Analysis of Tools for REST Contract Specification in Swagger/OpenAPI

Jéssica Soares Dos Santos[a], Leonardo Guerreiro Azevedo, Elton F. S. Soares, Raphael M. Thiago and Viviane T. Silva

IBM Research, Pasteur Ave, 146, Rio de Janeiro, Brazil

Keywords: REST Services, REST Contracts, REST APIs, Swagger, OpenAPI.

Abstract: REST is a resource-based architectural style that has emerged as a promising way for designing Web services. A REST API exposes services' functionalities through a contract that allows consumption by different clients. The contract specifies service's request and response schemes and related rules the service and the client should comply with. The process of documenting and keeping an API consistent is a time consuming human effort. The documentation should reflect the implementation which may evolve. This work compares different tools for REST APIs specifications. We focused on tools that automatically generate Swagger (Open API in version 3.0), a specification for designing REST APIs. We evaluated the tools using a set of criteria whose results may help software engineers to choose the most appropriate tool, and point out gaps for research initiatives.

1 INTRODUCTION

The design of an Application Programming Interface (API) creates its contract (the API specification) (Santos et al., 2019), *i.e.*, its functionalities, the communication protocol, input and output parameters, data format and endpoints (De, 2017).

The Representational State Transfer (REST) (Fielding and Taylor, 2000) is an architectural style designed to address the properties for a modern Web architecture. REST style defines a set of constraints/principles that services must follow in order to achieve *usability*, *simplicity*, *scalability*, and *extensibility* (Li and Chou, 2011).

The REST APIs have established a means for realizing distributed systems (Haupt et al., 2017); hence, it entails the importance of documenting REST APIs in a proper way (Li and Chou, 2011). Keeping an API documentation/contract aligned with the implementation is a challenging task due to services frequent update (De, 2017). It is still an open issue how to improve API design (Haupt et al., 2017), and one way towards this is the use of tools for API contract generation during API development (Santos et al., 2019).

There are two approaches for documenting APIs (Varga, 2016): (i) *bottom-up* or *contract-last*; (ii) *top–down* or *contract-first*. The former consists in generating the contract from the code, *i.e.*, first the code is implemented and then the contract is created/generated, *e.g.*, marking up the code with annotations and generating the API contract based on them. The latter creates the API documentation/contract before writing the service code. Several specifications have been proposed for specifying REST contracts, such as: Swagger (or OpenAPI in its third version)[1], WADL[2], WSDL2.0[3], and API Blueprint[4].

Swagger is the most popular language for specifying REST contracts (Tsouroplis et al., 2015), it provides development tools, and it can be used for *bottom-up* or *top-down* contract specification.

This work aims at exploring tools that automatically generate contract specification. Technically, it provides a description and a comparison of tools; help software designers to identify relevant characteristics to be considered when looking for a tool to document REST services. Academically, it presents a background for one (starting) working in the area; point out gaps for future researches.

The remainder of this work is organized as follows. Section 2 and Section 3 present background and related work. Section 4 describes the tools, and Section 5 compares them. Finally, Section 6 concludes and points out future work.

[a] https://orcid.org/0000-0001-5082-4583

[1] http://swagger.io
[2] https://www.w3.org/Submission/wadl/
[3] https://www.w3.org/TR/2007/REC-wsdl20-20070626
[4] http://raml.org

Santos, J., Azevedo, L., Soares, E., Thiago, R. and Silva, V.
Analysis of Tools for REST Contract Specification in Swagger/OpenAPI.
DOI: 10.5220/0009381202010208
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 2*, pages 201-208
ISBN: 978-989-758-423-7

Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 BACKGROUND

RESTful services must follow six main principles (Varanasi and Belida, 2015), (Fielding and Taylor, 2000): (i) Clear separation of client and server concerns; (ii) Stateless communication; (iii) Server responses must be declared if are cacheable; (iv) Services should have a uniform interface; (v) Intermediate layers of software or hardware can be introduced between client and server; (vi) Client may download code from server and execute it.

