Automated Generation of REST API Specification from Plain HTML Documentation

Hanyang Cao(✉), Jean-Rémy Falleri, and Xavier Blanc

University of Bordeaux, LaBRI, UMR 5800, 33400 Talence, France
{cao.hanyang,falleri,xblanc}@labri.fr

Abstract. REST is nowadays highly popular and widely adopted by Web services providers. However, most of the Web services providers only provide the documentation of their REST API in plain HTML pages, even if many specification formats exist such as WADL or OpenAPI for example. This prevents the Web Services users to benefit from all the advantages of having a machine-readable specification, such as generating client or server code, generating web services composition, checking formal properties, testing, etc. To face this issue, we provide a fully automated approach that builds a REST API specification from its corresponding plain HTML documentation. By given the root URL of the plain HTML API documentation, our approach automatically extracts the four mandatory parts that compose a specification: the base URL, the path templates, the HTTP verbs and the associated formal parameters. Our approach has been validated with topmost commercial REST based Web Services, and the validation shows that our approach achieves good precision and recall for popular Web Services.

Keywords: REST · APIs · Service description · Specification · OpenAPI

1 Introduction

REST, the architecture style defined by Fielding [5], is nowadays highly popular and widely adopted by most of the Web services providers. All the studies done by researchers [4] or by commercial sites such as ProgrammableWeb[1] state that more than 75% of Web services are now REST oriented.

However, Renzal et al. pinpoint that building REST services is still highly challenging [12]. They further highlight that the first REST best practice is to provide a rigorous specification of the REST API. Such a specification accelerates the development process by automatically generating client-side or server-side stubs [6], or even service composition [14]. Additionally, a rigorous specification can be used to reach a better quality by inferring parameters dependency constraints [16] or performing automating tests production [10] for example.

[1] https://www.programmableweb.com/api-research.

Several formats have been introduced for defining REST API specifications. One of them is the XML-based language WADL (Web Application Description Language), which is a *de jure* W3C standard [7]. Others, such as OpenAPI specification[2], RAML[3], and Blueprint[4] are JSON-based formats, and are *de facto* standards provided by the industry. However, even if many formats exist and if ones are more popular than the other (OpenAPI turns out to be the most popular one, with over 350,000 downloads per month), no format is widely adopted [11].

More precisely, as identified by Danielsen et al., most of the REST APIs providers only provide their documentation in plain HTML pages [4]. Further, according to an in-depth analysis of the most 20 popular REST Services [12], only 20% of them provide WSDL [2] specifications whereas 75% provide no rigorous specification and only plain HTML pages!

Such a situation then calls for an automatic transformation of plain HTML documentations into rigorous specifications. This will drastically help developers and make them benefit from all the advantages of having a rigorous specification: code and composition generation, test, type checking, etc.

In this paper, we face this problem and provide a fully automated approach that builds an OpenAPI specification from a corresponding plain HTML documentation. We choose OpenAPI because it is currently the most popular. Furthermore, once an OpenAPI specification exists, translating it into another format such as WADL for instance is very easy.

Our approach comes with a prototype implementation that inputs the root URL of the plain HTML API documentation, and that extracts the four mandatory parts that compose an OpenAPI specification: the base URL, the path templates, the HTTP verbs and the associated formal parameters. Our prototype has been validated with topmost commercial REST based Web Services as well as with Web Services selected at random into ProgrammableWeb. The validation shows that our approach achieves good precision and recall especially for popular Web Services.

As a main result, we provide:

– An automated approach that automatically generates an OpenAPI specification from the plain HTML documentation of an existing REST Web Service.
– A validation of our prototype and the OpenAPI specifications it yielded from topmost popular Web Services.

2 AutoREST: An Automatic Generator of REST API Specifications

This section first provides basic and simple definitions for the main concepts of REST API documentation and specification. It then presents an overview of our generator, called AutoREST, and finally presents its three main components.

[2] https://www.openapis.org/.
[3] http://raml.org/.
[4] https://apiblueprint.org/.

Definition 1 (REST API HTML Documentation). *A REST API HTML Documentation describes the resources provided by a REST service in plain HTML. It is composed of a set of web pages. Among the set of pages, one page is called the Root Page, and is linked directly or indirectly to all the pages of the set. Finally, all the pages belong to a same domain (the one of the Root Page) and each page may or may not contain useful information to access the service.*

As an example, the Root Page of the Instagram API HTML Documentation is https://www.instagram.com/developer/. From this Root Page, a set of 24 pages that belong to the same "www.instagram.com/developer" domain can be visited following the links between them. Finally some of these pages can be considered to be useful as they describe how to access the service. Other ones can be considered to be useless regarding this purpose as they don't describe how to access the service (e.g., service changelog information).

