May Contain Nuts: The Case for API Labels

Cesare Pautasso[1] and Erik Wilde[2]

[1] Software Institute, Faculty of Informatics, USI, Lugano, Switzerland
[2] CA Technologies, Zürich, Switzerland

Abstract. As APIs proliferate, managing the constantly growing and evolving API landscapes inside and across organizations becomes a challenge. Part of the management challenge is for APIs to be able to describe themselves, so that users and tooling can use descriptions for finding and filtering APIs. A standardized labeling scheme can help to cover some of the cases where API self-description allows API landscapes to become more usable and scalable. In this paper we present the vision for standardized API labels, which summarize and represent critical aspects of APIs. These aspect allow consumers to more easily become aware of the kind of dependency they are going to establish with the service provider when choosing to use them. API labels not only summarize critical coupling factors, but also can include claims that require to be validated by trusted third parties.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

APIs are the only visible parts of services in API-based service landscapes. The technical interface aspect of APIs has been widely discussed with description languages such as WSDL, RAML, and Swagger/OpenAPI. The non-functional aspects are harder to formalize (e.g., see the survey by García et al. [8]) but can also benefit from a framework in which information can be represented and used.

The idea of "API Labels" is equivalent to that of standardized labeling systems in other product spaces, for example for food, for device energy consumption, or for movie/games audience ratings. In these scenarios, labels enable consumers to understand a few key (and often safety-critical) aspects of the product. This framework is not intended to be a complete and exhaustive description of the product. Instead, it focuses on areas that are important and helpful to make an initial product selection. The assumption is that the information found on the label can be trusted, so that consumers can make decisions based on labels which are correct and do not contain fraudulent information.

In the API space, numerous standards and best practices have evolved how APIs can be formally described for machine processing and/or documented for human consumption [14] (e.g., WSDL [4], WADL [9], RESTdesc[24], hRESTS [12], RADL [19], RAML, Swagger/OpenAPI [22], SLA★ [10], RSLA [21], SLAC [23]

just to mention a few). However, there still is some uncertainty how to best combine and summarize these, and how to use them so that API description, documentation, and labeling can be combined. This paper proposes the API Labels Framework (AFL) to introduce API labels as a synthesis of existing API descriptions combined with additional metadata which can help customers assess several practical qualities of APIs and their providers and thus be useful to reduce the effort required to determine whether an API can be worthy of consideration.

The main motivation for labeling APIs is probably not so much about a way to enable providers to put marketing labels on their APIs nor is it a way to summarize information that is already present in existing formal API descriptions. Instead, it is about providing assurances for API consumers about crucial characteristics of the service behind the API that may not be visible on its surface.

The rest of this paper is structured as follows. In Sec. 2 we present general background on labeling and related work which has inspired the current paper. In Sec. 3 we apply the concept of labeling to APIs and discuss how to use OpenAPI Link Objects and Home Documents to make API labels easy to find. We discuss the issue of how to establish trust for API labels in Sec. 4 and then introduce different label types in Sec. 5. The following Sec. 6 provides a non-exaustive set of label type examples. The problem of discovering labels and ensuring that they can evolve over time are identified in Sec. 7.2. Finally we draw some conclusions in Sec. 8 and outline possible directions for future work in Sec. 9.

2 Background and Related Work

Labeling helps to identify, describe, assess and promote products [13]. Branding and labeling contribute to differentiate competing products by assuring the consumer of a guaranteed level of quality or by restoring consumer's confidence after some negative publicity leading to a loss of reputation. More specifically, food labeling has also been used to educate consumers on diet and health issues [5]. Labeling can thus be used as a marketing tool [1] by providers or as a provider selection tool by consumers [2].

This work is inspired by previous work on designing simplified privacy labels of Web sites [11] based on the now discontinued P3P standard [7]. It shares similar goals to provide a combined overview over a number of "API Facts". However, one important difference is that P3P was a single-purpose specification intended to standardize everything required for embedding privacy labels. It thus had fixed methods to locate privacy policies (four variations of discovering the policy resource), fixed ways how those were represented (using an XML-based vocabulary), and a fixed set of acceptable values (also encoded into the XML vocabulary) to be used in these policies.