2.1 REST API

An API provides a set of functionalities, their descriptions and dictates the rules for using them. APIs are responsible for opening a system/set of services for a broader audience and can make integration easier (Preibisch, 2018).

A process for service implementation (and contract design) may be composed by the following steps (Josuttis, 2007) (Figure 1[5]):

1. Create a requirement sketch: *e.g.*, in pseudocode, sequence diagram or use case description aiming at understanding the business needs;
2. Create a specification of the service: *e.g.*, an UML design model or a Swagger specification;
3. Search for potential services, *i.e.*, existing services that may fit the requirement, which may result in:
   (a) A service that fits the new requirement, which can be used directly, and the process ends;
   (b) A service that partially fits the new requirement, which may be refactored - service clients may be impacted;
   (c) No service that matches the requirements; then, a new service should be developed.
4. Search for existing type definitions to be used as input/output parameters of the service. Types may be created or changed.

Using a tool for contract specification may help depending on the approach the tool supports (*i.e.*, *contract-first* or *contract-last*). In the former, the service contract may be created in Step 2 and used in the search for potential service (Step 3) and the search for types (Step 4). In the latter, a tool for contract generation creates it after the implementation is done; hence, it does not help much the presented process.

A REST API should describe (Cao et al., 2017): a base URL corresponding to a common prefix of all other URLs that give access to the resources; a path template describing how the base URL should be completed to provide access to a given resource; supported actions, *i.e.*, HTTP methods; parameters needed to each pair path template-method.

[5]The model was designed in a UML activity diagram (Rumbaugh et al., 2004) model.

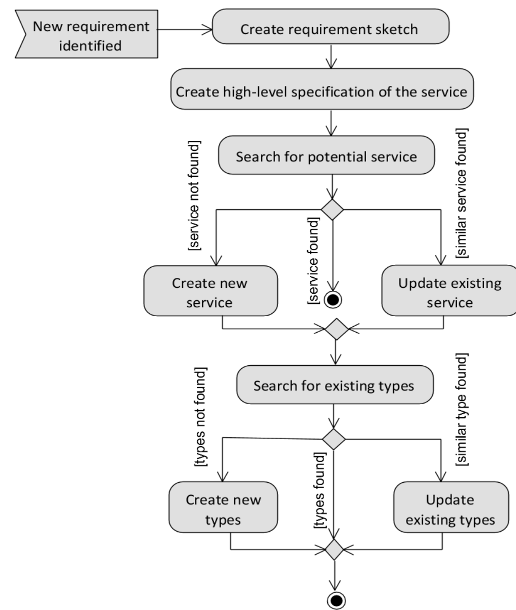

Figure 1: A Process for Service Implementation.

2.2 Swagger 2.0/OpenAPI

Swagger is a framework for documenting REST APIs[6]. It has a large community of users and supporters, and it can be written in JSON[7] or in YAML[8].

Swagger main elements are (OpenAPI initiative, 2018): (i) Swagger version; (ii) API title, version and description; (iii) API server's address; (iv) Paths and available HTTP methods (POST, GET, PUT, DELETE); (v) The operations' request parameters and responses; (vi) Possible return HTTP status code; (vii) Type definitions (also called API models). In December 2017, Swagger version was updated to 3.0 and renamed to OpenAPI Specification. Beyond structural changes, the main differences between Swagger 2.0 and 3.0 are: (i) Version property changed to *openapi*; (ii) New keywords for schema specification/validation, such as: *oneOf*, *anyOf*, *not*, *allOf*; (iii) Each endpoint may have several addresses; (iv) Specification of links between resources following HATEOAS; (v) Specification of examples.

[6]An example is presented at https://petstore.swagger.io/v2/swagger.json.
[7]JavaScript Object Notation (https://www.json.org)
[8]YAML Ain't Markup Language (http://yaml.org)

3 RELATED WORK

The majority of work compares existing specification languages for documenting service contracts or presents guidelines/recommendations to be followed when developing service contracts. Our work differs from them since we analyze *tools* for generation and maintenance of REST contracts written in Swagger.