Definition 2 (REST API Specification). *A REST API Specification rigorously defines how to access the resources provided by a REST service. It is written in a de jure or de facto standard format such as WADL or OpenAPI. At least, it has to describe the following information:*

- *Base URL: The Base URL is the common prefix of all URLs that give access to the resources.*
- *Path Templates: The templates describes how the* Base URL *must be completed to make an URL that does give access to a resource. A template can include variables that are used to identify different but similar resources.*
- *Verbs: The verbs list, for each* Path Template*, the HTTP verbs that are supported by the Web service (GET, PUT, POST, etc.).*
- *Parameters: The parameters, for each couple of* Path Template *and* Verb*, define the list of formal parameters that are supported by the request.*

The objective of our approach (named AutoREST) is to automatically generate a REST API Specification from a REST API HTML Documentation. The Fig. 1 presents the global architecture of our approach. It shows that AutoREST inputs the Root Page of the REST API Documentation of a given Web service and then returns a generated OpenAPI Specification. More precisely, AutoREST performs the following three steps:

Step 1: Identifying all the HTML documentation pages. It gathers all the pages that are directly or indirectly linked by the Root Page and that belong to its domain. The purpose of this step is to identify all the web pages that may describe the REST API. We built a simple crawler that identifies all the web pages that are directly or indirectly linked by the Root Page of the REST API HTML Documentation. Furthermore, our crawler never goes outside of the domain of the Root Page.

Step 2: Classify useful or useless documentation pages. The goal of this step is to select only web pages that do contain useful information for building a REST API specification. As this step preforms a classification, we decided to use machine learning techniques [9].

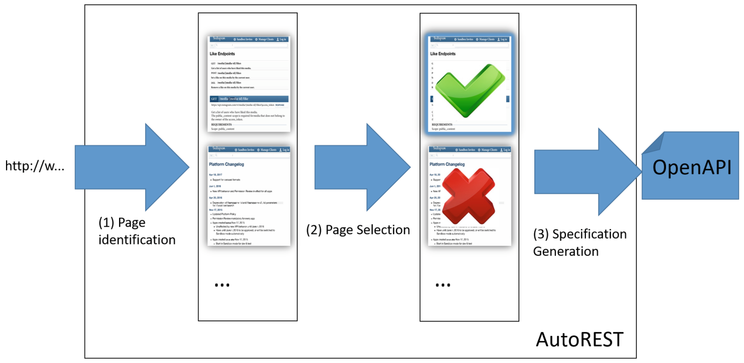

Fig. 1. Global process of our AutoREST

We therefore built a so-called *training set* that contains HTML pages that have been manually classified as being useful (Yes) or useless (No) regarding the purpose of generating a REST API specification. A page was said to be useful if it contains at least one information that can be used to generate a part of a REST API specification. We built that set by getting all the pages of the 15 topmost popular Web Services listed in ProgrammableWeb where popularity is expressed by the number of followers (see full list[5]). We chose to consider 15 Web Services because they gather 90% of all the followers.

Once the training set was built, we then extracted the features it contains. To that extent, each file of the training set has been treated as a plain text (one string) and transformed into a numerical feature vector by tokenizing it, counting tokens occurrences and normalizing tokens. For instance, the string "*Get a list of users who have liked this media ...*" is tokenized by using white spaces as token separators. Then, each token is assigned an integer id, such as {Get: 1, a: 2, list: 3}. Then the tokens are counted ad normalized by using the TF-IDF weighting to build the feature vector [15].

Finally, we computed and evaluated the classifier. We choose Random Forest [8] as the Machine-learning algorithm since it outperforms others on the supervised classification problem [9]. Regarding the size of the training set we tried various sets of different sizes. Result shows performance tends to be stable (96%) when size exceeds 200. Hence we chose to build a *training set* containing 200 HTML files. Our Classifier thus can select web pages that do contain useful information with a high precision (96%) and recall (96%).

Step 3: Extract Information and Generate REST API Specification. Since each Web service provider might have its own different patterns for displaying API documentation within HTML page, we made a simple comparative study on the same topmost 15 popular Web Services to better understand such patterns. The Table 1 lists the different patterns used by Web service providers

[5] https://github.com/caohanyang/REST_OPENAPI/blob/master/APIList.

Table 1. Patterns used by Web server providers to display REST API Specification in HTML pages.

Specification part	Patterns
`Base URL`	either in a dedicated part of the page or with each *Path Template*
`Path Template`	either with a partial URL starting with '/' or with a full URL including the *Base URL*
`Verbs`	Just before or after the path template
`Parameters`	In a list or in a array, just after the template

to display in an HTML page the four mandatory parts that compose a REST API specification.