The work presented in this paper is bigger in scope, and on the framework level. As such, we do not authoritatively prescribe any of the aspects that P3P was defining. Instead, we are assuming that with organizations and user groups

using API labels, certain patterns will emerge, and will be used inside these communities. We can easily envision a future where our framework is used as a foundation to define a more concrete set of requirements, but this is out of scope for this paper, and most likely would benefit substantially from initial usage and feedback of the API label framework presented here.

3 Labeling APIs

The idea of API labels is that they apply not just to individual resources, but to a *complete API*. Many APIs will provide access to a large set of resources. It depends on the API style how APIs and individual resources relate [18]. In the most popular styles for APIs today, which are HTTP-based, the API is established as a set of resources with distinct URI identities, meaning that the API is a set of (potentially many) resources. One exception to this are RPC-oriented API styles (such as the ones using SOAP, grpc or GraphQL) which "tunnel" all API interactions through a single "API endpoint". In that latter case, there is no such thing as a "set of HTTP-oriented resources establishing the API", but since we are mostly concerned with today's popular HTTP-based styles, the question of the scope of API labels remains relevant.

Applications consuming APIs are coupled to them, and the choice of API to be consumed introduces critical dependencies for consumers [17]. Consumers need to be made aware about non-functional aspects, concerning the short-term availability and long-term evolution of API resources [15]. Likewise, when a resource is made available by a different API, different terms of service may apply to its usage.

From the consumer point of view, the concept of an "API boundary" can seem arbitrary or irrelevant, or both. API consumers most importantly want to implement applications. To do so, they need to discover, select and invoke one or more APIs. However, even when from the strict application logic point of view the "boundary" between APIs may not matter (applications will simply traverse resources either driven by application logic or by hypermedia links), it still may be relevant for non-functional aspects, such as when each API resource is made available by a different provider and therefore different terms of service apply to its usage.

Generally speaking, the Web model is that applications use various resources to accomplish their goals, and these resources often will be provided by more than one API. In this case the question is how it is possible to get the API labels for every resource, if applications want to do so. What is the scope of API labels, and how is it possible, starting from *any* resource of an API, to find its API labels? And how can an application know when traversing resources that it traverses an "API boundary"? The Web (and HTTP-based URIs) has no built-in notion to indicate "API boundaries", so the question is how to establish such a model.

It seems wasteful to always include all API label information in all resources, given that in many cases, applications will not need this information and thus it

would make API responses unnecessarily large. However, there are approaches how this can be done in more efficient ways, and currently there are two solutions available (OpenAPI Link Objects and Home Documents). It is important to keep in mind that it is up to an API designer to decide if and how they will use these techniques to make labels easy to find.

3.1 OpenAPI Link Objects

The API description language *OpenAPI* (formerly known as *Swagger*) has added the concept of a *link object* with its first major release under the new name, version 3.0. Essentially, link objects are links that are defined in the OpenAPI description, and then can be considered to be applicable to specific resources of the API. In essence, this creates a shortcut mechanism where these links are factored out from actual API responses, and instead become part of the API description.

It is important to keep in mind that because of this design, the actual links in the OpenAPI link object never show up in the API itself; instead they are only part of the OpenAPI description. This design allows OpenAPI consumers to use these links without producing any runtime overhead, but it makes these links "invisible" for anybody not using the OpenAPI description and interpreting its link objects.

This design of OpenAPI thus can be seen as effective optimization, because it creates no runtime overhead. On the other hand, it limits self-descriptiveness and introduces substantial coupling by making the links in link objects exclusively visible to clients knowing and using the OpenAPI description.

For this reason, we believe that in environments where this coupling has been introduced already, OpenAPI link objects may be a good solution. This can be any environment where the assumption is that API consumers always know the OpenAPI descriptions of the APIs they are consuming. This may be a decision that is made in certain organizations or communities, but cannot be considered a design that is used in unconstrained API landscape.