Wideberg (Wideberg, 2015) presented a comparison of different specification formats for REST contracts, such as: Swagger, RAML, API Blueprint and WADL. The work concluded that Swagger is the specification language that most suits the desirable requirements in enterprise environments and the specification language that has the most active community, which is a good indicator of future support and continuous development. Similarly, Surwase (Surwase, 2016) analyzed the same specification languages from a developer's perspective, and pointed out that Swagger and RAML support many programming languages adopted in the industry. Among the dimensions used for comparing the specification languages, it was considered the output format that can be generated (YAML, Markdown and JSON) by the specifications and the companies that are the sponsors of these languages.

Pritchard *et al.* (Pritchard et al., 2017) compares the same languages that were considered by (Wideberg, 2015) and (Surwase, 2016), but including RESTCONF (Jethanandani, 2017) and WSDL 2.0 languages. They remark the objectives of each language, compared their output format, and highlighted their drawbacks and advantages. As Swagger advantages, they pointed out, *e.g.*,: vast development documentation; good community support; the possibility to be adopted by existing APIs; and, to be written independently of specific programming language. The lack of API testing interaction is its main drawback.

Robillard (Robillard, 2009) investigated challenges that software developers can face when learning new APIs. He pointed out a set of guidelines that API documentations must follow in order to be easy to learn and use. These guidelines include: be complete; present good examples; provide support to complex usage scenarios; and, be conveniently organized. In the same sense, Espinha *et al.* (Espinha et al., 2014), from interviews with developers, produced a set of recommendations API designers should follow in order to reduce the impact API clients face when the API evolves, such as: provide examples describing the usage of the new version functionalities; and, adopt and execute blackout tests, which consists in deactivating older versions of the API for short periods of time before deactivating it permanently.

4 TOOLS FOR SWAGGER GENERATION

Many tools have been proposed as alternatives to facilitate the Swagger generation and maintenance. In this section, we present a brief description of them. Our goal is not to present and extensive list of all existing tools, but rather tools that represent different perspectives of development, such as tools that generate code (and Swagger) from models, tools that allow generation of Swagger from code through annotations, tools that work apart from code and model etc. The tools were selected based on the following characteristics: Availability for download; Adoption by the industry; Provide documentation or usage examples that allows to understand their features.

4.1 Tools Provided by Swagger Project

The Swagger project provides three official open-source tools:

1. **Swagger Editor:** an API Editor for designing Swagger contracts. It can be downloaded and used locally or can be accessed online[9];
2. **Swagger UI:** It automatically generates an HTML page from a Swagger specification, providing an interactive interface to visualize and test Swagger specifications that can be accessed by the Web Browser;
3. **Swagger Codegen:** it generates client and server code from Swagger, *e.g.*, in Java, Python and PHP.

4.2 Visual Paradigm

Visual Paradigm[10] (VP) is a software engineering tool for modeling/designing software. This tool focuses on the Model Driven Development (MDD) (Pastor et al., 2008) approach, which consists in generating the software from the software model that specifies how it should work instead of starting from writing code. By using VP, it is possible to design a REST API by designing class diagrams that describe *REST resources*. The VP UML component called *REST resource* can be associated with a URI (resource identifier), request and response bodies (with HTTP status), path parameters, and a method that specifies the resource action.

The process of designing/documenting a RESTful API is totally graphical. Given a UML class diagram that describes REST service definitions, the VP API

[9] http://editor.swagger.io/

[10] https://www.visual-paradigm.com/

designer is able to automatically generate API definitions in accordance with Swagger 2.0. Default UML relationships can be defined between UML classes, such as, inheritance, association, multiplicity. Additionally, it is possible to generate from the diagrams: (i) the server code, which is the HTML code that shows how the API should be consumed; and, (ii) the client and servlet code, which corresponds to the code for a client to consume an API service and the code that a service provider should have to be able to process a client request, respectively. Code can be generated in programming languages like Java, Android, Obj-C, JavaScript, C#, PHP, Perl, Python. Visual REST API designer is available in the Enterprise and Professional editions of Visual Paradigm.