Our component embeds different strategies (Regular Expressions, GATE configurations [3], etc.) that corresponds to the different patterns of displaying the informations within the HTML pages. As it cannot have any prior knowledge on how the information is displayed when it analyses a page, it then loop the analysis for each possible configuration and returns the Specification that contains the more *Path Templates*, *Verbs* and *Parameters*.

3 Evaluation

The objective of our evaluation is to measure the quality of the specifications generated by AutoREST. This quality can be measured according to the four mandatory parts of the specification (Base URL, Path Templates, Verbs and Parameters). Furthermore, it has to reflect what the documentation describes. More precisely, we measure the quality of a generated specification according to the following criteria. All criteria are measured manually by comparing the generated specification with its corresponding documentation:

– The quality of the *Base URL* is measured by a boolean. True means that the specification exactly reflects what is written in the documentation.
– The quality of the *Path Templates* is measured by counting the number of Paths templates in the specification and in the documentation, and by checking how much of them match. The quality is then expressed with precision (No. Match/No. in Spec.) and recall (No. Match/No. in Doc.).
– The quality of the *Verbs* is measured by counting the number of Verbs in the specification and in the documentation, and by checking how much of them match. Two verbs match if they have the same Path Template and if they are the same.
– The quality of the *Parameters* is measured by counting the number of Parameters in the specification and in the documentation, and by checking how much of them match. Two parameters match if they have the same Path Template, the same verb, the same name and the same type.

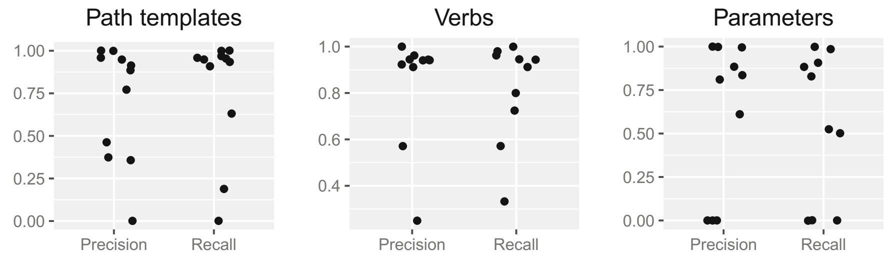

Fig. 2. Results of the topmost popular Web Services

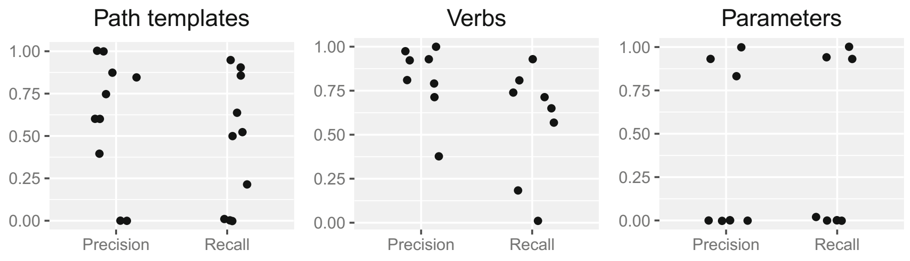

Fig. 3. Results of the random Web Services

Our AutoREST has been developed in Python and Java, and is available on-line as an Open Source Project[6]. We evaluate AutoREST on two sets of Web Services. The first set is composed of the 15 topmost popular Web Services. The second set is composed of 15 Web Services selected at random from ProgrammableWeb. The evaluation done with the first set expresses how AutoREST performs on popular Web Service knowing that it has been trained with a small subset of them for selecting interesting pages, and that its information retrieval rules have been defined by analyzing them (see Sect. 2). The evaluation done with the second set expresses the capacity of AutoREST to generated OpenAPI specifications without any prior knowledge.

As a main result, AutoREST has quite good results for finding the *Base URL*: 11/15 for the topmost popular Web Services, and 10/15 for random Web services. The Figs. 2 and 3 then present the precision and recall for the *Path Templates*, *Verbs* and *Parameters*. It should be noted that when AutoREST fails in finding the *Base URL*, it also fails for all of the other parts. As a consequence, we choose not to show these cases in the Figures.

As we just presented it, AutoREST is quite good for generating a *Base URL*. It fails when the *Base URL* is not documented neither in a dedicated place nor with the *Path Templates*. For example, it fails with Twilio that contains a variable in the base URL. More precisely the *Base URL*

[6] https://github.com/caohanyang/REST_OPENAPI.

of Twilio is "https://api.twilio.com/2010-04-01/Accounts/{AccountSid}" where AccountSid is used to authenticate the user.

For *Path templates* the results are good but more debatable. First of all, it is clear that AutoREST performs better for popular Web Services, as it has been trained on it. After the manually investigation, we found AutoREST fails mainly for two reasons. First it uses regular expression to detect URLs but as there are many URLs in web pages it sometimes fails to distinguish the ones that correspond to REST services. Second, it sometimes fails to infer the Path templates which contains path templating. Indeed, some API providers present the Path templates by providing examples. AutoREST then fails in extracting these generic cases. For instance, Twitter lists an example request https://api.twitter.com/1.1/geo/id/df51dec6f4ee2b2c.json in its documentation page. AutoREST then considers it as a Template Path!