In unconstrained API landscapes, it seems that the coupling introduced by making the knowledge and usage of all OpenAPI descriptions mandatory is substantial, and may be counterproductive to the self-describing and loosely coupled consumption of APIs. If the design goal is to focus on self-description and loose coupling, then OpenAPI link objects probably are not the best choice, and instead the approach of *home documents* may be the better one.

3.2 Home Documents

An alternative model to that of OpenAPI is established by the mechanism of *home documents* [16]. The idea of home documents is that there is a "general starting point" for an API. This starting point can provide a variety of information about the API, including information about its API labels. The home document then can be linked to from API resources, and there is a specific `home` link relation that is established as part of the home document model.

Using this model, all resources of an API can provide *one* additional link, which is to the API home document. The home document then becomes the starting point for accessing any information about the API, including an API's labels. This model means that there is an overhead of one link per resource. However, given modern mechanisms such as HTTP/2.0 header compression, it seems that this overhead is acceptable in the majority of cases, even if that link is not so much a functional part of the API itself, but instead provides access to metadata about the API.

One of the advantages of the idea of home documents and providing `home` links for resources is that this makes the API (or rather its resources) truly self-describing: Consumers do not need any additional information to find and use the information about an API's home document.

One downside to this model is that home documents are not yet a stable standard used across many APIs. The draft has been around for a while and has evolved over time, but it is not guaranteed that it will become a stable standard. One other hand, since this work is rooted in general Web architecture, even without the specification being a stable standard already using it is acceptable, and in fact this is how many IETF standards are conceived: drafts are proposed, already adopted by some, and the eventual standard then is informed by gathering feedback from those who already have gained experience with it.

4 Trusting API Descriptions and Documentations

API labels provide a human-readable format to summarize API descriptions including hyperlinks to relevant documentation and specifications. API labels are also meant to be machine processable to provide the basis for automated support for API landscape visualization and filtering capabilities.

One example for this are the link relation types for Web services [26]. These could be readily used as API labels (if they are made discoverable through the general API label mechanism). Some of the resources are likely just human-readable (for example API documentation provided as PDF), while other resources might be machine-readable and to some extent even machine-understandable (for example API description provided as OpenAPI which can be used by testing and documentation generation tools).

API labels are not meant to provide a complete specification of APIs and replace existing languages and service discovery tools. Instead, they are designed to include information that is currently not found in API descriptions as written by service providers, because this information may include claims that need to be verified by trusted third parties. Additionally, the summary described in the label can lead to more detailed original sources that can be used to confirm the validity of the summarized information.

While it is in a provider's best interest to provide a correct representation of its APIs functional characteristics (operation structure, data representation formats, suggested interaction conversations) so that clients may easily consume the API appropriately, questionable providers may be tempted to misrepresent

some of the Quality of Service levels they may be capable of guaranteeing. Hence labeling APIs could provide the necessary means to certify and validate the provided API metadata information complementing other means to establish and assess the reputation of the API provider [3]. This is a rather challenging task that would require to deal with a number of non-trivial issues.

For example, how would consumers establish trust with a given API label certification authority? Is one centralized authority enough or should there be multiple ones taking into advantage the decentralized nature of the Web [6]? If multiple parties can certify the same API, how should consumers deal with conflicting labels? How to ensure labels can be certified in an economically sustainable way (are consumers willing to pay to get verified labels?) without leading to corruption (providers are willing to pay to get positive labels)? How would the authority actually verify the QoS claims of the provider? How to avoid that a provider obtains good results when undergoing a certification benchmark but poor performance during normal operations when servicing ordinary customer requests? How to ensure API labels are not tampered with? Should labels be signed by reference or by value?

While it is out of scope of this paper to deal with all of these issues, we believe some form of delegation where APIs reference labels via links to label resources hosted by third parties will be one of the key mechanisms to enable trust into certified API labels. This way, even if the label value itself is not provided by the API, but by using the delegation mechanism, we could still make it discoverable through the API.