4.3 IBM Rational Software Architect Designer

IBM Rational Software Architect[11] (RSA) Designer is a MDD software engineering tool for the development and design of applications and Web services.

RSA supports the generation of Swagger from UML diagrams and provides also some kinds of transformations that convert Swagger specifications into REST models (*i.e.*, reverse engineering). In this way, it is possible to load and visualize existing Swagger files in the form of UML diagrams.

RSA allows conversion of REST models into code representing the skeleton of the service code, *e.g.*, in Java, NodeJS, C#. It supports only Swagger 2.0.

4.4 Typson

The Typson[12] library converts Typescript definitions to JSON-schemas which can be integrated with Swagger definitions and referenced as input or output parameter types. It supports types, *e.g.*, required properties, inheritance , and enumerations.

When Typson converts the Typescript definitions to JSON-schemas, it discards the explicit inheritance information, *i.e.*, when a Typescript interface *A* extends an interface *B*, the attributes of *B* are copied to *A* and the relationship between *A* and *B* is not maintained. The advantages of using Typson are: it provides the reuse of interfaces by different methods and by other interfaces (when one interface extends another one); the set of interfaces does not need to be organized inside the same file; and, the API documentation is coupled with a Typescript code but the service code is not coupled with any specific programming language. It also may help the integration with UI (User Interface) programming when both UI and services use the same type. Typson supports Swagger 2.0 and is available as a Node.js module.

4.5 Flask-RESTPlus

Flask RESTPlus[13] is an extension of Flask, a framework for developing Python applications. The REST documentation is generated based on a set of predefined decorators that are used to relate a method to an endpoint, input and output parameters. By default, Flask-RESTPlus generates automatically Swagger UI documentation. Method responses are specified with the decorator @*api.response*. Flask-RESTPlus has a builtin mechanism to validate request data. By importing *reqparse*, one can associate a parameter with a primitive type (for instance, type *int*). The *reqparse* provides a *RequestParser* that is able to show error messages by itself. Flask-RESTPlus also allows the definition of API models with the *api.model*, which represents type definitions, *i.e.*, objects with a set of attributes that can be passed as the payload of the method (input or output parameters). The types, descriptions, and other information (*e.g.*, attribute required or not) about attributes of an object can be specified when importing *fields* from `flask_restplus`. Inheritance can be specified between API models of the Flask-RESTPlus and API models can be nested inside existing models. Additionally, models can be extended by using the `api.inherit()` method.

4.6 Flasgger

Flasgger[14,15] is a Flask extension that is able to extract Swagger from Flask code. Flasgger provides an interactive API documentation. Swagger is generated based on decorators and docstrings that should be put inside the service code. The decorators are used to define the routes @*app.route*, which inform the path of the service endpoints and the HTTP methods that represents the operations. The docstrings are used to define input/output parameters, type definitions and responses. Flasgger allows specifications be defined in an external Swagger file and related to the code using an specific decorator called (*swag_from*) or a docstring *file* shortcut. Flasgger is totally compatible with the generation of Swagger 2.0 and previous versions.

[11] https://www.ibm.com/us-en/marketplace/rational-software-architect-designer

[12] https://github.com/lbovet/typson

[13] https://github.com/noirbizarre/flask-restplus

[14] https://github.com/rochacbruno/flasgger

[15] http://flasgger.pythonanywhere.com

Documentations states OpenAPI is also supported but we did not find examples showing this.

4.7 Flask-Restful-Swagger

The Flask-Restful-Swagger[16] supports Swagger by wrapping the API instance and including decorators in a Python code. It provides a swagger generator and a graphical interface of the services that are exposed. The decorator @*swagger.operation* is responsible for defining an operation with a set of parameters and response (*e.g.*, datatype, multiplicity, name, description). Methods that are not decorated with @*swagger.operation* are not added to Swagger documentation as operations. Classes can be decorated with @*swagger.model* in order to represent a Swagger type definition. Instead of using the decorator @*swagger.model*, it is also possible to use the decorator @*marshal_with* passing the class to define a Swagger type definition. Additionally, metadata can be passed to the Swagger by creating a *swagger.docs* file containing the description of the endpoints, api version, base path and so on.

