Web services.

[7] http://microformats.org/wiki/microformats-2#backward_compatible

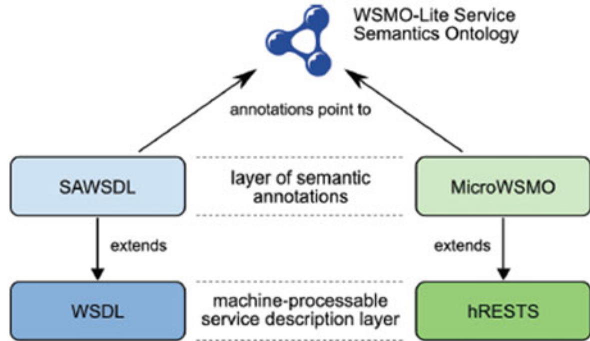

Fig. 5.1 Unifying SA-WSDL and MicroWSMO through WSMO-Lite

In addition, MicroWSMO can be complemented by the WSMO-Lite service ontology specifying the content of the semantic annotations (see Fig. 5.1[8]). Since both Web APIs and WSDL-based services can have WSMO-Lite annotations, this provides a basis for integrating the two types of services. Therefore, WSMO-Lite enables unified search over both Web APIs and WSDL-based services, and tasks such as discovery, composition and mediation can be performed based on WSMO-Lite, completely independently from the underlying Web service technology (WSDL/SOAP or REST/HTTP).

SA-REST Another formalism for the semantic description of RESTful services is SA-REST [212], which also applies the grounding principles of SA-WSDL but instead of using hRESTS relies on RDFa [2] for marking service properties. Similarly to MicroWSMO, SA-REST enables the annotation of existing HTML service descriptions by defining the following service elements: input, output, operation, lifting, lowering, or fault and linking these to semantic entities. The main differences between the two approaches are not the underlying principles but rather the implementation techniques. In addition, MicroWSMO aims to add semantics as means for Web service automation, while SA-REST is more oriented towards enabling tool support in the context of service discovery and composition by mashup or smashup [211].

5.2.3 Minimal Service Model

The Minimal Service Model (MSM) represents an operation-based approach towards describing Web APIs. It is a simple RDF(s) ontology that supports the annotation of common Web API descriptions. It also aims to enable the reusability of existing Semantic Web service approaches by capturing the maximum common denominator between existing conceptual models for services. Additionally, as opposed to most semantic Web services research up to date, MSM targets to support both traditional Web services, as well as Web APIs with procedural view on resources, so that they can be handled in a unified way.

[8] http://www.wst.univie.ac.at/workgroups/sem-nessi/index.php?t=semanticweb

Originally the MSM was introduced together with hRESTS [129], aiming to cover for the fact that Web API descriptions do not typically have any structure in terms of the resources handled or the operations exposed. Although its initial purpose was to provide structure to hRESTS, it has been subsequently adjusted and updated to its current form, in order to provide means for integrating heterogeneous services (*i.e.*, WSDLs and Web APIS), and together with WSMO-Lite it has been used as a means to facilitate a common framework covering the largest common denominator of the most used Semantic Web service formalisms on the Web. On the basis of MSM and WSMO-Lite, generic publication and discovery machinery has been developed that supports SA-WSDL, WSMO-Lite, hREST/MicroWSMO, and OWL-S services [186]. The Semantic Web Services exposed by this infrastructure [185] put the emphasis on reducing the complexity of conceptual models and integrating services with existing Linked Data [33]. This integration serves both as a means to simplify the creation and management of Semantic Web Services through reuse, as well as it provides a new view over semantic Web services understood as a means to support the generating and processing Linked Data on the Web. Subsequent work around the Minimal Service Model is focused on supporting the invocation and authentication of Web APIs [143], [149].

The original MSM [129] defined a Web API in terms of *Services* that has a number of *Operations*, which have an *Input*, an *Output*, and *Faults*. However, the original MSM, which was used as a basis for SA-REST [212] and MicroWSMO [128], fails to capture some significant parts of the descriptions, such as optional, default and mandatory parameters, which can have a crucial effect on discovery and invocation. In addition, it does not enable the description of the inputs, parts of the input, and parts of the parts of the input. As a result, the MSM has been extended [186] to support further service characteristics.

The MSM, in its current version [186], is visualized in Fig. 5.2. The MSM, denoted by the *msm* namespace, defines *Services* with a number of *Operations*. Operations in turn have input, output and fault *MessageContent* descriptions. A *MessageContent* may be composed of *MessageParts*, which can be mandatory or optional. The intent of the message part mechanism is to support finer-grain discovery based on message parts and allowing distinguishing between mandatory and optional parts.

