Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 219 (2023) 537–545

Procedia Computer Science

www.elsevier.com/locate/procedia

CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN – International Conference on Project MANagement / HCist – International Conference on Health and Social Care Information Systems and Technologies 2022

RapiTest: Continuous Black-Box Testing of RESTful Web APIs

Duarte Felício[a], José Simão[a,b], Nuno Datia[a,c,*]

[a]*FIT/ISEL, Lisbon School of Engineering, Politécnico de Lisboa, Portugal.*
[b]*INESC-ID Lisboa, Portugal.*
[c]*NOVA LINCS, NOVA School of Science and Technology, Portugal.*

Abstract

When it comes to web services, RESTful web APIs have become the *de facto standard* since 2000. Those APIs expose back-end data, so it is crucial that they are robust, secure, and reliable to keep sensitive data protected. Although existing tools for automating APIs test case generation have shown significant potential, they are limited in their applicability since they focus solely on random inputs through fuzzing. Using only API specifications, it is impractical to describe personalized and specific test case workflows. This paper introduces RapiTest, an open-source continuous black-box testing application for RESTful web APIs. It takes advantage of the API specification to automatically generate tests, but also makes use of a new DSL named Test Specification Language (TSL), to create rich test cases. The RapiTest web application allows the setup of several predefined verifications, regarding security and correctness of the responses, while running the tests at regular intervals, such as every 24 hours. In this way, the API can be monitored continuously to ensure it is running correctly.

Keywords: DSL; REST; API; Web Application; Black-box Testing; Reliability; System Integration.

* Corresponding author.
E-mail address: nuno.datia@isel.pt

1877-0509
10.1016/j.procs.2023.01.322

1. Introduction

The term Application Programming Interface (or API) consists of a software interface with the purpose of facilitating the communication between different components or systems. The component that invokes the API operation does not have to be aware of the other system's internal procedures, only how to call the intended functionality and how to structure the results. ProgrammableWeb [1], one of the most popular API directories, now lists over twenty-four thousand APIs. Such APIs are based on different protocols or architectural styles, mainly Representational State Transfer (REST) and SOAP However, taking into consideration the most used websites of modern times [8], REST has become the standard way of supplying a service through the web.

REST is an architectural style introduced by Roy Fielding in the year 2000 [9]. RESTful web APIs (i.e., APIs that adhere to the REST constraints) are generally composed of a multitude of different endpoints, each pertaining to certain data. Through these endpoints the user can carry out certain actions such as creating, reading, updating, or deleting (CRUD) data (e.g., a tweet on twitter). By its very nature, APIs effectively allow users to manipulate data of the service they are using, which if careless, can lead to security breaches, possibly resulting in stolen, modified or even completely deleted sensitive data. This has resulted in an increased demand to test APIs and guarantee that such malicious attacks are minimized.

RESTful APIs, due to their natural separation of functionality between different endpoints, are often deployed in distinct execution environments, such as containers, where you can easily and dynamically allocate and deallocate resources depending on which services are used at a certain point in time [10]. However, these microservices architectures make it difficult to perform tests on an API following a white-box logic, where you have access to the source code. A black-box approach, where you don't have access to the source code, relies only on the available interface provided to access the API. Additionally, the source code is often unavailable, so a black-box is the only viable option. Black-box methods rely heavily on the API's specification, which is often done using the OpenAPI Specification (OAS) [2]. By describing the API in such detail, humans and computers can discover and comprehend the capabilities of the service without having to access source code. Thus, black-box techniques automatically derive tests from it. These tests then use random and default values, data from dictionary files or even data from previous responses from the API with the intent of finding certain combinations of values that prompt certain responses [11, 12, 13, 14].

Although tools with a black-box approach show promising results in error detection, due to the exclusive use of the API specification, they present major limitations when performing other types of tests, such as tests for: 1) validation of the response and its content, 2) workflows between different endpoints, 3) load and latency, and 4) identification of security vulnerabilities. These tools focus mostly, or even exclusively, on detecting codes 5XX, and do not validate correct behavior under correct conditions.

After a brief overview of the related work in Section 2, Section 3 presents a new DSL named Test Specification Language (TSL), which allows users to define personalized and specific tests for specific problems, with the purpose of validating correct functionality. To complement it, we also present in Section 4 the RapiTest tool, an open-source continuous black-box testing web application for RESTful web APIs. It consists of a fully-fledged single page web application that takes full advantage of automatically generated test cases through the specification, while also supporting TSL files for a more personalized test execution. RapiTest, and by extension TSL, aim to allow users with limited technical knowledge of the domain to be able to validate and continuously monitor different APIs. Section 5 shows some of the use cases analyzed and Section 6 concludes the paper.