4.8 Spring MVC REST API

Spring[17] is a Java framework that can be used to build Web applications following the Model-View-Controller (MVC) architectural pattern (Sharan, 2015). Spring MVC provides many advantages, such as dependency injection model[18] and modules to facilitate data access.

The REST API support was introduced in Spring 3.0. The Spring MVC REST API[19] is able to generate a Swagger specification based on a set of predefined annotations that should be included in the Java code. First of all, a Swagger configuration should be defined for the Java project in a XML file. REST endpoints are defined by annotating classes with the @*Api* annotation. The annotators are: (i) @*Api*: indicates a class as a Swagger class; (ii) @*ApiOperation*: indicates the responsibility of the method; (iii) @*ApiParam*: corresponds to the parameter that the method is expecting and can be associated with its name, value, description and a flag; (iv) @*ApiResponse*: represents the response; and, (v) @*ApiModel*: used to define a Swagger type definition. In order to accelerate the code documentation, there are tools that generate documentation from code. The Springfox[20] has this purpose. It is a suite of Java libraries that generates specifications for JSON APIs written using the Spring family of projects. Springfox examines an application at runtime and infers semantics based on Spring configurations, class structures and compile time Java Annotations. It generates specifications such as Swagger, RAML and jsonapi. It has the goal to discourage using (swagger-core) annotations that are not used at runtime. It provides options to configure general characteristics, and one has to use swagger-core annotations when a specific documentation is needed. Hence, it does not replace the annotation use although it speeds up the documentation process. Therefore, we evaluated, in this work, the Spring MVC which is more general.

[16]https://github.com/andyzt/flask-restful-swagger

[17]https://spring.io/

[18]http://martinfowler.com/articles/injection.html

[19]https://spring.io/projects/spring-restdocs

[20]http://springfox.github.io/springfox

4.9 Swagger PHP

The Swagger PHP tool[21] scans a PHP project and merge all annotations in order to generate a Swagger file. A Swagger POST operation is defined by including the @*SWG\Post* annotation. Similarly, there are equivalent annotations for other HTTP methods (GET, PUT, DELETE). Inside an annotation of an operation, it is possible to inform the path, the id of the operation and the operation description. Additionally, Swagger PHP provides an annotation called @*SWG\Info* to define the description and version of the API. @*SWG\Parameter* should be nested inside an annotation of operation and indicates the parameter type, description and if it is required or not. The annotation @*SWG\Response* represents the operation response and defines the status, the description and the type. Responses also should be nested inside an annotation of operation. Type definitions can be specified using the annotation @*SWG\Definition*. Definitions can be referenced with the annotation @*SWG\Schema*. In this way, these definitions can be referenced and reused in different parts of the code. The keyword *allOf* can be put inside a @*SWG\Definition* in order to create a combination of definitions. The documentation of Swagger PHP states that it is able to generate Swagger in the most recent version, *i.e.*, Swagger (Open API) 3.0 and already can deal with *anyOf* and *oneOf* keywords. However, there is a lack of examples involving annotations that generate Swagger 3.0. The majority of annotations to generate Swagger 3.0 are similar to the annotations for generating Swagger 2.0, but the prefix @*SWG* is replaced by the prefix @*OA*.

[21]https://github.com/zircote/swagger-php

5 COMPARATIVE ANALYSIS

This section presents a comparative analysis of the tools. The evaluation criteria are presented in Table 1. They were defined according to what a tool should support to create contracts with the formalism required by service development initiatives as proposed by OpenAPI project (OpenAPI initiative, 2018). Besides, we added criteria MDD support and the approach (contract-first or a contract-last).

Table 1: Comparison criteria.