The MSM is used as the core model for describing Web APIs, while supplementary models are defined for capturing details that are of particular relevance for the separate service tasks. As part of identifying extensions to the MSM, work on supporting the automation of authentication [149] was conducted.

5.2.4 *Further Semantic Approaches*

ROSM The Resource-Oriented Service Model (ROSM [82]) ontology is a lightweight approach to the structural description of resource-oriented RESTful services, compatible with WSMO-Lite annotations. It enables the annotation of resources belonging to a service. In turn the resources can be described as being

Fig. 5.2 Minimal service model

part of collections and having addresses (URIs) serving as endpoints for access and manipulation. The organization of resource in collections, which again belongs to a service, allows capturing an arbitrary number of resources and attaching service semantics to them following the SA-WSDL approach.

Furthermore, resources can have certain HTTP methods associated with them, which define how it is possible to interact with a resource, which are connected through an operation. These operations are modeled in a much-more fine-grained way, since they basically only have to support the uniform interface of HTTP. In addition, ROSM[9] enables the explicit modeling of requests and responses with their associated aspects (*e.g.*, parameters or response codes). In summary, ROSM represents a simple ontology for describing resource-centered services, in terms of resources, collection of resources, addresses and HTTP methods. The specific semantics of the ROSM annotations can be described by linking to the WSMO-Lite ontology.

Servont is an ontology-based hybrid approach where different kinds of matchmaking strategies are combined together to provide an adaptive, flexible and efficient service discovery environment [62]. Semantic-enriched frameworks are considered a key issue to enforce timely discovery and dynamic composition of services. In contrast, the ServFace project aims at creating a model-driven service engineering methodology for an integrated development process for service-based applications[10]. Finally, the Simple Semantic Web Architecture and Protocol (SSWAP) is the driving technology for the iPlant Semantic Web Program[11]. It combines Web service

[9] http://www.wsmo.org/ns/rosm/0.1/
[10] http://www.servface.eu/
[11] http://sswap.info/

functionality with an extensible semantic framework to satisfy the conditions for high throughput integration [91]. SSWAP originates from the Semantic MOBY project, which is a branch of BioMOBY project [251]. Using SSWAP, users can create scientific workflows based on the discovery and execution of Web services and RESTful services.

5.3 SPARQL-Based Descriptions

Following the motivation to look beyond the exposure of fixed datasets, an extension of Linked Data with REST technologies has been proposed and explored for some time [24], [134], [168], [249], [217], [240]. These approaches extend the traditional use of HTTP in Linked Data, consistent with REST, by allowing all HTTP operations to be applied to Linked Data resources. REST services in this context are often perceived RDF prosumers, *i.e.*, the state of resources is made available encoded in RDF, at least as an alternative via Content Negotiation. Clients can interact with Linked Data resources by submitting RDF input data with HTTP methods (*e.g.*, POST, PUT) resulting in the manipulation or creation of resources. The output data a client receives after the successful submission of data describes the effected state change of resources (*i.e.*, their new content after creation or manipulation) and is serialized in RDF as well.

Such REST services contribute to the Web of Data by interlinking output data with existing Linked Data sets.

RESTful Linked Data resources consider how the data that results from computation over input can carry explicit semantics and base their service descriptions on the notion that Linked Data provides a description for resources' input and output requirements: the graph patterns provided by the SPARQL query language or the N3 notation. Graph pattern provide the advantage of a more thorough description of what should be communicated, familiarity to Linked Data producers and consumers, and the possibility for increased tool support.

The rational behind the descriptions for RESTful Linked Data resources is that the current state of a resource can be retrieved with an HTTP GET, while the data exchange that constitutes a manipulating interaction with a resource is described with two graph patterns:

- A graph pattern that describes the RDF input data that is submitted to a resource (*e.g.*, with HTTP POST), which is necessary to invoke the manipulation.
- A graph pattern that describes the RDF output data the client receives after a successful call.

The description implies that the input pattern has to match the input data to invoke the service and the output pattern will match the output data returned by the service.