5 Label Types

In order to be understandable, labels must follow a framework of well-defined types that can be "read" as API labels. Some of these may already exist as evolving or existing standards. The link relations for Web services discussed in the previous section can be considered potential API labels that are defined in an evolving standard. An example for an existing standard is the `license` link relation defined in RFC 4946 [20], which is meant to convey the license attached to resources made available through a service.

A label type identifies the kind of label information that is represented by attaching a label of this type. In principle, there are three different ways of how label types can communicate label information to consumers:

– By Value: If the label is simply an identifier, then the meaning of the label is communicated by the label value itself. The question then is what the permissible value space is (i.e., which values can be used to safely communicate a well-defined meaning between label creators and label readers). The value space can be fixed and defined by enumerating the values associated with the label type, or it can be defined in a way so that it can evolve. This second style of managing an evolving value space often is implementing through registries [25], which effectively decouples the definition of the label type and the definition of its value space.

– By Format: If the label is intended to communicate its meaning by reference, then it will link to a resource that represents the label's meaning. It is possible for label types to require that the format is always the same, and must be used when using that label. This is what P3P (the example mentioned earlier) did, by defining and requiring that P3P policies always must be represented by the defined format. This approach allows to build automation that can validate and interpret labels, by depending on the fact that there is one format that must be used for a given label type.

– By Link: It is also possible to not require the format being used. This is the most webby and open-ended approach, where a label links to a resource representing the label's value, but the link does not pre-determine the format of the linked resource. This approach has the advantage that label value representations can evolve and new ones can be added when required, but it has the disadvantage that there is no a priori interoperability of label producers and label consumers.

Returning to the examples given above, it becomes obvious that the existing mechanisms discussed so far that could be considered to be used as API labels already use different approaches from this spectrum. The link relation for licenses [20] is based on the assumption that a license is identified by value, thus requiring licenses to be identified by shared URI identifiers. P3P [7] defines its own format that has to be used for representing P3P labels. The link relations for Web services [26] identify information by link, and do not constrain the format that has to be used with those link relations.

6 API Label Examples

In this section we collect a preliminary list of API label types and values, characterizing several technical and non-technical concepts [27] which are meant to assist consumers during their API selection process. We have compiled this list based on the relevant literature, our experience, including feedback from our industry contacts.

– *Invocation Style:* This label defines on a technical level which kind style is required for clients to invoke the API. We distinguish between Synchronous RPC, Synchronous Callbacks, Asynchronous Events/Messages, REST, and Streaming.

– *Protocol Interoperability:* Which are the interaction protocols supported by the API? Which versions of the protocols? Examples values: SOAP, HTTP, GraphQL

– *Privacy:* Where is the data managed by the API stored? While clients do not care whether their data is stored in SQL or XML, they do worry whether their data is located in a different country and thus subject to different regulations.

– *Service Level Agreement:* Does an SLA explicitly exist? If it does: how is it enforced? are there penalties for violations? can it be negotiated? This helps to roughly distinguish between APIs without SLAs from APIs having an explicitly (formally or informally) defined SLA, which can be further annotated to highlight whether service providers make serious efforts to stand behind their

promises and whether they are willing to adapt to client needs by negotiating the terms of the agreement with them as opposed to offering a number of predefined usage plans.

– *Pricing:* Also related to SLA, clients want to know: whether there a free price plan? Can the API paid price plans be considered as cheap, reasonable, or expensive? This label needs to be computed based on the client expectations or by comparing with similar APIs.

– *Availability Track Record:* Does the API provider explicitly promises high availability? How well does the promise (e.g., "five nines" or 99.999%) matches the reality? Is the API provider's availability improving or getting worse? Additionally, clients need to know how to set their timeouts before giving up and determining that the API is no longer available. The Availability Track Record should label APIs for which such information is explicitly found in the corresponding SLA.