	Description
1	Ability to generate Swagger automatically
2	Ability to generate client or server code
3	Ability to generate UI documentation
4	Model Driven Development approach
5	Support to inheritance, reuse and validation
6	Low coupling with service code
7	Ability to check Swagger version syntax
8	Support to OpenAPI (Swagger version 3)
9	Contract-first or contract-last approach

Table 2 presents the evaluation. Criteria 7, 8 and 9 are not mentioned in Table 2 due to space limitations for the table presentation. However, a discussion about all criteria is presented as follows.

Criterion 1: Swagger codegen and Swagger UI are not able to generate Swagger specification file. Although the Swagger editor tool also does not present means to automatize Swagger generation, it can facilitate the manual creation of the Swagger specification file because it presents Swagger examples and can receive as input a Swagger specification and inform if its syntax is correct according to the Swagger version, highlighting errors. Typson tool is able to generate API models (definitions) using Typescript, but the information about the endpoints (*e.g.*, input and output parameters) has to be provided manually inside the Swagger file (in JSON or YAML). In Table 2, the sixth column (*Based on*) indicates the way that the tools whose column *Generate Swagger* is (✓) use to address Criterion 1.

Criterion 2: VP and Swagger codegen present the ability to generate server and client code, the IBM RSA can only generate server code. In Table 2, the column *Based on* indicates the way the tool which column *Generate Client Code* or *Generate Server Code* marked with (✓) use to address Criterion 2.

Criterion 3: An UI allows users to visualize the resources/operations that are available, the expected input parameters and responses, and try out the invocation of different operations on the resources. VP, Flask-RESTPlus, Flask-Restful-Swagger, Flasgger and Swagger UI can generate an HTML page based code exposing the interface of the services. Approaches that do not provide the generation of a specific UI Documentation can be combined with the Swagger UI official tool as is the case of, *e.g.*, Typson, Swagger PHP and Spring MVC REST API. In Table 2, the column *Based on* indicates the way that the tool which column *Generate UI Documentation* is marked with (✓) use to address Criterion 3.

Criterion 4: VP and IBM RSA are based on the MDD approach, providing mechanisms to generate Swagger from diagrams, and a graphical way to edit and visualize the Swagger specifications and Swagger type definitions (API models). In this approach, it is not required that service contract designers have programming abilities. Additionally, IBM RSA is the only one that can directly load existing Swagger specifications into UML diagrams – reverse engineering.

Criterion 5: *Swagger definitions* are objects that can be referenced as input and output parameters of operations. In Table 2, the column *API Model Inheritance* shows that, by using Flasgger and Swagger Editor, it is possible to associate a definition using a keyword (*e.g.*, *allOf*). In VP and IBM RSA the inheritance is represented using inheritance UML relationships. The column *API model Reuse* corresponds to the ability to reference an API model inside another one or in different operations. The column *Validate Input Data* exhibits whether the approach provide own mechanisms for validating input data against Swagger specification. *E.g.*, Flask-RESTPlus provides *reqparse* library for that.

Criterion 6: Flask-RESTPlus, Flask-Restful-Swagger, Spring MVC Rest API and Swagger PHP, generates specifications based on annotations/decorators provided inside the service code. Flasgger generates the specification based on docstrings inside the code. Although annotations/docstrings can be defined before the complete implementation of the service, they need to be put inside the service code. Therefore, VP, IBM RSA and Typson are the only tools that automatize the Swagger generation and Swagger Editor allows the edition of the contract decoupled form the code. They are completely independent of the technology that will be used to implement the services, *i.e.*, they can be adopted by software groups that desire to implement services in different programming languages.

Criterion 7: Swagger editor is the only tool that is able to verify whether a Swagger specification is in accordance with the syntax of a given Swagger version. This feature can be very helpful to the ones that want to edit parts of the REST APIs manually.

Criterion 8: Only the official tools (Swagger code-

Table 2: Tools Analysis Summary.