2. Related Work

Most proposed black-box approaches for testing RESTful APIs focus solely on the generation of test cases based on the API specification, often in OAS format, most of them relying on fuzzing for the generation of random test data with the goal of finding bugs [11, 12, 13, 14, 19]. More generally, Marculescu et al. used a search-based software testing approach to automate test data generation in a fast and efficient manner [21]. In their study, a fault classification taxonomy was developed, identifying faults under four major categories: (*i*) Configuration and Execution Faults – these faults are related to the setup of web frameworks, mocking structures, or other external systems; (*ii*) Schema Conflicts – these are faults resulting from mismatches between the OpenAPI schema and the analyzed system; (*iii*) Data Integrity Faults – initialization, data consistency, database injection faults; and (*iv*) Faults in the Business Logic – faults that reflect problems in the source code dealing with business logic.

Ehsan et al. [20] used a systematic literature review to answer some research questions in this field, namely "*What are the main challenges in generating unit tests for RESTful APIs*?". They conclude that security issues, non-compliance to description document standards, inconsistent descriptive documentation of the API and complex input types are the major challenges tackled across the current state of the art.

RestTestGen [15] is a recent tool that attempts to programmatically assert dependencies between different operations through the analysis of the specification, in order to better understand and test the API with valid values, e.g., one request returns a list of identifiers, and another returns the specific object associated with the identifier.

RESTest [16] attempts to generate valid tests more efficiently to globally used APIs. Martin-Lopez et al., the creators of *RESTest*, noted the presence of inter-parameter dependency on a major part of these APIs [17]. These dependencies consist of two or more parameters in a request where their value is directly correlated, e.g., on Youtube's API, one request has the parameter *videoDefinition* which if present, forces another parameter, *type*, to have the value: *video*. This ultimately led to more intelligent fuzzing tests, avoiding inter-parameter errors.

bBOXRT is a black-box tool that, taking advantage of a specification in OAS, generates multiple tests in order to try to find errors, proposed by Laranjeiro et al. [11]. This tool has a strong focus on Fuzzing tests, introducing 57 different have mutations on input values, applicable to numeric type values, strings, booleans, dates, times, arrays, etc. Execution starts by trying to generate responses valid (2XX codes), where the returned results are reused for future testing. On the other hand, responses with error code (4XX or 5XX) are retried with different mutations or completely discarded. The tool does not support any kind of response validation and requires manual intervention for the analysis of test results.

Several commercial tools exist for the generation of tests for RESTful APIs such as Postman [3], SauceLabs [4] and Soap [5]. These tools, while offering a wide variety of test cases, such as the possibility to do custom verifications, stress tests and even workflows, they however, offer little to no automation in the generation of such tests, requiring mostly a manual approach.

These approaches demonstrate promising results in detecting bugs. However, they fall short of validating the service's correct functionality. For instance, most of the tools only verify the response status code to check if any returned a code of 5XX, failing to verify if the tests should have returned a certain code given its input. Since the specification only mentions that certain requests can return certain status codes, one cannot know under which circumstances those codes should be returned, e.g., with what input data should the server respond with the response code 400. These limitations led to the creation of a new domain specific language (named TSL and presented in Section 3) to organize and specify tests to then automate them.

3. Test Specification Language - TSL

RESTful web services are now a part of several systems, with many human resources being used do develop and test them. Automation plays a key role in testing these systems. To correctly test and validate an API some specific types of tests were taken into consideration, namely validation tests, workflow tests and stress.

Validation tests mostly cover tests regarding the body payload and status codes to verify that the response was adequate. However, checks exclusively on the response code should not be the only method of validating the correct functioning of the API, as responses often have a body, which must also be validated. The tool must verify if the received schema is the same as the one specified.

Workflow tests try to simulate real use cases allowing for the chain of multiple operations such as create, read, update, and delete (CRUD). This scenario has as an additional challenge because in many cases it is necessary to store values of certain responses to future requests, usually primary keys of certain objects, necessary to perform specific operations on that object.

