Towards Understanding of Structural Attributes of Web APIs Using Metrics Based on API Call Responses

Andrea Janes, Tadas Remencius, Alberto Sillitti, and Giancarlo Succi

Free University of Bozen-Bolzano, Bolzano, Italy
{andrea.janes,tadas.remencius,alberto.sillitti,giancarlo.succi}@unibz.it

Abstract. The latest trend across different industries is to move towards (open) web APIs. Creating a successful API, however, is not easy. A lot depends on consumers and their interest and willingness to work with the exposed interface. Structural quality, learning difficulty, design consistency, and backwards compatibility are some of the important factors in this process. The question, however, is how one can measure and track such attributes. This paper presents the beginnings of a measurement framework for web APIs that is based on the information readily available both to API providers and API consumers - API call responses. In particular, we analyze the tree-based hierarchical structure of JSON and XML data returned from API calls. We propose a set of easy-to-compute metrics as a starting point and describe sample usage scenarios. These metrics are illustrated by examples from some of the popular open web APIs.

1 Introduction

More and more businesses are taking advantage of the so-called API Economy every day. Powered by the (open) web APIs, this new way of doing business [1] offers a number of exciting opportunities [2], such as getting additional value of your company's business assets and fostering innovation from third-parties [3] - all at a very low cost[1].

The number of available web APIs has been increasing in a rapid manner. For example, the registry of open web APIs at *programmableweb.com* [4] shows close to exponential growth rate (Figure 1).

In principle, a web API can be implemented both as a proprietary and as an open-source software application, yet, conceptually, it represents a middle-ground between these two models. The code of the API or at least the source of the data/services it exposes is typically hidden from the API consumers, giving protection to the API provider and maintaining its ownership of the valuable business assets. The exposed API interface, on the other hand, becomes the open

[1] Compared to normal effort and investment needed to enact new business initiatives, such as development of new products/features or starting marketing campaigns for reaching new markets.

L. Corral et al. (Eds.): OSS 2014, IFIP AICT 427, pp. 83–92, 2014.

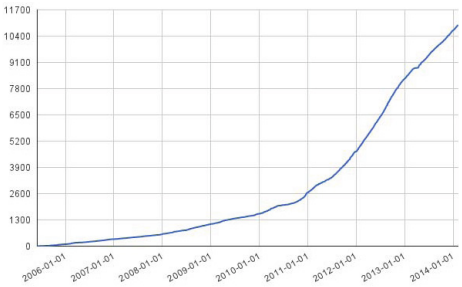


Fig. 1. Open web API growth rate, as extracted from programmableweb.com [4]

pseudo-code that is generally available [1] to third-parties to use and extend as they please. For example, consumers can combine APIs from different providers to offer completely new products with very little effort required. It's a model that follows a *win-win* approach, enabling both providers and consumers to get benefits from the exposed API while maintaining a certain degree of separation and independence from each other. Success of one side means greater chances of success for the other. This makes the risk of one of the parties abusing the work of the other much lower.

As one could expect, however, the API Economy comes with its own set of challenges and difficulties [2], [5]. One of those is how to create a successful API. This includes being able to attract API consumers and providing the right level of help and support so that they remain satisfied. Bad interface design can make the API very difficult to learn and to use, thereby hindering its adoption. Failing to maintain backwards compatibility or introducing breaking changes can not only make existing consumers abandon the API but can also prevent arrival of new ones, because of the damage to the reputation of the provider.

From the technical (development) side, implementation of web APIs is not much different from implementation of other type of software solutions. In fact, because the code of the API is a *black-box* to consumers, the way how it is actually implemented is not that important - what matters is that it works according to current expectations.

The API interface, on the other hand, becomes the visible *meta code* that really matters. Its structure has direct impact on API complexity, maintainability (i.e., providing backwards-compatibility) and easiness to learn and use. Unfortunately, there are yet no established metrics that would allow to measure and understand the APIs based on their exposed interface. Our work is aimed to facilitate progress in this direction.

In the following sections, we describe a set of relatively simple metrics that could be used as a starting point in API analysis. While it is still too early to say how beneficial these metrics could be in practice, one can already get a sense

of their potential applicability to different situations. The examples provided in the paper come from our analysis of some of the popular open web APIs[2].