– *Maturity/Stability:* The Maturity label should provide a metric to determine whether the API has reached flying altitude and can be considered as mature enough, i.e., it is likely to be feature complete and stable during the entire lifecycle of clients consuming it. This can be inferred from versioning metadata, or some kind of metric summarizing the API version history (e.g., the number of changes over time, or how many alternative versions of the same API are supported in parallel by the provider). Conversely, if APIs are not yet mature and unstable, clients would benefit from knowing how much time they have to react to breaking API changes. Different providers may allow different amounts of time between announcing changes and carrying them out. In a similar way, as APIs eventually disappear, does the provider support some notion of *sunset* metadata? Are API features first deprecated and eventually retired, or does the API provider simply remove features without any warning?

– *Popularity:* How many clients are using the API? Is this the mostly used API within the ecosystem/architecture? is it in the top 10 APIs based on daily traffic? or only very few clients rarely invoke it?

– *Alternative Providers:* Are there alternative and competing providers for the API? or there exists only one monopolistic provider? How easy is it to replace the service provider of the API? How easy is it to find a replacement API within minimal differences from the current one?

Additional label types describing energy consumption, sustainability, quality management (e.g. ISO 9001 compliance) or trust certificates are possible.

7 A Recipe for API Labels

As mentioned already, the exact way of how to implement labels is not yet standardized. In this paper, we discuss the parts that need to be in place to use API labels, but we do not prescribe one single correct way. In order to summarize these parts, and to give organizations looking at using API labels a useful starting point, we are summarizing the required parts in an "API label landscape". We also recommend specific ways of solving these individual issues. In particular,

Section 7.1 provides methods to make labels findable, and Section 7.2 provides methods to manage the types and the values of those findable labels so that the set of labels used in an API landscape can organically grow over time.

7.1 Findable Labels

In order for API labels to be usable and useful, they must be findable. One possibility is to manage them separate from APIs themselves, but this approach is likely to let APIs and their labels go out of sync easily. A more robust approach is to make API labels parts of APIs themselves, which allows labels to be managed and updated by the APIs themselves, and also allows labels to be found and accessed by those that have access to these APIs.

Using such an approach, making API labels findable amounts to allowing them to be accessed through the API. For this to be consistent across APIs, there need to be conventions that are used across APIs to find and access labels. What these conventions look like, depends on the style and technology of APIs. For HTTP APIs that are based on the resource-oriented or the hypermedia style of APIs this amount to providing resources that represent label information.

In terms of currently available practices, using home documents as described in Section 3.2 works well, if it is acceptable as a general API guideline to require APIs to provide home documents. If it is, labels still need to be made discoverable from that home document. We are suggesting to represent labels in a way that represents a set of labels, and that has the ability to "delegate" label representation to third parties, so that that scenarios like the ones discussed in Section 4 can be implemented.

7.2 Extensible Label Sets

Once there is a defined way how labels can be found for APIs and, as suggested above, through the APIs themselves, then the next question is what types of labels can be found (Section 6 suggests a starting set of label types). It is likely that the set of label types is going to evolve over time, so the question is not only which types of labels to support, but also how to manage the continuous evolution of that set of types.

A flexible way to manage label sets is to use registries [25], as mentioned in Section 5. Once the necessary registry infrastructure is in place, registries need to be combined with policies so that values in the registry have a well-defined way how they evolve. For API label types and their corresponding values, a rather standard set of policies for registry management would most likely work well:

– Initial Set Any API label landscape will start with a set of initial label types. This set should be the "minimal viable product", meaning that it is more important to get API label use off the ground, than to have the perfectly curated set of label types. Likewise, the initial values of each label type will be chosen among values with a fixed and well-understood meaning.

– Additions after community review and consensus: The label landscape will continually grow, with new label types and values being added as required. Additional label types should have some motivation documented, and that motivation should be the starting point for a community review. If there is sufficient consensus to add the type, it is added to the set of existing label types. In a similar way, new values should undergo some review so that they broadly follow the general idea of the label type, and ideally do not created overlaps or conflicts with existing entries.