Stress tests attempts to simulate high traffic situations to evaluate the response times. Stress tests play a key role in identifying speed, stability, and reliability of an API. This type of tests, when used since the start of development, help to avoid waiting times that are too long or even, in the worst case, a total crash. These tests aim to find the point failure of an API under extremely high load during a certain period. In this way it is possible to identify specific areas of the API where it is necessary to increase resources so that the API continues to function properly. These tests are mainly based on one parameter, the time between a request the reply. Taking advantage of multiple threads, numerous requests are sent to the API to evaluate its performance under stress. They can even be performed on workflows to try to understand which of the requests within a workflow would benefit from more resources.

The DSL Test Specification Language (or TSL), aims to support all the above-mentioned tests, defining an easy-to-read file, by both humans and machines, that then can be supplied to a service so that its execution is automated. TSL files, constructed as YAML files, define five major features: a workflow, a stress test and a test with verifications and their results. A workflow consists of a chain of one or more tests while a test consists of an HTTP request with some verifications to the response which produce results. One workflow can also have one, and only one, stress test which defines the number of times this workflow will be executed. The model supporting the TSL files can be seen in Fig. 1, with the five major features and the relations between them.

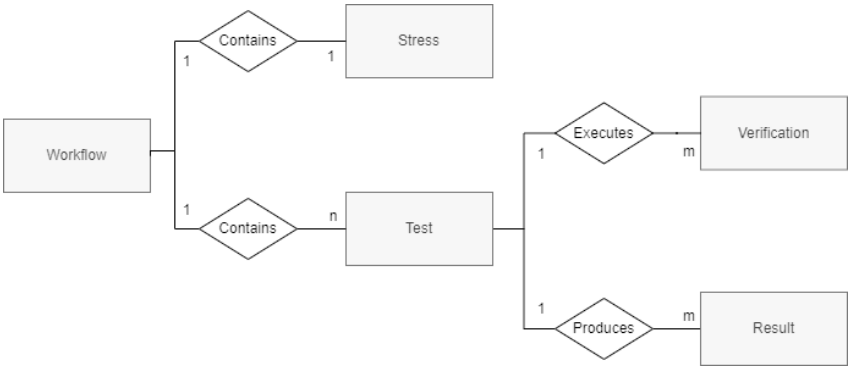

Fig. 1. TSL model.

An example of a TSL file showing the major features can be seen in Listing 1. It features one workflow, representing half of a CRUD operation, namely only the create and read, with one stress element and two tests. Both tests also feature some validations.

```
- WorkflowID: cr_pet
  Stress:
    Count: 10
    Threads: 2
    Delay: 0
  Tests:
  - TestID: createPet
    Server: "https://petstore3.io/api/v3"
    Path: "/pet"
    Method: Post
    Headers:
      - Content-Type:application/json
      - Accept:application/json
    Body: "$ref/dictionary/petExample"
    Retain:
      - petId#$.id
    Verifications:
      - Code: 200
        Schema: "$ref/definitions/Pet"
  - TestID: readPet
    Server: "https://petstore3.io/api/v3"
    Path: "/pet/{petId}"
    Method: Get
    Headers:
      - Accept:application/xml
    Verifications:
      - Code: 200
        Contains: id
        Count: doggie#1
        Schema: "$ref/definitions/Pet"
        Match: /Pet/name#doggie
        Custom: ["CustomVerification.dll"]
```

Listing. 1. Test Specification Language example.

3.1. Workflow

Every TSL file must include at least one workflow, which is identified through a unique string, *WorkflowID*.

```
- WorkflowID: cr_pet
```

Every workflow can then have one or more tests, and optionally, one stress test. All the tests inside the workflow are assumed to be executed sequentially, since the TSL specification allows situations where the output of one test influences the input of the other.

3.1.1. Stress

Optionally, one workflow can have one stress element that can be used as a throttling mechanism, or to simulate high traffic to a service to test and validate its robustness. The stress definition has 3 fields, namely:

- *Count* – defines the number of times the complete workflow will be executed;
- *Threads* – defines the number of threads by which `Count` will be divided (For example, `Count:` 10 and `Threads:` 2 means each thread will execute the workflow 5 times). This is useful to simulate a real-life scenario where distinct clients make concurrent requests to the API;
- *Delay* – defines the delay in milliseconds between every full execution, which can be useful to prevent errors of too many executions.