1.1 Web APIs

The term *web API* can be somewhat confusing to the old-school programmers who are used to the original notion of the API (i.e., as a library of functions, methods, and/or classes written for a specific programming language). In the context of the API Economy this term becomes more of a *meta* concept. The part that stays the same is that the API is still a collection of functionalities. However, it is no longer tied to a specific programming language and its purpose is to expose business assets as opposed to facilitation of reuse of the existing code. Here are the definitions that we use:

Definition 1 (API Economy). *An economy in which companies expose their (internal) business assets or services in the form of (web) APIs to third parties with the goal of unlocking additional business value.*

Definition 2 (Web API). *A software interface that exposes certain business assets to other parties (systems and/or people) over the web.*

Definition 3 (API Call). *An HTTP request to the web API and a corresponding response.*

Web APIs can generally be executed using a web browser and their output typically is in a human-readable form. In particular, REST [6] and REST-like[3] APIs that use XML or JSON format are dominating the market (Table 1).

Table 1. Most used protocols and response formats[4]

Format	APIs	%	Protocol	APIs	%
XML or JSON	8882	81.08%	REST-like	7771	70.94%
XML	6047	55.20%	SOAP	2135	19.49%
JSON	5176	47.25%	not specified	1086	9.91%
XML and JSON	2341	21.37%	JavaScript	594	5.42%
not specified	1843	16.82%			
Total APIs: 10955					

[2] Based on the list of popular APIs from *programmableweb.com* [4], `http://www.programmableweb.com/apis/directory/1?sort=mashups`

[3] Following loose adherence to REST.

[4] Based on data extracted from *programmableweb.com* [4]

1.2 JSON and XML Based Responses

JSON is a data format that has been gaining a lot of popularity in recent years. One of the main reasons for its quick spread is that it is very easy to transform data from an XML structure to a JSON one and visa-versa. As a result, a number of web APIs support both output formats (Table 1).

JSON and XML enable representation of hierarchical semi-structured data. As such, every response to a request could be represented as a tree. An API as whole could then be seen as a tree composed of all call trees. This is the basis of our approach to the analysis of the API structure.

1.3 Our Approach

Our current strategy is to focus solely on the response part of API calls. The structure of call requests is generally much simpler and shorter and, therefore, has less impact on the resulting API. That said, however, we do have plans for incorporating request analysis in the future.

We also consider only the structure of the response and not the actual values contained within. Analyzing actual data could have its uses, in particular when it comes to testing, but again, we felt that it was important to focus on the structural aspects of the APIs first. This has a side benefit, as it means that data does not need to be present during the analysis, eliminating information security and privacy concerns.

The results of our ongoing research can be followed and freely accessed at our website [7]. We provide an open web API implementing analysis and metric computation [8,9], as well as a web GUI interface that is built on top of that API. We encourage any interested readers to visit our site and contribute with feedback, ideas, and suggestions.

2 Measurement Framework

Our analysis of API structure is based on three directions: (1) computation of tree-based metrics, (2) computation of node-name-based (semantic) metrics, and (3) visualization of APIs and API calls.

2.1 Objectives

It is difficult to predict all possible uses for the metrics in advance. Nevertheless, we have several general objectives that serve as our guideline:

- Evaluate how difficult it is to learn and/or use the given API.
- Understand what the problems in the current design of the API are.
- Evaluate API consistency and backwards-compatibility.
- Understand how one API differs from another and what the impact of those differences is.

2.2 Metrics

As described in the previous section, at the moment we use only the structure of call responses as our analysis target. As such, when we analyze an API we include only those calls that have output in the form of XML or JSON. We also consider only the typical responses to calls, disregarding special cases, like errors and exceptions. The latter typically have a very different structure from normal responses and their inclusion into analysis is left for future research.

We do consider different call responses based on variations in request parameters, but only when there is a structural difference present (e.g., when different request parameters result in different type of data returned).

It is common for responses to contain sets of elements of the same type. The same request might result in a different number of items returned depending on the context (e.g., the user who makes the request, current amount of data, etc.). From the structural point of view we do not care how many items are returned as long as they all are of the same type. For this reason, every request is first *sanitized* by merging sibling nodes that have exactly the same structure (see Figures 2 and 3). This means that no matter the amount of actual data returned, the resulting response representation stays the same.