For *verbs* the results are quite similar than *Path Templates*. AutoREST performs a little bit better for popular Web Service.

Finally, AutoREST is good to extract the Parameters for popular Web Services but not for the ones that have been randomly selected. The main reason is because the documentation provided by the latter is not structured with tables or lists, as it is expected by our information retrieval component.

4 Related Work

Only three existing works are related to the generation of REST API specifications.

In [13], Sohan et al. provide SpyREST, an approach for generating RESTful API documentation by using an HTTP proxy server. In contrast to our approach that is static, SpyREST is dynamic as it listens to the communications that are performed with the REST Services to generate the documentation. It then requires a client that knows how to call the REST Services and also requires the client to perform all the possible calls.

In [1], Alarcón et al. provides RESTler that crawls a RESTful Service and aims to generate a map that presents all the provided resources and their links. This approach then does not generate a rigorous specification.

5 Conclusion

In this paper we then present AutoREST, an approach for automatically transform an HTML documentation into an OpenAPI specification. It can then be used as a black box tool that only inputs one root URL and that generates an OpenAPI specification. The validation we done shows that AutoREST has quite good results especially with popular Web Services. For randomly selected Web Services it is less successful mainly because the provided HTML documentation is not structured as the one of the topmost popular Web Services.

As a further work, we plan to work on a component that validates the returned OpenAPI specification after its generation by generating and testing calls. Thanks to this component, we then aim at returning an OpenAPI specification that does not contain any faults (100% precision). We also plan to extend machine learning component. Our goal is to strengthen our approach to better identify weak HTML documentation, with the intent to provide error messages indicating that the OpenAPI generation cannot be performed.

References

1. Alarcón, R., Wilde, E.: Restler: crawling restful services. In: Proceedings of the 19th International Conference on World Wide Web, pp. 1051–1052. ACM (2010)
2. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation 26, 19 (2007)
3. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrell, G., Funk, A., Roberts, A., Damljanovic, D., et al.: Developing language processing components with gate version 6 (a user guide). University of Sheffield, UK (2013). http://gate.ac.uk/sale/tao/index.html
4. Danielsen, P.J., Jeffrey, A.: Validation and interactivity of web API documentation. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 523–530. IEEE (2013)
5. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. (TOIT) **2**(2), 115–150 (2002). http://dl.acm.org/citation.cfm?id=514185
6. Fokaefs, M., Stroulia, E.: Using WADL specifications to develop and maintain rest client applications. In: 2015 IEEE International Conference on Web Services (ICWS), pp. 81–88. IEEE (2015)
7. Hadley, M.J.: Web application description language (WADL) (2006)
8. Ho, T.K.: Random decision forests. In: Proceedings of the Third International Conference on Document Analysis and Recognition, vol. 1, pp. 278–282. IEEE (1995)
9. Koprinska, I., Poon, J., Clark, J., Chan, J.: Learning to classify e-mail. Inf. Sci. **177**(10), 2167–2187 (2007)
10. López, M., Ferreiro, H., Francisco, M.A., Castro, L.M.: Automatic generation of test models for web services using WSDL and OCL. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 483–490. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45005-1_37
11. Lucky, M.N., Cremaschi, M., Lodigiani, B., Menolascina, A., De Paoli, F.: Enriching API descriptions by adding API profiles through semantic annotation. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 780–794. Springer, Cham (2016). doi:10.1007/978-3-319-46295-0_55
12. Renzel, D., Schlebusch, P., Klamma, R.: Todays top restful services and why they are not restful. In: Web Information Systems Engineering, WISE 2012, pp. 354–367 (2012)
13. Sohan, S., Anslow, C., Maurer, F.: Spyrest: automated restful API documentation using an HTTP proxy server (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 271–276. IEEE (2015)

14. Wagner, F., Klöpper, B., Ishikawa, F., Honiden, S.: Towards robust service compositions in the context of functionally diverse services. In: Proceedings of the 21st International Conference on World Wide Web, pp. 969–978. ACM (2012)
15. Wu, H.C., Luk, R.W.P., Wong, K.F., Kwok, K.L.: Interpreting TF-IDF term weights as making relevance decisions. ACM Trans. Inf. Syst. **26**(3), 13:1–13:37 (2008). http://doi.acm.org/10.1145/1361684.1361686
16. Wu, Q., Wu, L., Liang, G., Wang, Q., Xie, T., Mei, H.: Inferring dependency constraints on parameters for web services. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1421–1432. ACM (2013)