To illustrate the use of graph pattern-based IO descriptions we introduce an example based on an API of a social network platform. A common feature of social networks is to post a message to the timeline of a user. The approach in the context

Table 5.1 Example description for a timeline resource in a social network

Input:	{ ?post a sioc:Post. ?post sioc:content ?content. ?post sioc:has-creator ?user. }
Output:	{ ?post sioc:content ?content. ?post dcterms:created ?date. }

of Linked Data would be to wrap this call to provide the information in RDF, reusing existing vocabularies. Here the natural choice is the SIOC vocabulary[12].

The description of a resource representing the timeline of a user, would make explicit as required input for an HTTP POST call the content of the message and its creator identified in that social network. Furthermore the output would be described as the created message with enriched with additional information (*e.g.*, the creation date). A client would receive data matching this pattern together with the HTTP status code indicating the success of the call. The resulting input and output descriptions for the Linked API are represented in Table 5.1.

Note that the reuse of the variables ?post and ?content across input and output implies that this variable will have the same binding in the output as provided in the input. In SPARQL terms these would be 'safe variables' if we considered the service to be a SPARQL CONSTRUCT query from the input to the output patterns. Variables in the output pattern that do not appear in the input pattern are bound by the functionality of the resource, *i.e.*, the binding is a result of the manipulation or creation of a resource.

Graph pattern based descriptions allow for a thorough descriptions of what a client has to communicate to successfully interact with a RESTful Linked Data resource. The output descriptions that share variables with the input descriptions allow clients to anticipate the result of their interaction with respect to the input they intended to provide. Such an anticipation is due to the circumstance that the actual output message of an interaction is intended to convey the effected state change of an interaction. The predictability of effects of manipulating actions is essential to enable (semi-)automated clients that use Linked Data REST services.

5.4 Logic-Based Descriptions

Since its inception, the Semantic Web has always had a strong link with logic [27]. The impact of logic is visible in essential building stones such as RDF [123], whose *open world* assumption prohibits conclusions based on the absence of certain triples. Two logic families are predominant on the Semantic Web: *description logic*, the underlying model of OWL [156], is the most widespread, followed by *first-order logic*, which is typically expressed in extensions of RDF with rules. Description logic is in essence a decidable fragment of first-order logic, at the cost of a loss in expressivity.

[12] http://sioc-project.org/ontology/

This reduced expressivity is a motivator for LOS and LIDS to adopt SPARQL and the reason for a few other methods use rule-based expression languages. For example, in OWL-S [153], one of the early semantic description formalisms for traditional Web services, rule languages such as KIF or SWIRL have to be used to express various aspects that extend beyond the expressivity of RDF. These languages capture expressions for pre- and postconditions, results, and effects of a Web service invocation. In general, rules are a straightforward mechanism to express dynamic relationships, such as those that occur with Web APIs.

5.4.1 RESTdesc

RESTdesc [240], [241] is a logic-based description method that focuses on exposing the functional aspect of Web APIs in machine-processable form. Concretely, RESTdesc descriptions are rules expressed in the Notation3 language (N3 [25]), which adds support for variables and quantification to RDF. The latter is a prerequisite to natively describe statements such as "all requests to y result in x", which is not directly supported in RDF (*i.e.*, only through the use of modeling vocabularies). In fact, RESTdesc uses quantification to express functionality as follows:

$$\forall x \quad \mathit{preconditions}(x) \Rightarrow \exists r(\mathit{request}(r,x) \wedge \mathit{postconditions}(x))$$

Herein, the predicates are expressed as RDF graphs, called *formulae* in N3. Reasoners interpret the above as "for every situation x where certain preconditions are met, there exists a specified request r that makes certain postconditions true." Moreover, for specific situations x_n, reasoners can instantiate the above rule, thereby creating an RDF representation of a concrete HTTP request r_n, which can be executed by any RDF-compatible HTTP client.

Descriptions can be used to express the effects that occur as a result of a POST request, including the description of that request itself. Representations are *not* described, as RESTdesc aims to be representation-independent, but it can describe properties of these representations. This is not unlike the graph patterns used in SPARQL-based methods (see Sect. 5.3), but represented in a syntax that integrates the request description. RESTdesc descriptions can also be useful to describe requests with safe methods such as GET, in order to explain what properties the response will satisfy, again without constraining the representation. This can be useful in two cases: (*a*) if the client does not want to retrieve the resource (for instance, if there are too many) or (*b*) if the resource does not exist yet, but can be created as the result of another action. In both cases, clients can then predict certain properties of the resource, without actually accessing it.