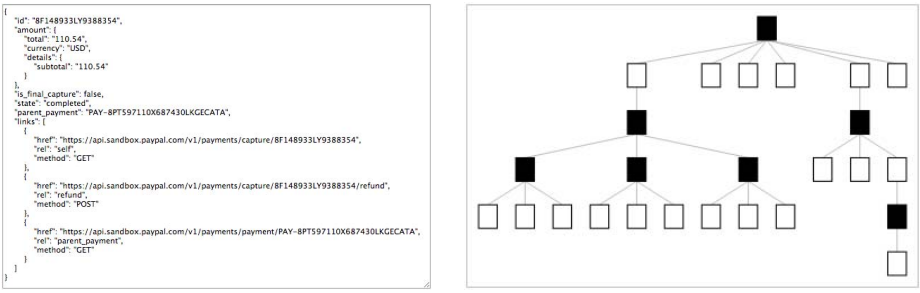


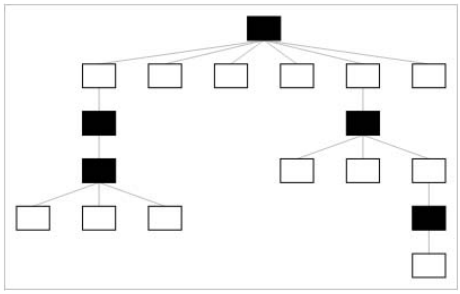
Fig. 2. JSON response and its visualization as a tree (black rectangles represent *meta* nodes). API: PayPal, API call: GET
`https://api.sandbox.paypal.com/v1/payments/capture/8F148933LY9388354`

2.3 Basic Tree Metrics

Every tree can be characterized by some simple metrics that reflect on its dimensions and/or shape.

One simple way to look at the size of a tree is to count the *number-of-nodes* (structural elements) it contains.

A related metric is the *number-of-unique-nodes*. *Unique* in this context means having a different node name from those that were counted before. This measure shows the *richness* of structure in terms of node variety. Used in conjunction with normal node count (e.g., as a ratio) it can be employed to identify the



Tree Info:

- Unique Nodes: 15
- Total Nodes: 18
- Tree Depth: 6
- Tree Breadth: 6
- Avg Children per Node: 0.944
- Max Children per Node: 6
- API Similarity Score: 0.2966
- Avg Call Similarity: 0.3636

Fig. 3. Visualization of the call tree after merging matching subtrees (left), displayed with some of the call metrics (right). API: PayPal, API call: GET `https://api.sandbox.paypal.com/v1/payments/capture/8F148933LY9388354`

presence of dominant node names, which might be an indication of duplicated values or overloaded naming (e.g., semantically different data parts given the same names).

Another way of looking at the size of a tree is to consider its node topology. We use three metrics for that purpose: *tree-depth*, *tree-breadth*, and *number-of-children-per-node*.

Tree-depth is the length of the path from the root of the tree to its furthest leaf.

Tree-breadth is the maximum number of nodes on one level of the tree. The level is the distance of a node from the root (i.e., the root node is at level 0, its child nodes at level 1, and so on).

When depicted visually, depth represents the height of the tree shape and breadth - the width.

The last metric we use is the *number-of-children-per-node*. It is computed as an average, though minimum and maximum values per tree might also offer valuable insights. This metric shows how much nodes tend to branch.

2.4 API Metrics

We employ two ways of computing API-wide metrics. The first one is based on the aggregation of metrics computed for each API call. This usually means computing average, minimum, and maximum values.

The second approach is to combine all call trees into one big API tree (see Figure 4 for an example of an API tree visualization) and compute the metrics defined in the previous subsection. This is performed in two steps:

1. We introduce a virtual root node and add all call trees as child nodes.
2. We merge sibling nodes that have the same structure. This way two or more calls might get combined into one subtree.

The merging step has two parameters that can be adjusted based on the desired effect. The first one determines whether to do the merge when a single

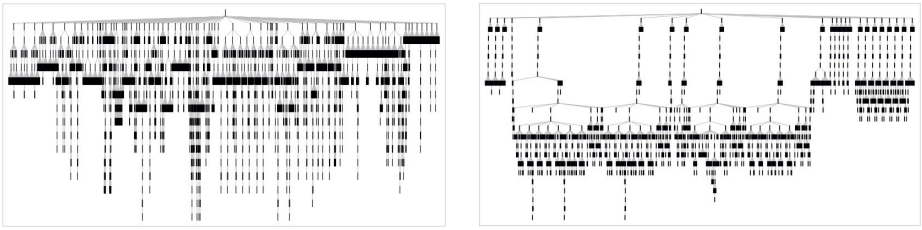


Fig. 4. Visualization of the API trees: left - *Twitter* API, right - *Bing Maps* API

element is compared to a list of elements of the same type as that element. Our default strategy is to execute the merge. This is because a single node can be seen as a list containing just one element (that node). In the situations when the multiplicity of nodes (i.e., one vs. many relationship) is considered to be important, such nodes can be left separate.