– Semantics of registered label types and values do not change: API labels should always mean the same, so the meaning of an API label type should never be changed. Once it has been registered, users will start using it and will depend on its registered meaning, so changing its meaning would be a breaking change for all uses of the API label. One exception to this rule is that it is possible to clarify and correct the meaning of a registered label value, but this should be used very carefully because any change being made to a label values meaning should retroactively invalidate or change the way how a label value has been used before.

– Registered label types and values cannot be removed, but can be retired: Label types should never change meaning, but their usage may not be supported or required anymore. If that is the case, there should be a mechanism how a label type or value can be marked as *deprecated* in the registry, so that it becomes clear that this label may appear, but that it should not be actively used anymore. As opposed to removing it from the registry, the semantics of the deprecated value remain registered and available, allowing everybody to still look up what an assigned label type or value means. However, the status also makes it clear that this value should not be used for new labels.

While this recipe for managing label types and values is not the only possible way, it ensures that label management can evolve, and does not suffer from breaking changes along the way. This is thanks to the combination of stable semantics, and the policies on how to evolve them. Because this is a general pattern how to achieve robust extensibility, a very similar recipe can be used to manage the evolution of the value space of individual labels.

8 Conclusion

In this position paper we have made the case for API Labels. Labeling APIs is driven by the real world needs of consumers to quickly assess the main quality attributes of an API and its provider, which are likely to affect the consumer application built using the API in the long term. We have proposed the *API Label Framework (ALF)*: a framework based on the "API the APIs" principle to make API self-descriptive by attaching API labels as metadata to API resources. We also included an initial proposal for a number of possible label types. Some of these can be automatically derived by summarizing information found in API descriptions written by the providers. Other require some external input by a third-party authority. For API Labels to become a trusted mechanism

for API annotation, comparison and selection, there needs to be a verification and validation process which guarantees that consumers can trust the "facts" mentioned in the label.

9 Future Work

As part of future work we plan to make labels self-describing by creating identifiers for each label type you want to support and make label values self-describing by clearly defining the value space for each label. Tooling will be required to automatically extract labels and validate the consistency of labels with the corresponding detailed API descriptions so that API owners can easily test their labels and see how they are working. Once a number of machine-readable API labels become available, tooling to crawl labels will make it easier for developers to explore the "label graph" of the labels that one or more API providers define.

Also policies around label changes will need to be established so that it is well-defined when and how to expect label updates and how these are communicated by tracking the history of a given API. Given that label types and values themselves will likely evolve, it will be important to determine how the set of possible known values is defined and where can the identified label types can be reused from. Registries [25] for API labels and possibly their value spaces are like to play a key role for addressing this challenge.

References

1. Atkinson, L., Rosenthal, S.: Signaling the green sell: the influence of eco-label source, argument specificity, and product involvement on consumer trust. Journal of Advertising **43**(1), 33–45 (2014)
2. Becker, T.: To what extent are consumer requirements met by public quality policy? In: Quality policy and consumer behaviour in the European Union., pp. 247–266. Wissenschaftsverlag Vauk Kiel KG (2000)
3. Bidgoly, A.J., Ladani, B.T.: Benchmarking reputation systems: A quantitative verification approach. Computers in Human Behavior **57**, 274 – 291 (2016). https://doi.org/https://doi.org/10.1016/j.chb.2015.12.024
4. Booth, D., Liu, C.K.: Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. World Wide Web Consortium, Recommendation REC-wsdl20-primer-20070626 (June 2007)
5. Caswell, J.A., Mojduszka, E.M.: Using informational labeling to influence the market for quality in food products. American Journal of Agricultural Economics **78**(5), 1248–1253 (1996)
6. Chu, Y.H., Feigenbaum, J., LaMacchia, B., Resnick, P., Strauss, M.: REFEREE: Trust management for Web applications. Computer Networks and ISDN systems **29**(8-13), 953–964 (1997)
7. Cranor, L.F.: Web Privacy with P3P. O'Reilly & Associates, Sebastopol, California (September 2002)
8. García, J.M., Fernandez, P., Pedrinaci, C., Resinas, M., Cardoso, J.S., Cortés, A.R.: Modeling Service Level Agreements with Linked USDL Agreement. IEEE Trans. Services Computing **10**(1), 52–65 (2017). https://doi.org/10.1109/TSC.2016.2593925