Since RESTdesc descriptions are expressed as rules, they inherit the logical functionality, in particular the *chaining* property:

$$P \Rightarrow Q,\ Q \Rightarrow R \quad \vdash \quad P \Rightarrow R$$

Therefore, existing Semantic Web reasoners with N3 support can create compositions of RESTdesc-described Web APIs by chaining RESTdesc descriptions [238], either

in a forward or backward (goal-driven) manner. This enables agents to respectively discover possible actions and to find a sequence of actions to satisfy a predetermined goal.

RESTdesc descriptions have been designed [240] to support hypermedia-driven applications [78], as they essentially enable machines to anticipate on possible controls a resource might have. Client still need to operate as hypermedia consumers by following links, but the descriptions allow them to predict beforehand what resources can be of interest. For example, understanding that a POST request on a certain resource will lead to the creation of a subresource with a certain affordance, allows to client to reason about whether this action is desired.

Further work on RESTdesc includes the incorporation of quality parameters, possibly subjective, in Web API descriptions [239].

5.5 JSON-Based Descriptions

So far we have explored solutions that are directly based on Semantic Web technologies. However, many Web developers are reluctant to integrate RDF, SPARQL or N3 in their applications. Lanthaler and Gütl provide three reasons for what they call *semaphobia* [137]. Firstly, they observe that the perceived complexity of the Semantic Web, in combination with its background in artificial intelligence, makes developers assume integration will be difficult. Furthermore, some of them are unsure whether the integration cost will be worthwhile. After all, the Semantic Web is sometimes regarded as a solution in search for a problem, suffering from the chicken-and-egg syndrome and thus still waiting for a *killer application* (although the W3C maintains a list of case studies and use cases [20]). Finally, the Semantic Web is often incorrectly considered a disruptive technology that is hard to implement in an evolving ecosystem [215].

Whatever the reasons for semaphobia, adoption is often a decisive factor for technologies. Therefore, description formats anchored on a technology Web developers are already familiar with have a head start. As of 2012, more than 44 % of all APIs on ProgrammableWeb, the largest API index [64], communicate in JSON. JSON is the JavaScript Object Notation language, which quickly became popular on the Web as it was natively parsed by JavaScript, the only scripting language supported by the majority of browsers. Its simplicity and extensibility make it also an interesting target for many other environments [207].

5.5.1 SEREDASj

As the letter *j* in its name suggests, SEREDASj (*s*emantic *re*stful DAta services [137]) is a semantic description language format expressed in JSON. The motivation behind

SEREDASj is to provide simpler descriptions (in contrast to OWL-S [153] or WSMO [141]) that contain necessary semantics (such as SA-REST and MicroWSMO, see Sect. 5.2.2) in a widely used machine-targeted media type (*application/json*). SEREDASj defines a syntactic structure for JSON documents and a corresponding interpretation. It enables the representation of hypermedia links, which are not natively present in JSON. It is, however, not a strictly validated approach such as JSON schema [258], which enforces the presence of certain elements and properties in a JSON document.

The authors of SEREDASj put forward three use cases [137]. Firstly, they envision it as a means of creating documentation, both in machine-processable and human-readable form. The human-readable counterpart is generated from labels of predicates defined in ontologies, which are referenced by URI in the SEREDASj description. The benefit here is reuse—on the one hand by having a single description for humans and machines, and on the other hand by referencing to existing ontological definitions. Secondly, the goal is to enable more flexibility in Web API clients. By adding support for hyperlinks to JSON, SEREDASj descriptions become a hypermedia format through which clients can navigate a Web API in accordance with the hypermedia constraint [78]. Thirdly, SEREDASj aims to facilitate data integration, as its annotations enable the transformation of JSON into RDF. However, as we will argue below, other means to this end exist, so employing SEREDASj specifically for this last use case might prove suboptimal.

Generally speaking, one SEREDASj description is created per resource type. As SEREDASj currently only supports JSON, it is assumed that resources of this type have at least a JSON representation. The accompanying SEREDASj description documents the elements in these JSON documents by mapping them to predicates of ontologies in RDF format. This principle can be applied to the whole hierarchy of the document, including arrays, which represent multi-valued properties. Every subtree can additionally be associated with an RDF type. Next to this, SEREDASj describes the controls to navigate through the Web API in the form of URI templates [99], as well as the RDF predicates they correspond to, capturing their meaning. SEREDASj can furthermore detail the format of entities for use in PUT or POST requests.