The second parameter is the merge factor. By default, we do the merge only when the structure of siblings matches exactly. However, this criteria can be relaxed using the *similarity* measure (described later in the paper). The motivation for merging similar but not exactly the same subtrees comes from the fact that some APIs have optional fields that are only returned if the appropriate values are present or if a specific parameter was added to the request.

2.5 Node-Name-Based (Semantic) Analysis

A different approach to analyzing API structure is to focus on the actual names of the nodes. These names are normally not given at random and represent semantics of the structure.

The simplest set of metrics of this type is the *count-of-each-named-node*. It can be computed for individual calls and for the whole API. The most frequent and the least frequent names are usually of most interest. Outlier values, in terms of name frequency, might indicate bad design or names that require special attention in the help manual or in the API reference.

A more advanced form of analysis is to consider which node names go together and how often. For example, one can count the occurrences of different *node pairings* when they appear as siblings (nodes that have the same parent node).

2.6 Similarity-Based Metrics

A known problem in the analysis of tree-based data is how to compare two trees. A common way of computing the similarity between trees is to use the so-called *edit-based distance* [10]. It can be expressed as the minimum number of edit operations required in order to convert one tree to the other.

Such *similarity* measure can also be used for comparing API responses, but it has one major disadvantage, besides being complex and computationally intensive. It considers the number of required edits but does not consider their

location (does not account for the significance of the hierarchy). In the case of the API response structure, we feel that the differences on the higher level of the tree (closer to the root) should have more impact. At the same time, we do not need a measure with the perfect precision. It is enough that we can tell exactly when two trees are the same, and to express their similarity in a somewhat intuitive manner in other cases. For this purpose we have devised the following *similarity* metric.

We consider that the *similarity* of two trees can be described by *similarities* among their subtrees, where each subtree has the same impact, independently from its size. So, if there are 5 subtrees across both trees, the impact of each would be 20%. However, we also want to account for the names of the root nodes, otherwise two trees that have the same root names but totally different subtrees would have a *similarity* of 0. There are different possibilities here in terms of how much weight we could give to the matching of such roots. It all depends on the point of view. It could be a fixed ratio, like 33% or 50%, but that would set the fixed minimum value for trees with matching roots. Therefore we feel that a weight related to the impact of one subtree is better fitting. At this point we simply use the weight of 1.0 - the same impact as that of a single subtree. When the root names do not match the trees are considered not similar and get the score of 0. The exception to this rule are the *unnamed* nodes from JSON - elements and lists (`{}` and `[]`) - they are counted as matching but are not considered to impact similarity by themselves (impact weight of 0).

The detailed steps to compute the *similarity* are shown below:

1. Check if the names of the roots match or if both are *meta* nodes. If not, the similarity is 0. Otherwise proceed to the analysis of subtrees (step 2).
2. Take the tree that has fewer subtrees (T_S; L_{T_S} - the number of subtrees of T_S). For each subtree i (i = 1, ..., L_{T_S}) of T_S compute the similarity with subtrees of the other tree (T_L; L_{T_L} - the number of subtrees of T_L). Memorize the highest similarity S_i (take the first one if multiple *similarities* match; stop the checks if a *similarity* of 1.0 - an exact match - is found) and remove the corresponding subtree of the tree T_L from further computations, unless no similar subtrees are found (when maximum similarity S_i was 0). In the latter case just memorize the value (0) and proceed to the next subtree. Once all subtrees from T_S are processed, combine the memorized impacts (step 3).
3. If the names of the roots matched, compute the final *similarity* score thusly:

$$S = \left(1 + 2 \cdot \sum_{i=0}^{L_{T_\mathrm{S}}} S_i\right) \div \left(1 + L_{T_\mathrm{S}} + L_{T_\mathrm{L}}\right) \ . \tag{1}$$

Otherwise (in the case of *meta* nodes), the roots are considered to have no impact:

$$S = \left(2 \cdot \sum_{i=0}^{L_{T_\mathrm{S}}} S_i\right) \div \left(L_{T_\mathrm{S}} + L_{T_\mathrm{L}}\right) \ . \tag{2}$$

The resulting value of the *similarity* ranges from 0.0 (not similar) to 1.0 (exactly the same).