	Criterion 1	Criterion 2	Criterion 3			Criterion 4		Criterion 5			Criterion 6
Tool	Generate Swagger	Generate Client Code	Generate Server Code	Generate UI Documentation	Based on	Generate Swagger from Model	Generate Model from Swagger	API Model Inheritance	API Model Reuse	Validate Input Data	Low Coupling with Service Code
VP	✓	✓	✓	✓	UML Diagram	✓		UML Relationship	✓		✓
IBM RSA	✓		✓		UML Diagram	✓	✓	UML Relationship	✓		✓
Flask-RESTPlus	✓			✓	Decorators			Method	✓	✓	
Flask-RESTful-Swagger	✓			✓	Decorators			Decorator	✓		
Flasgger	✓			✓	Decorators Docstring			Swagger keyword	✓	✓	
Swagger editor								Swagger keyword			✓
Swagger codegen		✓	✓		Swagger specification						
Swagger UI				✓	Swagger specification				✓		
Spring MVC REST API	✓				Annotations			Annotation	✓	✓	
Typson	✓				Typescript definitions			Typescript keyword	✓		✓
Swagger PHP	✓			✓	Annotations			Annotation	✓		

gen, Swagger UI and Swagger Editor), Flasgger and the Swagger PHP support the current version of Swagger 3.0 (OpenAPI). However, it is not clear in the documentation of Flasgger and Swagger PHP if they support all new features of the Swagger 3.0.

Criterion 9: All tools based on annotating the code follow contract-last approach. Swagger Editor is a tool for creating the contract; therefore, it may be used on a contract-first approach. MDD tools (like VP and IBM RSA) generates code from model; hence, they are contract-first tools since the contract is defined by the model. These tools support better the process for service contract design (Figure 1).

6 CONCLUSIONS

The REST architectural style has become a popular approach for development in Service-Oriented Architecture (SOA) and in Microservice Architecture (MSA), in which applications are organized into several independent services (Newman, 2015). REST style is centered in the idea of transferring data (*resources*) through the use of HTTP methods in a stateless way, and defines a set of desirable principles/ good practices to be followed when building modern Web Services. Systems based on REST style can expose their services to be consumed via REST APIs.

Several specification languages have been proposed to document REST services. In this research, we focused on Swagger since it is: the most popular framework for documenting REST APIs (Tsouroplis et al., 2015); the language that most suits the desirable requirements in enterprise environments (Wideberg, 2015); and, highly adopted in the industry (Scherer, 2016). Many tools have been proposed to facilitate the creation and maintenance of Swagger specifications.

In this work, we surveyed the literature in order to find tools that generate Swagger specification, and analyzed them considering their adherence to a set of criteria the Swagger project presents as the advantages of using such specification and the best fit to a process for contract development.

The results of our comparative analysis are useful to understand the features provided by each tool and can help one to combine different tools. As an example, Flasgger and Swagger UI can be combined to obtain the UI documentation together with the Swagger specification file. Additionally, existing Swagger specification files generated by tools such as, Flasgger or Typson, can be loaded into the IBM RSA tool in order to achieve the graphical management of REST contracts, what can facilitate the understanding or refactoring of existing service specifications.

MDD tools best support the process for contract development; however, they may not fit all the criteria for service development. *E.g.*, the Visual Paradigm meets almost all criteria except input data validation, while IBM RSA does not meet this criterion besides does not generate client code and neither UI documentation.

The tools provided by the Swagger Project lack for one that generates the contract from a model, *i.e.*, a more abstract representation. The initiative could handle this feature in a new tool or researchers could study the combination of existing tools and the Swagger project tools to support it.

The tools based on annotations or decorators do not generate client code nor Swagger specification, which they could support in new versions. They could also support the creation of the annotations out of the code in order to reduce this coupling (Criterion 6).

The automatic validation of input data against the types and definitions presented in the Swagger specification is a feature that should be better explored in future tools. The automatic validation of input data is a crucial factor due to security issues (Sudhakar, 2011). Also, we have identified a lack of tools that provide support to Swagger 3.0 (OpenAPI). Furthermore, considering the few tools that support it, there is still a lack of examples illustrating their usage in-

volving specific Swagger 3.0 features.