3.1.2. Test

You can define one or more tests within each workflow and every test is identified through a string that is assumed to be unique across all tests, *TestID*.

```
- TestID: createPet
```

It is then followed by three mandatory fields for a successful HTTP request, namely the server, the path and the method.

```
Server: "https://petstore3.swagger.io/api/v3"
Path: "/pet"
Method: Post
```

Headers and query parameters can also be defined following a standard key/value structure. The body data can be defined directly on the TSL file, however they can sometimes be large which would hurt the clarity and readability of the file. You can instead create an auxiliary text file (dictionary file) which contains the actual body and a unique identifier within the dictionary, leaving only the reference to said identifier on the TSL file.

```
Body: "$ref/dictionary/petExample"
```

In some requests, the input is based on the output of a previous request, usually in simple workflows, like create, read, update, and delete. The keyword `Retain` allows the user to retain some information from the response body of the request to then be used in other tests of the same workflow. For instance, the value present at the JSON path `$.id` will be retained with the identifier *petId* which can then be used in following tests.

```
Retain:
    - petId#$.id
...
Path: "/pet/{petId}"
```

Each test can have multiple verifications, only the Code verification is mandatory. Currently 6 different verifications are supported, as described in Table 1.

Table 1. Supported Verifications.

Name	Mandatory	Input Type	Description
Code	Yes	Integer	Response code matches the given code
Contains	No	String	Response body contains the given string
Count	No	String#Integer	Response body contains given string # times
Schema	No	String	Response body matches the given schema
Match	No	StringPath#String	Response body matches the given value present in the StringPath
Custom	No	String	Runs Custom verifications given by the user

The schema verification can be supplied directly, or through reference to the dictionary file or to any schema present in the supplied OAS. If users wish to not be limited to the verifications supported by the application, they can create their own custom verifications by implementing an interface, creating the *DLL* file, and then supplying it. More details about the TSL files can be found available on the supplemental material provided with this paper [6].

4. RapiTest

In this section we explain the basic workflow of RapiTest, depicted in Fig. 2, starting from the user input up until the app's output.

A. Represents the minimum information required provided by the user for the tool to work. In this case the name of the API to be tested, which is used to differentiate tests from the same user, although it does not have to be unique, as well as the specification file of the API in OAS format.
B. Represents additional optional files that the user can provide. Namely the TSL files that specify tests to be carried out on the API, a dictionary file that contains extra information needed to carry out the tests specified in the TSL files, and custom verification DLLs, in order to carry out validations not provided natively by the application.
C. Responsible for serialization and verification of all files provided. The final product is a JSON file with all the tests that will be carried out with their respective verifications. This file allows for the easy repetition of the tests every time they are needed.
D. Responsible for deserialization of the file resulting from the previous step and performing all tests and verifications. The final product is a JSON file with the results.

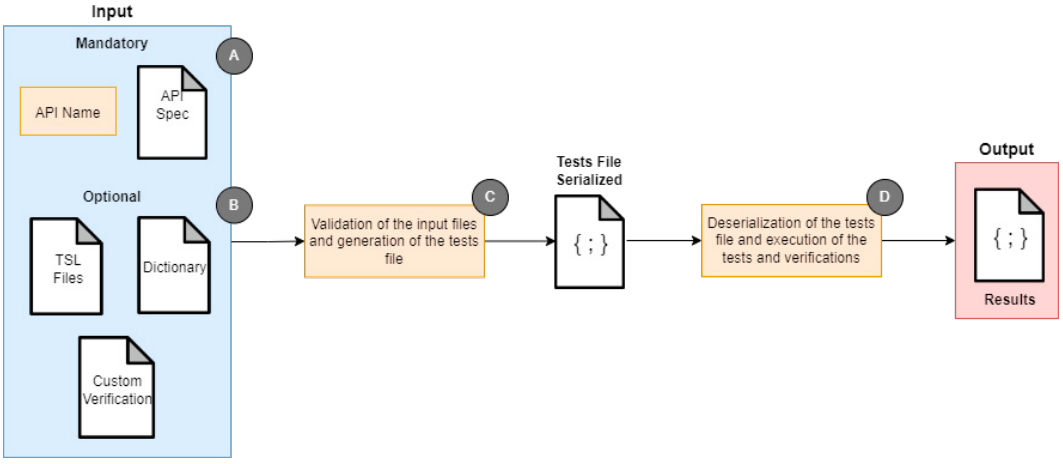

Fig. 2. RapiTest workflow.