Besides using this metric directly to compare two API calls and as a criteria for merging sibling nodes (as described in the previous sections), we also use it to build the API *similarity matrix* (see Figure 5 for an example).

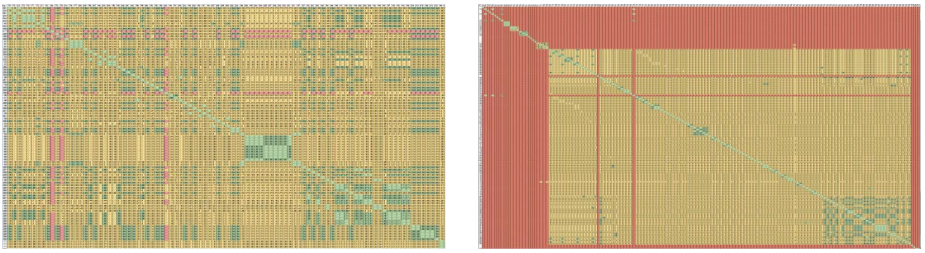


Fig. 5. Color-coded *similarity matrices* for the *Twillio* (left) and *Wikipedia* (right) APIs. Colors have the following mapping to the *similarity scores*: red - 0 (not at all similar), yellow - (0.0-0.5] (somewhat similar), green - (0.5, 1.0] (very similar)

API *similarity matrix* is the matrix containing *similarities* between every two calls of the API. Such matrix can be used to determine which calls could be grouped together (e.g., in the API manual, for better comprehension) or to verify if existing groupings are consistent.

Table 2. Examples of API *similarity scores* computed on different APIs

API	Number of Calls	Similarity Score
PayPal	19	0.2966
Twillio	94	0.2263
Twitter	78	0.1104
NASA	9	0.3723
Wikipedia	154	0.1726
BingMaps	24	0.9041
Salesforce	26	0.0644

We call the average of all similarities from the matrix, computed excluding self-similarities (i.e., a call compared to itself), the *similarity score* of the API. It represents the general consistency of the response structure within that API (see Table 2 for examples).

The *similarity* metric can also be used to analyze version evolution and the amount of structural changes (impact on backwards-compatibility) for an API call. It can also be applied for comparison of two different APIs (Figure 3).

Table 3. Examples of *similarity scores* computed between pairs of APIs

	PayPal	Twitter	NASA	Twillio	Wikipedia	Salesforce
PayPal	0.2966	0.0145	0	0.0003	0	0.0223
Twitter	0.0145	0.1104	0.0035	0.0033	0.004	0.0102

3 Conclusion

Although the metrics presented here have been tested on a set of open web APIs and appear to highlight certain differences in the call response structure, their practical value has yet to be properly evaluated.

Nevertheless, we hope that this paper will give ideas and motivation to other researchers and practitioners, and help in establishing a deeper body of knowledge regarding web APIs, their design, and their evolution throughout the API lifecycle.

References

1. Gat, I., Remencius, T., Sillitti, A., Succi, G., Vlasenko, J.: API Economy: Playing the Devil's Advocate. Cutter IT Journal 26(9), 6–11 (2013)
2. Gat, I., Succi, G.: A Survey of the API Economy. Cutter Consortium Agile Product & Project Management Executive Update 14(6) (2013), `http://www.cutter.com/content-and-analysis/resource-centers/agile-project-management/sample-our-research/apmu1306.html`
3. Clark, J., Clarke, C., De Panfilis, S., Granatella, G., Predonzani, P., Sillitti, A., Succi, G., Vernazza, T.: Selecting components in large cots repositories. Journal of Systems and Software 73(2), 323–331 (2004)
4. programmableweb.com, `http://www.programmableweb.com/`
5. Remencius, T., Succi, G.: Tailoring ITIL for the Management of APIs. Cutter IT Journal 26(9), 22–29 (2013)
6. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. dissertation, University of California, Irvine (2000)
7. apiwisdom.com, `http://www.apiwisdom.com/`
8. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A relational approach to software metrics. In: Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 1536–1540. ACM (2004)
9. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A non-invasive approach to product metrics collection. Journal of Systems Architecture 52(11), 668–675 (2006)
10. Bille, P.: A survey on tree edit distance and related problems. Theoretical Computer Science 337(1), 217–239 (2005)