Part of the functionality of SEREDASj is currently offered by JSON-LD [139], whose specification is currently a W3C editor's draft [218]. JSON-LD similarly provides predicate and type annotations that allow JSON data to be translated into RDF, but these annotations are included in the JSON document itself as opposed to a separate SEREDASj description document. However, the question arises whether it would not be more beneficial for servers to provide separate JSON and RDF representations of a resource, allowing clients to indicate through content negotiation with which representation they would like to proceed.

Further work on SEREDASj includes an architecture to integrate Web APIs into the Web of Data, which makes use of SEREDASj descriptions [140].

5.6 Tools

In previous sections, different solutions for describing semantics of REST APIs have been investigated. However, there are some obstacles preventing them from being widely adopted. First, writing semantic service descriptions by hand is a time consuming and tedious task. Furthermore, to model APIs, most of these approaches require some degree of expertise in Semantic Web languages such as RDF, SPARQL, and N3 in addition to the domain knowledge. Tools can play a significant role by providing a user interface to rapidly build semantic descriptions, making the complexity of formal specification transparent to the user.

The here introduced formalisms, including MicroWSMO, SA-REST and the MSM, which make HTML service descriptions machine-processable and enable the adding of semantic information, provide the means for creating semantic descriptions of Web APIs. However, without supporting tools or guidelines, developers would have to modify and enhance the descriptions manually by using a simple text/HTML editor. In addition, the complete annotation process would have to be completed manually, if there are no tools, which enable the search for suitable domain ontologies or the reuse of annotations of previously semantically described services.

5.6.1 Karma

Karma[13] [232] is a Web-based framework for integrating structured data from a variety of sources. Users can load data from relational databases, spreadsheets, delimited text files, *kml* (Keyhole Markup Language) files, and semi-structured Web pages. Users can clean and normalize data with a programming by example interface [255]. Then, Karma semi-automatically builds a semantic model of the source by mapping it a domain ontology chosen by the user [124]. Karma models each column in terms of the classes and data properties defined in the ontology, and models the relationships among columns using object properties. Once data is modeled, Karma can translate the data into a variety of formats including RDF. The semantic models also enable Karma to integrate information from multiple sources and to invoke services to compute new information.

The main goal in Karma is to make it easy and efficient for users to perform all information integration tasks. Karma enables users to perform operations on a small set of input instances, and then learns from these examples a general procedure that it can apply to all inputs. Compared to other data integration tools, Karma significantly reduces the time and effort needed to perform the data integration tasks.

Recently, Karma has been extended with the capability to build semantic descriptions of Web APIs as a foundation to compose data sources and Web services [223].

[13] http://www.isi.edu/integration/karma

In this section, we explain how it enables users to rapidly generate semantic service descriptions.

To model services, Karma first asks the user to provide samples of the API invocations URLs. This conforms to the main idea of Karma that examples are the basis to carry out the tasks. These sample URLs can also be automatically extracted from the documentation pages of the APIs. Next, the user interactively builds a semantic model of the API by mapping the service inputs and outputs to the domain ontology. Building semantic models is the central part of the approach and it is very similar to how Karma models the data from other sources. Once the semantic model is built, Karma formalizes it using a new expressive RDF vocabulary that represents both the syntactic part and the functionality of the API. Karma stores the service specifications in a triple store, enabling users to query the service models using SPARQL. Finally, Karma deploys a Linked API that consumes and produces Linked Data. The Linked API provides REST interfaces enabling users to send RDF data in the body of a POST request and get back RDF output linked to the input. In the following paragraphs, we explain each step in more detail.

The input to the system is a set of examples of the API requests. Karma parses the URLs and extracts the individual input parameters along with their values. For each invocation example, Karma calls the API and extracts the output attributes and their values from the XML or JSON response. Then, Karma joins the input and the output values into one table and shows that in a worksheet. Karma treats this table as a regular data source and applies its source modeling technique to build a semantic model of the API.

The goal of semantic modeling is to express the API functionality in terms of classes and properties defined in a domain ontology. The modeling process consists of two steps. The first step is to identify the type of data by assigning a *semantic type* to each column. A semantic type can be either an ontology class or a pair consisting of a data property and its domain. Karma uses a conditional random field (CRF) [135] model to learn the assignment of semantic types to columns of data [93]. Karma uses this classifier to automatically suggest semantic types for new data columns. If the correct label is not among the suggested labels, users can browse the ontology through a user-friendly interface to find the appropriate type. The system re-trains the CRF model after these manual assignments.