Considering the workflow describe in Fig. 2, the project contains four main components:

1. The server, that is responsible for receiving and handling requests according to the application logic;

2. The client, that presents an interactive Web interface to the user which communicates with the server, supporting the steps A and B;
3. The component responsible with validating the input files and generating the tests file, step C;
4. The component in charge of running all tests and verifications, generating the file with the results, step D.

The last two components, to improve server performance and offer a more scalable architecture, were decoupled from the server. The communication between them is made taking advantage of the protocol Advanced Message Queuing Protocol [7]. Doing so, it enables the test of APIs with lengthy delays or errors, without compromising the overall usability.

5. Demonstration and Results

RapiTest was developed in a partnership between Lisbon City council (Câmara Municipal de Lisboa, CML) and Lisbon School of Engineering (ISEL). Since 2017, CML has been using an urban data platform (PGIL) to manage some city' verticals, with many integrated services, some of which are real-time data streams [18]. Today, PGIL has more than 900 integration processes with Web APIs as sources. Thus, to ensure that data sources are compliant with specifications, it is necessary to validate and monitor them. CML was directly involved over the course of the app's development which was demonstrated multiple times to several CML operatives for feedback and guidance purposes. The feedback regarding the user interface was overwhelmingly positive due to its emphasis on simplicity, however, regarding the functionality the users felt it required extensive knowledge of the problem's domain including, the new TSL files. The user interface was enhanced to support the creation of these files to streamline the user experience, requiring less technical knowledge to use the web application. In Fig. 3, we see a screenshot of the user interface for creating TSL files, where a workflow and test have already been created.

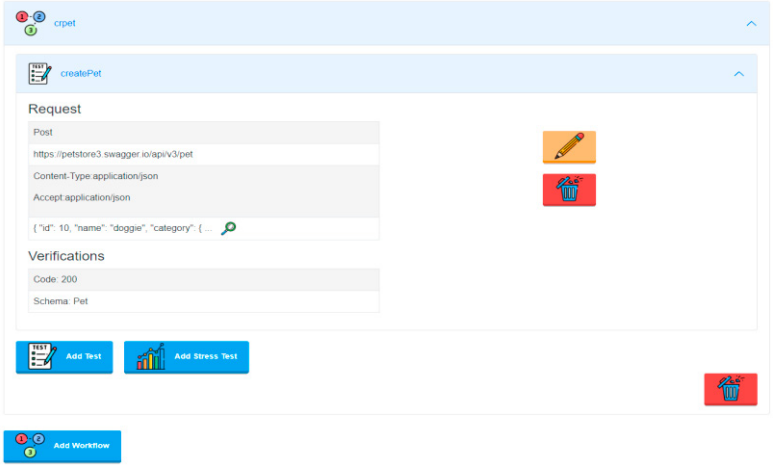

Fig. 3. RapiTest screenshot, TSL construction.

CML made available two APIs for testing the application's potential. Both APIs require a specific API key to access them; this immediately invalidates all approaches that take sole use of the API specification since such private keys are not supplied on the specification. They can however be specified on TSL files to correctly test them. Specific TSL files were then manually created for both APIs and were then supplied and executed through the web application. In both APIs errors were found relating to the schemas specified on the specification. Both schemas didn't comply with the actual response objects due to several issues, namely certain fields were `Null` on the response, but they weren't specified as nullable on the schema. Fig. 4 presents a screenshot of RapiTest, namely the page to view the test's results, where the test executed five validations. Four of them were successful, represented in green, and one of them, the schema validation, wasn't, represented in red. It is also possible to view why the validation was unsuccessful on the right panel analyzing the description. In this case several fields had `Null` value while expecting some other type.

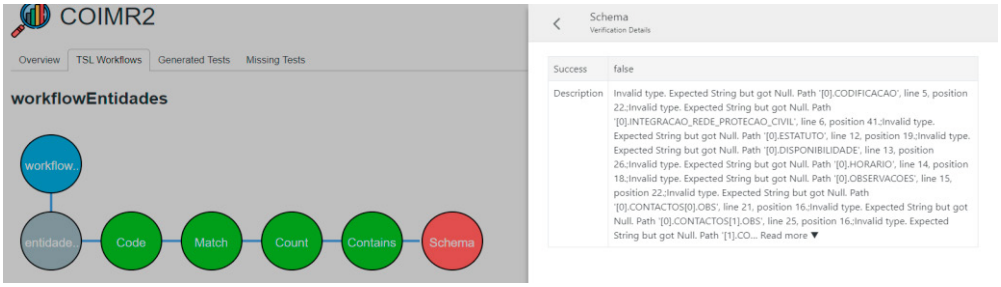

Fig. 4. RapiTest screenshot, with results from a test execution.