As other future work, we propose empirical evaluation of the contract-first tools to the process for contract specification (Figure 1) and investigate how contract-last tools could also support some of that process activities. In other words, how the tools may speed up and improve the quality of contract specification.

This work was a survey based on documentation analysis; so, as future work, we intend to check our findings by performing case studies using the tools in practice.

Another direction of future work would be evaluate the documentation as a whole, and not only the contract specification, and its use in the development lifecycle. It would include also other tools like Postman[22].

ACKNOWLEDGEMENT

This project was executed under the Brazilian National Petroleum Agency (ANP) R&D incentive regulatory framework.

REFERENCES

Cao, H., Falleri, J.-R., and Blanc, X. (2017). Automated Generation of REST API Specification from Plain HTML Documentation. In *International Conference on Service-Oriented Computing*, pages 453–461. Springer.

De, B. (2017). *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization - API Documentation*, pages 59–80. Apress.

Espinha, T., Zaidman, A., and Gross, H.-G. (2014). Web API growing pains: Stories from client developers and their code. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 84–93.

Fielding, R. T. and Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation.

Haupt, F., Leymann, F., Scherer, A., and Vukojevic-Haupt, K. (2017). A framework for the structural analysis of REST APIs. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 55–58.

Jethanandani, M. (2017). Yang, netconf, restconf: What is this all about and how is it used for multi-layer networks. In *Optical Fiber Communication Conference*, pages W1D–1. Optical Society of America.

Josuttis, N. M. (2007). *SOA in practice: the art of distributed system design*. "'Reilly Media, Inc".

Li, L. and Chou, W. (2011). Design and describe REST API without violating REST: A Petri net based approach. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 508–515. IEEE.

Newman, S. (2015). *Building microservices: designing fine-grained systems*. "O'Reilly Media, Inc.".

OpenAPI initiative (2018). OpenAPI Specification. https://github.com/oai/openapi-specification/blob/master/versions/3.0.1.md.

Pastor, O., España, S., Panach, J. I., and Aquino, N. (2008). Model-driven development. *Informatik-Spektrum*, 31(5):394–407.

Preibisch, S. (2018). API Design. In *API Development*, pages 41–60. Springer.

Pritchard, S. W., Malekian, R., Hancke, G. P., and Abu-Mahfouz, A. M. (2017). Improving northbound interface communication in SDWSN. In *I Annual Conference of the IEEE Industrial Electronics Society*, pages 8361–8366. IEEE.

Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE software*, 26(6):27–34.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education.

Santos, R., Pereira, I., and Azevedo, I. (2019). Dynamic Generation of Documentation, Code, and Tests for a Digital Marketing Platform's API. In *Code Generation, Analysis Tools, and Testing for Quality*, pages 1–35. IGI Global.

Scherer, A. (2016). Description languages for REST APIs-state of the art, comparison, and transformation. Master's thesis, University of Stuttgart, Germany.

Sharan, K. (2015). Model-view-controller pattern. In *Learn JavaFX 8*, pages 419–434. Springer.

Sudhakar, A. (2011). Techniques for securing rest. *CA Technology Exchange*, 1:32–40.

Surwase, V. (2016). REST API Modeling Languages-A Developer's Perspective. *International Journal of Science Technology & Engineering*, 2(10):634–637.

Tsouroplis, R., Petychakis, M., Alvertis, I., Biliri, E., and Askounis, D. (2015). Community-based API Builder to manage APIs and their connections with Cloud-based Services. In *CAiSE Forum*, pages 17–23.

Varanasi, B. and Belida, S. (2015). *Introduction to REST*, pages 1–13. Apress, Berkeley, CA.

Varga, E. (2016). Documenting REST APIs. In *Creating Maintainable APIs*, pages 143–157. Springer.

Wideberg, R. (2015). Restful services in an enterprise environment - a comparative case study of specification formats and HATEOAS. Master's thesis, Royal Institute of Technology, Stockholm, Sweden.

[22] https://www.postman.com/