The second part of the modeling process is to extract the relationships between the inferred semantic types. Given the domain ontology and the assigned semantic types, Karma creates a graph that defines the space of all possible mappings between the source and the ontology [124]. The nodes in this graph represent classes in the ontology connected by direct and inferred properties. Once Karma constructs the graph, it computes the API model as the minimal tree that connects all the semantic types. It is possible that multiple minimal trees exist, or that the correct model of the data is captured by a non-minimal tree. In these cases, Karma allows the user to interactively impose constraints on the algorithm to build the correct model. Karma provides an easy-to-use *gui* where the user can adjust the relationships between the columns.

The models that Karma builds are themselves represented in RDF according to an ontology[14] reusing existing ontologies such as SWRL[15] and hRESTS[16] [223]. This ontology is semantically richer than WSMO-Lite[17] and Minimal Service Model (MSM) [187] because in addition to annotating each input and output with semantic types, it also explicitly represents the relationships among inputs and outputs. Another advantage of the Karma models is that they are represented in RDF, making it possible for clients to query and discover models using SPARQL. Other approaches use graph patterns to represent the service models, so it is not possible to use SPARQL to query the models.

The Karma API models are expressive enough that it would be possible to export then to other formal specifications such as LOS and MSM. This is a direction for future work [224].

Karma also has a Web server where the modeled API will be deployed as a Linked API. The Linked API implements a REST interface allowing clients to send RDF data in a POST request. One benefit of service descriptions in Karma is that they contain all the information needed to automatically execute APIs and do the required lowering and lifting, obviating the need to manually write separate instructions using formalisms such as XSLT and SPARQL. Once the Web server receives the user POST request, it uses the service description in the Linked API repository to lower the RDF data and build the invocation URL. Then, it invokes the Web API and again uses the service description to automatically lift the XML or JSON response to generate linked data. Karma is available as an open source[18] software and users can use it to model the APIs based on their own needs.

5.6.2 SWEET

To facilitate the easier adoption of semantic description of Web APIs by supporting users in their creation, KMi has developed SWEET: Semantic Web sErvices Editing Tool[19].

SWEET is developed as a Web application that can be launched in a common Web browser and does not require any installation or additional configuration. It provides key functionalities for modifying the HTML Web API descriptions in order to include markup that identifies the different parts of the API, such as operations, inputs and outputs, and also supports the adding of semantic annotations by linking the different service parts to semantic entities. As a result, SWEET enables the creation of complete semantic Web API descriptions, based on the previously introduced models, given only the existing HTML documentation. More importantly, the

[14] http://isi.edu/integration/karma/ontologies/model/current#
[15] Semantic Web Rule Language: http://www.w3.org/Submission/SWRL
[16] http://purl.org/hrests/current#
[17] http://www.w3.org/Submission/WSMO-Lite
[18] http://github.com/InformationIntegrationGroup/Web-Karma-Public
[19] http://sweet.kmi.open.ac.uk

tool hides formalism and annotation complexities from the user by simply visualizing and highlighting the parts of the API that are already annotated and produces an HTML description that is visually equivalent to the original one but is enhanced with metadata that captures the syntactical and semantic details of the APIs. The resulting HTML description also serves as the basis for extracting an RDF-based semantic Web API description, which can be published and shared in a service repository, such as iServe, enabling service browsing and search.

SWEET[20] is a Web application developed using JavaScript and Ext*gwt*[21], which is started in a Web browser by calling the host URL. It is part of a fully-fledged framework, developed within the scope of the SOA4All European project[22], for supporting the lifecycle of services, particularly targeted at enabling the creation of semantic Web API descriptions. SWEET takes as input an HTML Web page documenting a Web API and offers functionalities for annotating service properties and for associating semantic information with them. A current version of the tool can be found online[23].

SWEET is designed as a classical three-layered Web application. The architecture of SWEET consists of three main components, including the visualization component, the data preprocessing component and the annotations recommender. The visualization component is based on a model-view-controller architecture design pattern, where the model implements an internal representation of the annotated Web API, in accordance with the elements foreseen by the semantic formalisms detailed in the previous sections. In this way, every time the user adds a new annotation via the interface, the model representation of the Web API description is automatically updated. Similarly, when parts of the model representation are altered or deleted, the highlighting and visualization in the user interface is also adjusted. When the annotation process is complete, the resulting HTML and RDF Web API descriptions are generated based on the produced internal model.