9. Hadley, M.: Web Application Description Language (WADL). Tech. Rep. TR-2006-153, Sun Microsystems (April 2006)
10. Kearney, K.T., Torelli, F., Kotsokalis, C.: SLA★: An abstract syntax for Service Level Agreements. In: Proc. of the 11th IEEE/ACM International Conference on Grid Computing (GRID). pp. 217–224 (2010)
11. Kelley, P.G., Bresee, J., Cranor, L.F., Reeder, R.W.: A nutrition label for privacy. In: Proceedings of the 5th Symposium on Usable Privacy and Security. p. 4. ACM (2009)
12. Kopecký, J., Gomadam, K., Vitvar, T.: hRESTS: An HTML Microformat for Describing RESTful Web Services. In: 2008 IEEE/WIC/ACM International Conference on Web Intelligence. pp. 619–625. Sydney, Australia (December 2008). https://doi.org/10.1109/WIIAT.2008.469
13. Kotler, P.: Marketing management: analysis, planning, implementation and control. Prentice Hall (1997)
14. Lethbridge, T.C., Singer, J., Forward, A.: How software engineers use documentation: the state of the practice. IEEE Software **20**(6), 35–39 (Nov 2003). https://doi.org/10.1109/MS.2003.1241364
15. Li, J., Xiong, Y., Liu, X., Zhang, L.: How Does Web Service API Evolution Affect Clients? In: 2013 IEEE 20th International Conference on Web Services(ICWS). pp. 300–307 (June 2013)
16. Nottingham, M.: Home Documents for HTTP APIs. Internet Draft draft-nottingham-json-home-06 (August 2017)
17. Pautasso, C., Wilde, E.: Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In: Quemada, J., León, G., Maarek, Y.S., Nejdl, W. (eds.) 18th International World Wide Web Conference. pp. 911–920. ACM Press, Madrid, Spain (April 2009)
18. Pautasso, C., Zimmermann, O.: The Web as a Software Connector: Integration Resting on Linked Resources. IEEE Software **35**(1), 93–98 (2018)
19. Robie, J., Sinnema, R., Wilde, E.: RADL: RESTful API Description Language. In: Kosek, J. (ed.) XML Prague 2014. pp. 181–209. Prague, Czech Republic (February 2014)
20. Snell, J.M.: Atom License Extension. Internet RFC 4946 (July 2007)
21. Tata, S., Mohamed, M., Sakairi, T., Mandagere, N., Anya, O., Ludwig, H.: RSLA: A service level agreement language for cloud services. In: Proc. of the 9th International Conference on Cloud Computing (CLOUD2016). pp. 415–422. IEEE (2016). https://doi.org/10.1109/CLOUD.2016.60
22. The Open API Initiative: OAI. https://openapis.org (2016), https://openapis.org/
23. Uriarte, R.B., Tiezzi, F., De Nicola, R.: SLAC: A formal service-level-agreement language for cloud computing. In: UCC. pp. 419–426. IEEE (December 2014)
24. Verborgh, R., Steiner, T., Deursen, D.V., Coppens, S., Vallés, J.G., de Walle, R.V.: Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web. In: Alarcón, R., Pautasso, C., Wilde, E. (eds.) Third International Workshop on RESTful Design (WS-REST 2012). pp. 33–40. Lyon, France (April 2012). https://doi.org/10.1145/2307819.2307828
25. Wilde, E.: The Use of Registries. Internet Draft draft-wilde-registries-01 (February 2016)
26. Wilde, E.: Link Relation Types for Web Services. Internet Draft draft-wilde-service-link-rel-06 (August 2018)
27. Wilde, E.: Surfing the API Web: Web Concepts. In: 27th International World Wide Web Conference. ACM Press, Lyon, France (April 2018)