6. Conclusion and future work

This paper presents a DSL named Test Specification Language (TSL), which allows users to define personalized and specific tests for specific problems, complemented with RapiTest, an open-source continuous black-box testing web application for RESTful web APIs. TSL can easily be extended with new functionalities, such as support for other types of tests, new validations, new test data generators, among others. RapiTest has already proven useful in the automated detection of errors in API consistency in currently used, real-world, APIs.

For future work, we plan to fully deploy the web application in CML servers to test its usage with many more APIs. We also intend to enrich the app with many more functionalities, such as exporting the results directly to an excel file, edit currently configured tests, support additional payload types in validations other than JSON or XML, as well as implement more validations. Finally, we will also consider a mobile compatible version.

Acknowledgements

This work has been developed in the scope of the “Data in the service of Lisbon” protocol, done in partnership between Lisbon City council and Lisbon School of Engineering.

References

[1] ProgrammableWeb API Directory. http://www.programmableweb.com/ accessed June 2022.

[2] OpenAPI Specification. https://www.openapis.org accessed June 2022

[3] Postman. https://www.getpostman.com accessed June 2022

[4] SauceLabs. https://saucelabs.com/platform/api-testing accessed June 2022.

[5] SOAP. https://www.soapui.org/tools/soapui/ accessed June 2022

[6] “Supplementary material of the paper.” [Online]. Available: https://github.com/ISEL-DEETC/RAPITest

[7] OASIS. “Advanced Message Queuing Protocol”. http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html accessed June 2022

[8] Andy Neumann, Nuno Laranjeiro and Jorge Bernardino. (2021) “An Analysis of Public REST Web Service APIs”, IEEE Transactions on Services Computing, vol. 14, n.º 4, 957–970.

[9] Roy Fielding & Architectural Styles. (2000) “The design of network-based software architectures”, Doctoral Dissertation, School of Information and Computer Science.

[10] Claus Pahl & Pooyan Jamshidi. (2016) “Microservices: A Systematic Mapping Study.”, on CLOSER (1), pages 137–146.

[11] Nuno Laranjeiro, João Agnelo and Jorge Bernardino. (2021) “A Black Box Tool for Robustness Testing of REST Services”, IEEE Access, vol. 9, pages 24 738–24 754.

[12] Vaggelis Atlidakis, Patrice Godefroid and Marina Polishchuk. (2019) “Restler: Stateful rest api fuzzing”, IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pages 748–758.

[13] Andrea Arcuri. 2021. Automated Blackbox and Whitebox Testing of RESTful APIs With EvoMaster. IEEE Software (2021).

[14] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo and Jordi Cabot. (2018) “Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach”. In EDOC. 181–190.

[15] Emanuele Viglianisi, Michael Dallago and Mariano Ceccato. (2020) “RestTestGen: automated black-box testing of RESTful APIs”, IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, 2020, pages 142–152.

[16] Alberto Martin-Lopez, Sergio Segura and Antonio Ruiz-Cortés. (2021) “RESTest: automated black-box testing of RESTful web APIs”, on Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 682–685.

[17] Alberto Martin-Lopez, Sergio Segura and Antonio Ruiz-Cortés. (2019) “A catalogue of inter-parameter dependencies in RESTful web APIs”, on International Conference on Service-Oriented Computing, Springer, pages 399–414.

[18] Serrador, A.; Tremoceiro, J.; Cota, N.; Cruz, N. and Datia, N. (2018) “iLX - A Success Case in Public Tender Methodology”, CENTERIS 2018 - Conference on ENTERprise Information Systems / ProjMAN 2018 - International Conference on Project MANagement / HCIST 2018 - International Conference on Health and Social Care Information Systems and Technologies.

[19] D. Corradini, A. Zampieri, M. Pasqua and M. Ceccato. (2021) “Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs”, 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 226-236.

[20] Ehsan, A.; Abuhaliqa, M.A.M.E.; Catal, C. and Mishra, D. (2022) RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions”. Appl. Sci., 12, 4369. https://doi.org/10.3390/app12094369

[21] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. (2022) “On the Faults Found in REST APIs by Automated Test Generation”. ACM Trans. Softw. Eng. Methodol. 31, 3, Article 41, 43 pages. https://doi.org/10.1145/3491038