The GUI of the visualization component is shown in Fig. 5.3 and it has three main panels. The HTML of the Web API is loaded in the *Navigator* panel, which implements a reverse proxy that enables the communication between the annotation functions and the HTML by rerouting all sources and connections from the original HTML through the Web application. Based on this, the HTML *dom* of the RESTful service can freely be manipulated by using functionalities of the *Annotation Editor* panel. The current status of the annotation is visualized in the form of a tree structure in the *Semantic Description* panel, which is implemented by automatically synchronizing the visualization of the service annotation with an internal model representation, every time the user manipulates it.

In addition to these three main panels, SWEET offers a number of supplementary useful functionalities. It guides the user thorough the process of marking service properties with hRESTS tags, by limiting the available tags depending on the current

[20] http://sweet.kmi.open.ac.uk
[21] http://extjs.com/products/gxt/
[22] SOA4AllEU project FP7 - 215219, http://soa4all.eu/
[23] http://sweetdemo.kmi.open.ac.uk/soa4all/MicroWSMOeditor.html

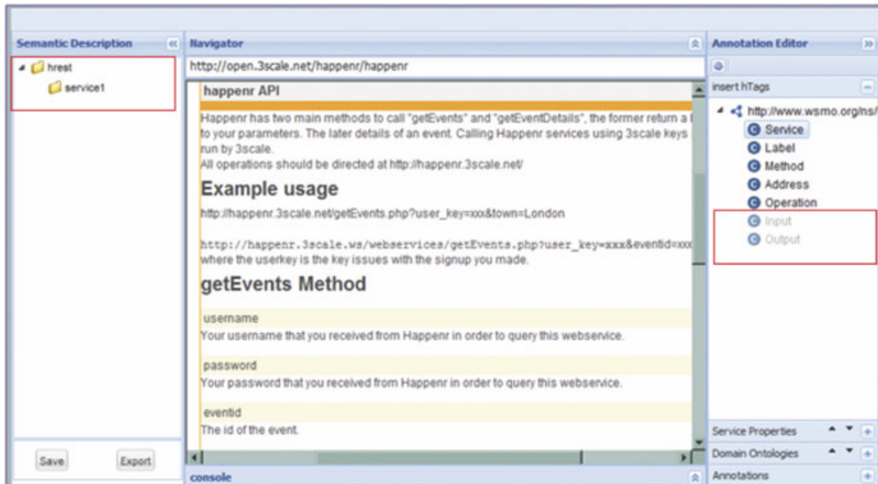

Fig. 5.3 SWEET: Inserting hRESTS tags

state of the annotation. This implements measures for reducing possible mistakes during the creation of annotations. In addition, based on the hRESTS tagged HTML, which provides the structure of the Web API, the user can link service properties to semantic content. This is done by selecting service properties, searching for suitable domain ontologies by accessing Watson [59] in an integrated way, and by browsing ontology information. Based on this details the user can decide to associate a service property with particular semantic information by inserting a SA-WSDL model reference tag.

In summary, SWEET takes as input the HTML Website description of the Web API, which is loaded in the central panel, and returns a semantically annotated version of the HTML or a RDF semantic description. In order to do this the user needs to complete the following four mains steps:

1. Identify service properties (service, operation, address, HTTP method, input, output and label) by inserting hRESTS tags in the HTML service description.
2. Search for domain ontologies suitable for annotating the service properties.
3. Annotate service properties with semantic information.
4. Save or export the annotated Web API.

The first step can easily be completed by simply selecting the part of the HTML, which describes a particular service property, and clicking on the corresponding tag in the *inset hTags* pane. In the beginning, only the *Service* node of the hRESTS tree is enabled. After the user marks the body of the service, additional tags, such as the *Operation* and *Method*, are enabled. In this way, the user is guided though the process of structuring the RESTful service description and is prevented from making annotation mistakes. After the user structures the HTML description and identifies all service properties, the adding of semantic information can begin. The new version

of SWEET, just like the bookmarklet, supports users in searching for suitable domain ontologies by providing an integrated search with Watson [59]. The search is done by selecting a service property and sending it as a search request to Watson. The result is a set of ontology entities, matching the service property search. Once the user has decided, which ontology to use for the service property annotation, he/she can do an annotation by selecting a part of the service HTML description and clicking on *Semantic Annotation* in the *Service Properties* context menu. This results in inserting a model attribute and a reference pointing to the URI, of the linked semantic concept.

The resulting descriptions can be directly posted to iServe [186] or can be re-posted on the Web. The use of the microformat tags enables the automated search and crawling for APIs, since it serves as a basis for distinguishing simple HTML websites from Web API descriptions. Furthermore, when posted to iServe, the Web API descriptions can be browsed and searched alongside with WSDL-based services. Since the semantic Web API descriptions use SA-WSDL-like annotations in combination with the WSMO-Lite service ontology, both WSDL-based services and APIs can be retrieved by using the same queries. For example, a search query for music services would return the Last.fm description as well as all other APIs or services related to music. As a result, all type of services, can be retrieved in a unified way.

5.7 Open Problems and Future Work

5.7.1 Cross-Origin Resource Sharing (CORS)

The XMLHttpRequest specification [19] defines an API that provides scripted client functionality for transferring data between a client and a server. In today's common Web applications like online spreadsheets, word processors, presentation tools *etc.*—and even more in so-called mash-up applications[24]—the majority of data transfers between server and client happen based on XMLHttpRequest. An important security aspect of XMLHttpRequest, however, is the so-called Same Origin Policy (SOP). This policy permits scripts running on pages originating from the *same* site to access each other's methods and properties with *no* specific restrictions, however, *prevents* access to most methods and properties across pages on *different* sites. While providing at least some protection from rogue Web pages accessing private data, SOP also has severe implications for cases where cross-origin data transfers are actually legit. Past attempts to legally circumvent SOP include using proxy servers, Adobe Flash, and JSON-P,[25] however, more recently, the tendency goes in the direction of properly handling cross-origin resource sharing (CORS) through a mechanism documented in a em W3C Working Draft [236]. The CORS standard works by adding new HTTP

[24] Web applications that combine data from multiple sources to create new services, many of them listed in [64].

[25] First documented appearance of JSON-P in Bob Ippolito's December 2005 blog post: http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/.

headers that allow servers to serve resources to permitted origin domains. Browsers support these headers and enforce the restrictions they establish. While not all APIs support CORS yet, there is a remarkable momentum of Web API and data providers in general to open up their data and becoming CORS-enabled[26].

5.7.2 *Authentication*

The IEEE defines[27] authentication as *the act of confirming the truth of an attribute of a datum or entity. This might involve confirming the identity of a person [...], or assuring that a computer program is a trusted one*. In the world of APIs, simple authentication paradigms include (but are not limited to) API keys (codes passed in by computer programs calling an API to identify the calling program, its developer, or its user to the API provider), HTTP Basic authentication, HTTP headers, or HTTP cookies. In recent times, different versions of the *authorization* protocol OAuth[28] gain traction as the *de facto* default standard for authorization. *Authentication* is the mechanism whereby systems may securely identify their users. *Authorization*, by contrast, is the mechanism by which a system determines what level of access a particular authenticated user should have to secured resources controlled by the system[29]. In the case of OAuth, if the user grants access to a resource, the application can retrieve the unique identifier for establishing the identity in turn by using the particular API calls, and thus effectively enabling pseudo-authentication using OAuth.

[149] provides as extensive overview of currently used authentication approaches, the required credentials, ways of transmitting the credentials and used authentication mechanisms. The provided solution is based on defining authentication extensions to the MSM defined in Sect. 2.

5.7.3 *CORS and Authentication in API Descriptions*

To the best of our knowledge, neither authentication nor CORS are covered by the before-mentioned API description formats, the honorable exception being the Web Application Description Language (WADL [104]), where (authentication) HTTP headers can be described for API query parameters. While the implementation status of CORS can be determined at runtime by examining a sample API request and checking for the existence of the particular HTTP header, there is no general way to discover the authentication requirements of an API at runtime. In consequence, both CORS and authentication and ways to semantically describe them remain a field for future research.

[26] http://enable-cors.org/
[27] http://technav.ieee.org/tag/2585/authentication
[28] http://oauth.net/
[29] http://www.duke.edu/~rob/kerberos/authvauth.html

5.8 Conclusion

We have surveyed the current state-of-the-art in descriptions of Web APIs and classified the various approaches. The main strength of RESTful APIS, the flexibility which with the APIs can be designed and deployed, at the same time burdens client application developers with the manual work of understanding, interpreting, and reconciling the various approaches to API design. Almost all of today's Web APIs come with a textual description, lacking coherence. A little structure in architecting and documenting the APIs could greatly benefit application developers and reduce the amount of manual effort required when integrating multiple APIs.

Acknowledgments R. Verborgh and R. Van de Walle are funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (iMinds), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union. A. Harth and S. Speiser acknowledge the support of the European Commission's Seventh Framework Programme FP7/2007-2013 (PlanetData, Grant 257641). S. Stadtmüller has been supported by a Software Campus grant.